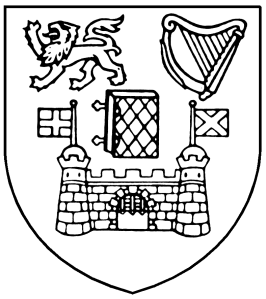
The SMG DSM System

Enabling Shared Memory for the Grid

A thesis submitted for the degree of
Doctor of Philosophy

at

University of Dublin, Trinity College



Title:	**The SMG DSM System** *Enabling Shared Memory for the Grid*
Presented by:	**John Paul Ryan**
School:	**Computer Science and Statistics**
Supervisor:	**Dr. Brian Coghlan**
Date:	**June 2007**

Declaration

This thesis has not been submitted as an exercise for a degree at any other University. Except where otherwise stated, the work described herein has been carried out by the author alone.
I agree that the Library in Trinity College may lend or copy this thesis upon request.

John P. Ryan
June 2007

Acknowledgements

Firstly, I would like to offer my my sincerest gratitude and thanks to my supervisor Dr. Brian Coghlan, for his support and input to this thesis over the past few years. He was always willing to offer advice and debate even the minutest detail of this work.
My grateful thanks to Professor John G. Byrne of the Department of Computer Science for my initial postgraduate funding, and the European Union and Science Foundation Ireland (SFI) for their funding through the Crossgrid and WebCom-G projects. Also my thanks to the Trinity Centre for High Performance Computing (TCHPC) [1] and Irish Centre for High-End Computing (ICHEC) [2] for permission to use their computing facilities and for the support of their personnel, in particular Jimmy Tang of TCHPC, in helping to resolve problems.
Within the Dept. of Computer Science, Trinity College Dublin, I would also like to thank various members (past & present) of the departmental admin staff, the hardware technicians and systems support staff for their time, that on occasions that was above and beyond the call of duty.
I would also like to thank all members of the research group of which I am a member - Computer Architecture & Grid Research Group (CAG). Many people went out of their way to help with the numerous problems, some in particular spending aeons solving my problems. My thanks to (in no particular order): Eamonn Kenny and Geoff Quigley who have been closest (literally) in times of need (apologies for all those annoying questions); Jonathan Dukes, David O' Callaghan and Stuart Kenny for help with the myriad of issues that required your input when I shared an office with you; and under the guise of Grid-Ireland Help, John Walsh and Stephen Childs for their support with issues arising from the use of the MPI and Grid software.
Outside the confines of Trinity I would like to thank friends and in particular my housemates John and Thomas Conway for tolerating my bad moods, procrastinations, and highly irregular working hours over the past few years. Lastly I would like to thank my family for their encouragement and support over the years, these include my sisters: Natasha, Katie (I think our shared experience in completing a research degree was far more beneficial to me!), Fiona, Monica, Pamela, and my parents, Mary and Bill.
Again, thank you all (and anyone that I've omitted).

Summary

Parallel computing has taken the first steps along its next evolutionary route: computational grids are now a reality. Success, however, depends not only on the tools available, but allowing the knowledge base that currently exists in constructing parallel applications to be employed by those same engineers, scientists and other groups that will make use of this new platform. One basic requirement will be the availability of familiar programming methods and paradigms. Message passing has a natural affinity towards wide area network computing and requires little effort, if any, in order to grid enable. Implementations exist for the most common operating systems and hardware architectures. The shared memory paradigm is a different question as the necessary physical memory resources cannot be readily shared among distributed processors.
Distributed Shared Memory (DSM) has being promoted since the 1990s as a method to execute shared memory programs on distributed machines. In this thesis the barriers to a successful DSM implementation are highlighted; some of these include: support for heterogeneous environments, overall application performance, and absence of a standard programming interface. It will show how the latter problem could be resolved by providing a source-to-source compiler that takes as input a shared-memory application written using a standard such as OpenMP, where the target is the DSM API. If implemented, many applications written for a shared memory setting would require minimal changes in order to execute in a grid setting.
The approach taken involving DSM is less efficient than message passing, but this is mitigated by using the DSM to help identify the areas of a program that account for the most overhead, and in a localised fashion and with minimal effort on behalf of the programmer convert these areas of shared memory code to the more efficient message passing code; this process can be done in an incremental fashion, thereby allowing for the exertion of resources only where necessary. In order to achieve this a hybrid shared-memory/message passing environment was developed, in addition to tools to direct the 'hybridisation'.
Since grids are coming into the mainstream, it would be most beneficial if the most natural parallel computing paradigm, shared memory, was supported. At this time the most obvious way to do this is to construct a DSM run-time system for the grid, that provides various levels of user configuration and allows user-directed optimisations to mask the latencies between physically distributed nodes.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

Parallel computing is an enabling approach that allows the execution of compute intensive applications in a timelier manner. This is ultimately achieved by dividing work among a number of computer processors. Shared memory is the favoured environment for constructing these parallel applications, mainly due to its single address space, which dramatically reduces the programming effort required compared with the use of other programming models. Due to technical, and by association, financial, reasons with the underlying shared-memory hardware architectures, the scalability of these applications have been limited to tens, and rarely hundreds, of tightly-coupled processors.

In the early 1990s cluster computing started to become more prevalent as an alternative architecture. Clusters, usually consisting of a group of loosely coupled commodity machines, potentially consisting of thousands of nodes interconnected with a high-bandwidth, low-latency interconnect, provide a means of constructing large scale, low-cost high performance computers. Similar technical (and financial) reasons have resulted in only a small proportion of clusters supporting a common shared memory.

With the advent of the Internet and wide area computing, the natural progression was toward the sharing of resources, and so computational grids evolved. One important question is how to efficiently exploit this new platform.

A computational grid will consist of a number of physically distributed sites, potentially consisting of a variety of different hardware architectures. Data intensive (input/output) applications would be particularly suited to a computational grid, while coarsely grained (see Section 2.5) applications such as *seti@home* [3] and *folding@home* have already demonstrated that wide area computing is feasible. Such applications can execute in parallel across a number of distributed sites when the application data is co-located. It is vital to overall performance that these applications produce little inter-site traffic, while intra-site traffic is of a lesser concern as it impinges less on performance. Sites can be selected according to the availability of suitable hardware.

The shared memory paradigm natural to multiprocessors and some clusters is not available on the grid (outside the work of this thesis), yet the apparently more efficient message passing paradigm introduces excessive programming burdens. In order to em-

ploy a shared memory style on a grid, a virtual implementation of a shared memory subsystem (and supporting management layer), known as Distributed Shared Memory (DSM) would be required, allowing a programmer to develop an application in a shared memory style, yet execute it in a grid environment. This approach would need to be flexible in order to allow dynamic optimisations, especially in the communication patterns of the application (otherwise performance would be very poor due to the excessive latencies involved on grids). Where an application employing the DSM exhibits excessive communication in certain locations, then these areas might employ the more efficient message passing style.
Ideally the DSM would be constructed using only standard libraries, and no special compiler would be required for the compilation of the user's application code. A user's grid application could then be constructed using a shared memory API, or where practical a parallelising compiler could be used to target the DSM API. Optimisations could be obtained from employing profiling information which can be gathered from a grid information system. The important point is that only an optimised, not a naive, approach has any possibility of being worthwhile in a grid environment.

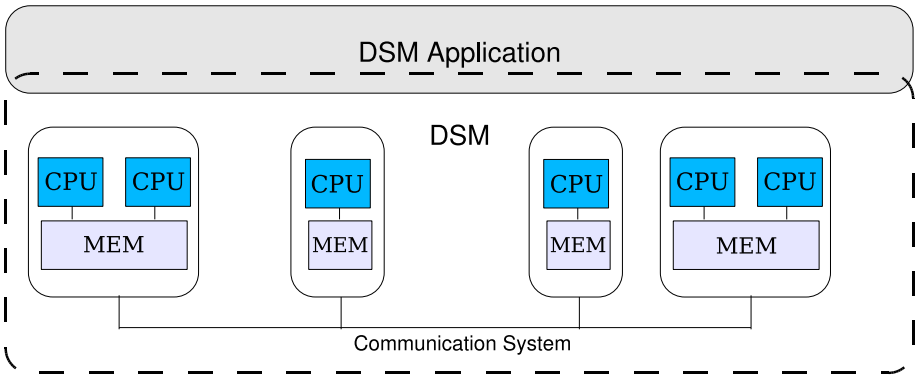


Figure 1.1: *DSM Concept*

1.1 Research Goals

The main goal of this thesis is to construct and evaluate the use of a grid-enabled DSM system, that will allow easy porting of explicit shared memory codes using a familiar and standard API. Assuming this can be achieved, the application can then be executed in a distributed environment such as a grid.
The areas of the code that exhibit high communication patterns might be converted, or 'hybridised', to the more efficient message passing style, while also allowing multi-threaded user applications to take advantage of real shared memory where available. The hybridisation of such an application into the mixed mode form might be done in an incremental fashion as the law of diminishing returns dictates, i.e. cost benefits will probably diminish with the level of hybridisation.

Within this ambitious goal there are a significant number of issues that need to be first addressed. Some of these include:

- *Granularity*: What will be the granularity of the grid applications? We know that fine grain sharing of data is unsuitable for grids, and that very coarse grain applications can work efficiently. How will medium granularity applications perform? Different types of application need to be investigated. Are multi-granular applications suitable?

- *Algorithms*: Are there algorithms that can be altered/developed in order to mask the obvious drawbacks associated with multiple-site parallel computing in grids? Multi-threaded user applications might be utilised to hide communication latencies, so that communication and computation is overlapped.

- *DSM*: What type of DSM system would be most suited to the grid? The choice of consistency model, coherency protocol, shared data granularity, DSM algorithm and the overall DSM management are all factors in need of consideration. The DSM must present the programmer with an intuitive API in order for it to gain acceptance.

- *Heterogeneity*: It is highly likely that a Grid will consist of a heterogeneous collection of architectures and platforms. The sharing of application data will prove problematic as different architectures may have different native data representations. To make this appear transparent will require a tremendous effort.

- *Performance*: What will be the performance of grid applications? The performance may be adversely effected by the use of a DSM, but how significant will this be? Can the hybridisation process efficiently reduce the performance difference between an application constructed using a DSM, and the corresponding algorithm implemented using message passing methods?

- *Compiler target*: Can the DSM system form the basis of a target for a compiler[1], so that an existing open shared memory programming standard, such as OpenMP [4], may be supported? Additionally, what are the differences in the requirements for a DSM that will be used directly by an application developer, and one that can provide the target for an OpenMP compiler? Will the platform provided by the DSM be suitable for such a task?

- *Programming burden*: How will the programming burden be effected in relation to the increased complexity of a grid environment, and using multiple programming paradigms in one application? Can the effort expended be offset with an increase in performance? Programming burden will have to be quantified; the burden is different for different paradigms; should candidate metrics include the number of lines of code, programmer man-hours, or a function of both?

[1] This compiler could take the form of a simple source-to-source translator

- *Feasibility*: Is the use of distributed shared memory in a grid environment worthwhile? In order for this to be so the loss in performance relative to the most efficient message passing version must be acceptable in return for the ease of application development. This may only occur after some 'hybridisation' has been performed.
- *Incremental Hybridisation*: Is a process of incrementally hybridising an application realistic? Is the time spent in analysing the application and hybridising it commensurate with any performance benefits returned? This will be influenced by the answers to the two previous questions.

As depicted in Figure 1.2, the primary goals of this thesis are as follows:

i. The development of an efficient distributed shared memory system that will allow distributed computing resources to appear to be a unified computing system, that is highly portable through the utilisation of portable and non-propriety libraries. No support should have to be supplied by way of modifications to the operating system. The system should not need specialised support from memory management or communication hardware. The DSM should be able to be targeted by a standards compliant compiler such as an OpenMP compiler.

ii. Integration of the DSM with an information and monitoring system, allowing for the acquisition of information for directing the hybridisation of the user application. This information must allow for the targeting of the areas of the application that influence performance the most [5].

iii. Evaluation of the hybridisation process through the utilisation of test cases to develop proper models of applications that are suitable for execution across multiple sites on a computational grid. Applications that are not executed across multiple sites are not of interest. A metric should be derived that illustrates the potential benefits that will accrue from the incremental hybridisation approach. The user should be provided with an intuitive interface where the hybridisation process can be directed and observed.

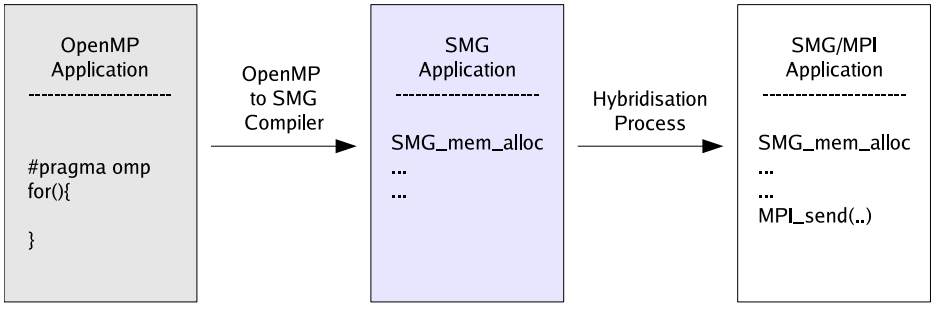


Figure 1.2: *Enabling efficient shared memory on the grid*

1.2 Document Structure

Chapter 2 gives an overview of the basic theory of parallel computing and current hardware implementations. An insight is given into the factors that limit the potential speedup of an application that can be parallelised, with a particular emphasis on the two programming models relating to this project, namely message passing and distributed shared memory. The chapter concludes with a synopsis of synchronisation and other considerations that might be particularly relevant to execution on a computational grid.

Chapter 3 delves into the motivation of this thesis, focusing on the theory explored in the previous chapter and the extrapolation of parallel computing architectures and programming models. We explore the extent to which parallel programming is feasible on the grid, and the justification of going to such parallel computing extremes.

Chapter 4 is devoted entirely to the fundamentals of Distributed Shared Memory (DSM), which starts by providing an overview of how the principle of shared memory programming was expanded to support distributed environments. The unit of data sharing or 'granularity' is discussed in relation to the effect it causes on performance and the design of the overall system. The challenge of keeping shared data consistent across distributed nodes is explored, followed by a description of coherency protocols necessary to achieve this. Algorithms for the management of the overall DSM system are also highlighted. A review of previous DSM implementations with particular reference to the topics above concludes this chapter.

Chapter 5 presents an overview on all the parallel and distributed technologies that are applicable to this project. The area of message passing is reviewed, and how an understanding of network protocols is an important consideration. A survey of the salient issues of current message passing standards is given in addition to a brief overview of the notable features of current implementations. An introduction to information and system monitoring requirements is also provided. The issues involved with using an information and monitoring system are explored. The chapter includes a broad overview of current systems available, and the reasons for the choice of the information and monitoring system used within this project.

Chapter 6 is the first of three chapters that deals with the actual implementation of the author's DSM, which is called SMG : Shared Memory for Grids. A schema for the user API is proposed, followed by an examination of case studies of other distributed shared memory systems in the context of the attributes and design principles employed, including a review of APIs of other DSM implementations. Potential issues regarding the use of the DSM in a grid setting are examined in relation to the provision of end-user functionality through the defined API. Internal DSM engine workings together with functions that allow the user to alter the internal management of the DSM are highlighted. Side-effects of these functions are also discussed.

Chapter 7 describes the implementation of the aspects relating to the provision of distributed shared memory. API functions for the allocation and freeing of memory are described. This is followed by a description of issues that arise and decisions made. The use of the shared memory allocation functions within user applications follows, along with a description of access to the shared regions in user space. It details the principles concerning the governing consistency model together with the use of synchronisation primitives with respect to shared memory. Consideration for the additional requirements of multi-threaded user applications conclude the chapter.

Chapter 8 describes the section on the design, implementation, and use of system synchronisation functions/primitives.

Chapter 9 elicits how support is provided for the DSM operating in a grid environment and how 'hybridisation' is supported within the DSM and its execution environment. The development of the module used to do this is described, where an independent API is defined in order that in future other information and/or monitoring systems can be used.

Chapter 10 provides a description of the run-time execution of the SMG DSM applications and deals with the experimental evaluation of the SMG system for a trivial setup. The applications used to perform the evaluation are discussed with interesting points/attributes highlighted. A performance metric that the system can be evaluated against is defined, with reasons given for the choice. A brief overview of the testbed is given followed by the results obtained from running the test applications with the DSM system in strict cluster mode (used as a reference). The process of hybridisation of the test examples is documented and the results of running the applications in grid mode concludes the chapter.

Chapter 11 focuses on the outcome of the thesis. The initial goals of the thesis are reviewed. Analysis of results is performed followed by a summary. Finally the thesis concludes with an examination of prospects for future work.

CHAPTER 2

Parallel Computing

Engineering and scientific research increasingly involves computational applications that perform simulations in order gain a better understanding in their field of study. They set out to model natural phenomena in a discrete manner, examples can range from computational fluid dynamics (CFD), that allow for the modelling of fluid flow about objects, such as aeronautical components, to the simulation of protein folding in genetic analysis [6].

A common occurrence in these types of applications are linear algebra operations, such as the evaluation of matrix determinants, multiplication, etc, which are inherently parallel tasks i.e. that contain sub-operations within the overall operation, that can be performed independently of the other sub-operations since there is no dependency between the output of one sub-operation to the input of another (i.e. no data dependency). These sub-jobs can be allocated to multiple processing elements to be performed concurrently, thus allowing for a reduction in the execution time for the overall task. Parallel computing attempts to provide for such circumstances.

Consider a situation as depicted in Figure 2.1(a), where initially there exists an application with parallel components interleaved with serial ones. The application when executed in a serial fashion requires wall clock time of $t_s + t_p$, where t_s is the time taken to complete serial components, and t_p is the time taken to complete the parallel tasks. However, if the parallel components can be evenly distributed among among N processors, as shown in Figure 2.1(b) (for N = 4), and ignoring other overheads such as communication, then a reduction in the overall wall clock time results. This reduction in wall clock time for the execution of the application is governed by a number of factors, the ratio of t_p to t_s being the most obvious. The degree of distribution for the work that can be parallelised (the granularity) is also a prime determinant, as this governs the number of processors (the maximum value of N) for which work can be assigned. The minimum possible time to execute the original job with the parallel components distributed across N processors is:

$$T(N) = t_s + t_p/N \tag{2.1}$$

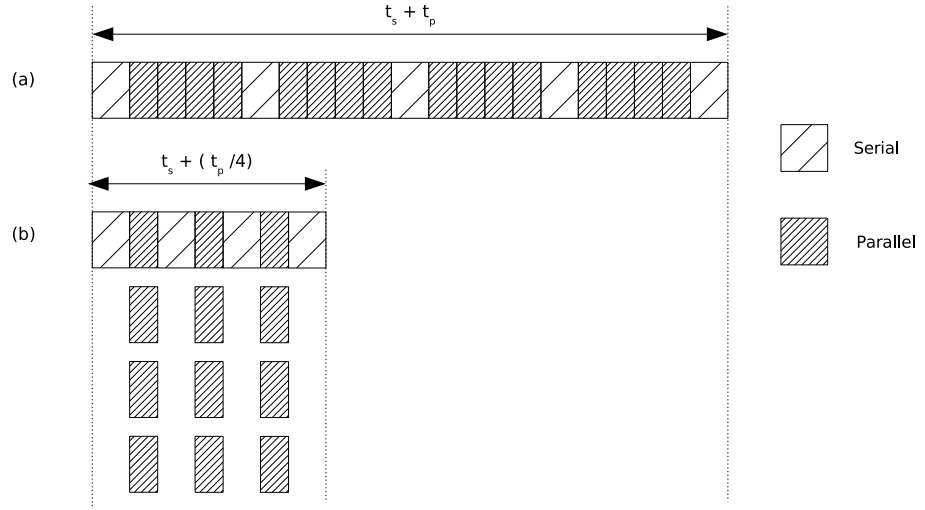


Figure 2.1: *Parallel decomposition*

2.1 A Case for Parallel Processing

Obviously for Equation 2.1 with $N = 1$, the application is executed serially. For $N > 1$ parallelisation occurs. Equation 2.2 represents the maximum speedup that can achieved using parallel processing methods for a given task of fixed size. The limiting term, t_s, the time taken to complete the serial component, dictates that as N $\rightarrow \infty$, the upper bound of speedup can never exceed $\frac{1}{t_S}$ [1].

$$S(N) = \frac{T(1)}{T(N)} = \frac{t_s + t_p}{t_s + t_p/N} \tag{2.2}$$

This equation is often expressed in a normalised fashion, where $t_{total} = t_p + t_s$, $f_s = t_s/t_{total}$, $f_p = t_p/t_{total}$ yielding $f_p = 1 - f_s$. Substituting gives equation 2.3, which is commonly known as *Amdahl's law*. If the serial component of the job is 5%, then according to this equation, the maximum speedup attainable is 20, no matter how many processors are available ($N \rightarrow \infty$). Given this speedup function the efficiency of the system, ϕ, is S_n/n. Thus, by estimating the amount of serial work involved in a task a decision can be made whether the effort expended to parallelise the task can be offset by the performance gain, even if there are numerous processing nodes and ideal hardware is available.

$$S(N)/N = \frac{1}{f_s + (1 - f_s)/N} \tag{2.3}$$

[1] In extreme circumstances this may not hold as the aggregate effect of sophisticated hardware resources, such as processor cache, may allow for minor super-linear speedup

So, in essence parallel computing may be defined as the simultaneous use of multiple compute resources to solve a computational problem [7]. Parallel computing can also be applied to a problem where there are many independent tasks to be accomplished, with an absence of data-dependence between them. In this sense parallel computing employs a divide-and-conquer strategy [8].
Although numerous tasks can be parallelised the effectiveness can vary. The classical levels of job parallelism are:

- **Arithmetic-level parallelism** This level is usually of concern only to logic and compiler designers, and is usually invisible to an application developer. Modern processor instruction sets such as Intel's x86 *MMX* and *SSE* extensions [9], and PowerPC's *Altivec* provide functionality that implements this level of parallelism.

- **Instruction-level parallelism** Involves techniques such as super-scalar processor design, and pipelining, allowing multiple instructions to be executed per clock cycle in a single processor. Again this stage is ofter invisible to a high-level application developer. The processor logic required to implement this functionality (out-of-order execution unit) requires a substantial trade-off in the size of other processor architecture features (e.g. cache).

- **Program-level parallelism** (The term thread-level parallelism is a common synonym). At this level an application is partitioned into different components that can be performed in parallel on separate processors. This form of parallelism is usually found in federated sections of a program and individual iterations of a code loop. The partitioning algorithm, typically implemented by the user, greatly affects the effectiveness.

- **Job-level parallelism** Where one job executes independently of another. They can run at the same time on different processors or time-share a common one. A modern computer system operates in this pseudo-parallel manner where many jobs appear to run concurrently but in effect the operating system (OS) manages the allocation of scarce processor resources among the running processes.

Clearly if we are interested in constructing parallel programs in a high level language we are interested in program-level parallelism. One could construct an application to multiply the two matrices at the arithmetic or instruction level but this would require implementation in a hardware description language such as VHDL or low-level using assembly language. In this thesis, however, we are interested in 'generic' program-level parallelism techniques for parallel applications.
Again ignoring communications overheads, consider the problem of multiplying two matrices A & B, that are of dimensions $n \times m$ & $m \times p$ respectively.

$$\begin{vmatrix} c_{11} & \dots & c_{1p} \\ c_{n1} & \dots & c_{np} \end{vmatrix} = \begin{vmatrix} a_{11} & \dots & a_{1m} \\ a_{n1} & \dots & a_{nm} \end{vmatrix} \begin{vmatrix} b_{11} & \dots & b_{1p} \\ b_{m1} & \dots & b_{mp} \end{vmatrix}$$

The resultant matrix C, has dimensions $n \times p$. Each element ($C_{ij} = \sum_{k=1}^{m} a_{in} b_{nj}$) will require m multiplications and $m-1$ additions per element. If the cost of an addition is t_a and a multiplication is t_m then the time to perform the matrix multiplication in a serial will be $n \times p \times ((t_a \times m) + (t_m \times (m-1)))$. However, there are no data dependencies in the evaluation of the co-factors of the resultant matrix, so they can be evaluated in parallel. Co-factor evaluation is dependent on the calculation of partial sums so it is reasonable to choose to parallelise at the level of the calculation of each co-factor.
If we assume that the incident matrices are of the same dimensions i.e. square ($N \times N$), and assuming that distribution of initial data and collection of partial results across all processors incurs no costs, then a speedup up of $\frac{time for 1 process}{time for N processes}$ is possible, i.e. a speedup of S(N) = N is possible, or speedup is *perfect* (as depicted in Figure 2.2). The calculation of each co-factor can be assigned to an individual processor, which is the finest granularity suitable for high level language implementations.

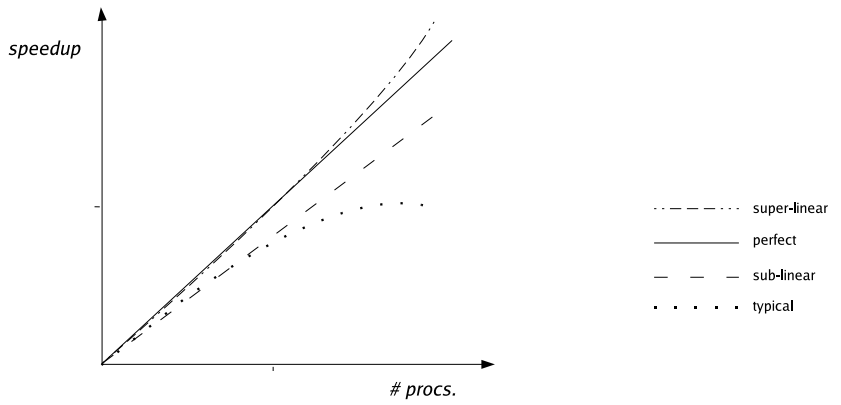


Figure 2.2: *Potential Speedup*

However, our assumption that overheads are negligible is generally incorrect. The actual overhead component, $T_o(N)$, can be significant, and is usually governed by the type of hardware platform employed. Typical performance involves sub-linear speedup, and often severe degradation occurs for a hardware platform beyond a certain point (as depicted with the 'typical' curve in Figure 2.2). This defines the scalability of a parallel system[2].
Parallelisation overheads are the overheads associated in distributing the parallel task data, the collection of results, and any synchronisation costs incurred. This overhead time $T_o(N)$ consists of serial and parallel components (T_{os} & T_{op}) [10]. A more precise expression for obtainable speedup is now given in equation 2.4.

$$S(N) = \frac{t_s + t_p}{t_s + t_o(N) + (1 - t_s)/N} = \frac{1}{t_s + t_{op} + (N \times t_{os}) + (1 - t_s)/N} \tag{2.4}$$

[2]system refers to the combination of the parallel algorithm, implementation and execution platform

It was observed by Gustafson [11], that if the size of the parallel component of a job is scaled up, while the amount of time required for serial work, f_s, remains relatively constant, then it is possible that f_s is no longer such a significant limiting term, i.e. if a given efficiency is required then this can be achieved by scaling the problem size. Such scenarios occur frequently where a larger problem is required to be solved in the same amount of time, but there are additional resources to accomplish this. From this observation Gustafson developed his fixed-time speedup model Equation 2.5.

$$S'_n = \frac{Scaled\text{-}up\ sequential\ time}{Scaled\text{-}up\ parallel\ time} = \frac{f_s + (1 - f_s)N}{1} \tag{2.5}$$

Often the desire to increase the size of a problem is a prime motivation for using parallel computing. A sub-class of parallel applications known as embarrassingly parallel benefit from this simply because computation will scale well, while overheads in general do not. This fact is demonstrated in Table 2.1, which lists the computations required for the multiplication of two square matrices of size n. The workload is of $O(n^3)$, while the overhead is $O(nlogn + n^2)$ [12].
Other models exist such as Isoeffiency [13], Karp-Flatt scaled speed-up [14], and that developed by Sun and Ni [15] that generalises for when the speedup of an application is memory bounded. Ultimately an application should be developed with the largest problem size determined by the available memory resources.

n	*elements*	*additions*	*multiplications*
1024	1,048,576	1,072,693,248	1,073,741,824
2048	4,194,304	8,585,740,288	8,589,934,592
4096	16,777,216	68,702,699,520	68,719,476,736
8192	67,108,864	549,688,705,024	549,755,813,888

Table 2.1: *Workload versus application size for matrix multiplication*

According to [16] various factors exist that will limit the overall speedup S(N). Granularity of the parallel job, for a fixed problem size, is the prime determinant (i.e. the $(1 - f_s/\mathrm{N})$ term of equation 2.4). Granularity is a qualitative measure of the ratio of computation to communication [8]. Jobs are often defined as being *coarse*, *medium*, or *fine* grained with the distinction between them being the number of operations that can be executed in parallel before a data dependence occurs (i.e. a serial section where communication is required). Fine granularity allows for relatively small amounts of computation to be done between serial sections, while coarse granularity allows for a large amount of work to be done.
Overhead can also be a significant factor. This overhead can consist of many components with the most notable being parallelism start-up and termination times, synchronisation, data communication, and overhead imposed by the parallelising methodology (discussed below) i.e. operating system, parallel library, compiler, or other tools.

2.2 Computing Platforms

The benefits of parallel computing were alluded to in the previous section, and also the associated dependence on the properties of the underlying platforms. Such properties have a major influence in determining the overhead component, $t_o(N)$ of equation 2.4. These properties include such factors as data communication overheads, start-up and termination time, synchronisation time, and the overhead introduced by the software system which can include contributions from the operating system and the application itself. As not all parallel architectures are equal in these respects it is important to highlight the inefficiencies of the primary architectures so that these drawbacks can be specifically accounted for.
The most popular general classification for computing platforms is Flynn's Taxonomy [17], which classifies computer architectures according to the number of instruction and data streams they have, or more precisely whether they have single or multiple streams. Accordingly there are four distinct classifications:

- **SISD** (*Single Instruction Single Data*): classifies a sequential computer system that can only execute one instruction per unit of processor time, which can only use one data stream for input. Early commodity uni-processors implementations are generally categorised under this class (see below).

- **SIMD** (*Single Instruction Multiple Data*): more than one functional processor exists but all units execute the same operation during at any given clock cycle. However, each unit can operate on different data. This architecture typically consists of an instruction dispatcher and a number of instruction units. This is best suited to applications that exhibit a high degree of regularity such as multimedia processing. This architecture is found in modern processors that implement arithmetic-level parallelism, processor arrays, and vector pipelines.

- **MISD** (*Multiple Instruction Single Data*): this type of architecture, allows multiple instructions to be performed on the same input data, is unusual and of very specialised use. Conceivable machines that could be represented by this topology might serve multiple potential (speculative) operations, e.g. in cryptography.

- **MIMD** (*Multiple Instruction Multiple Data*): here multiple execution paths (threads of execution) potentially operate on different data streams. Execution can be synchronous or asynchronous. The vast majority of parallel computing platforms fit into this class.

Although there are machines that exist which cannot be easily classified under this taxonomy, it is still one of the best general classifications for computing architectures. Johnson's Extension of this taxonomy [18], is depicted in Figure 2.3. It further classifies the `MIMD` class depending on whether memory is Global (GM) or Distributed (DM), and how communication is achieved, through Shared Variables (SV) or Message passing (MP).

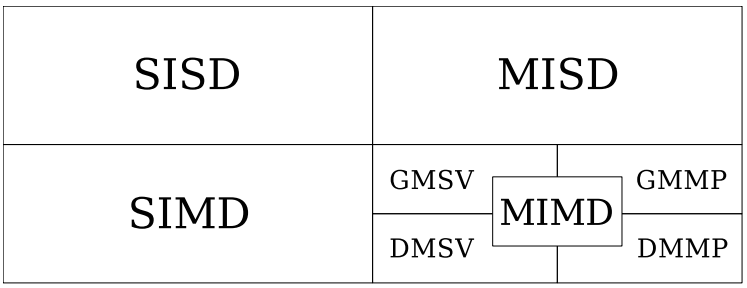


Figure 2.3: *Johnson's Extension to Flynn's Taxonomy*

A computer with one central processor unit (CPU) is often termed a uni-processor. It has its own private memory space. As only one thread of execution may modify data at any one time there are no immediately obvious problems associated with data inconsistency, although in fact the processor-memory pair must obey at least a minimum level of consistency that preserves the ordering of actions. The inclusion of caching to hide memory access latency forces the use of cache consistency protocols to maintain consistency across the processor-cache-memory triplet. This architecture is depicted in Figure 2.4(a). Only pseudo-concurrency is possible, where independent threads of execution are interleaved and allocated to the processor in a round-robin fashion. Thus, in isolation a uni-processor is not suitable for parallel computing, however it will be seen below (Section 2.2.2) that it can be used as a component in a parallel computing system. The definition of a uni-processor will possibly need to be redefined with the advent of multi-core CPUs, where a number of CPU cores are located on the same package. In truth these are typically MIMD architectures. These developments will further promote the development of applications with multiple threads of execution (multi-threaded).
Parallel systems come in three flavours: those where memory is shared among all processors, those where it is not, and those where limited sharing is possible.

2.2.1 Shared-memory multi-processing

Computing systems that consist of a number of processors (N > 1) that share the same memory resources are termed Multi-Processors. These MIMD architecture have the advantage that two or more threads of execution may run concurrently on individual processors and access the same physical memory space. Hardware cache coherency protocols exist to ensure that when two (or more) processors are competing for access to the same memory location then the data remains consistent.
The symmetric multiprocessor (SMP) system depicted in Figure 2.4(b) is one of a number of possible implementations of a shared memory parallel platform. All accesses to memory exhibit the same latencies for any processor. These SMPs belong to the uniform memory access (UMA) architecture family. The primary disadvantage of these shared-memory architectures is the lack of scalability that comes from high memory-bus

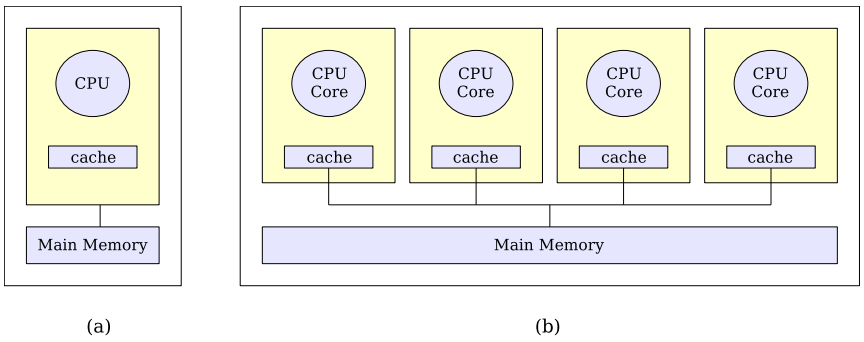


Figure 2.4: *Architectures: (a) Uni-Processor and (b) SMP UMA*

contention due to an insistence on uniformity. Their counter advantage is the simplicity of a uniform programming model, and the very good price/performance ratio for a small number (2 to 4) of CPUs.

The rest of the multiprocessor space comprises machines where processors can directly access the memory of a remote system (without the intervention of a remote operating system), but with non-uniform latencies. Unsurprisingly, these are called non-uniform memory access (NUMA) architectures. The scalability of the NUMA architecture is of the order of thousands, and is the choice for the most powerful supercomputers in existence, such as the Earth Simulator [19]. This is an expensive option as specialised interconnect, switch, and memory interface hardware is required. Non-commodity interconnects such as SCI [20], Myrinet [21], and Infiniband [22], yielding low latency data transfers, enable the construction of NUMA machines using commodity hardware. Interconnects such as SCI can maintain cache consistency (i.e. ccNUMA), but this feature is rarely used.

Most NUMA implementations rely on hardware to signal the various processors that some action is required, but there are some (see table B.2) where the signalling is performed by messaging in software, creating a hybrid where hardware and software cooperate to provide NUMA. Alewife was one such implementation [23]. In the extreme, shared memory can be handled entirely in software, in which case it is termed Distributed Shared Memory (DSM), see Chapter 4.

Most DSM implementations assume the processors are logically separated by distances that introduce significant latencies (e.g. on an Ethernet network with latencies > 100ms) but this is not a principal property of DSM (for example, SCI can support DSM with latencies $< 5\mu$s over distances up to about 10 metres).

Terminology has evolved to categorise three variants. Entirely hardware-based NUMA is called H-DSM. Entirely software-based DSM is called S-DSM, but most interpret DSM to mean this variant too. Hybrid hardware/software approaches are termed Hybrid-DSM. In general DSM is notable for latency-hiding mechanisms that have been created to mitigate the loss of data consistency and/or the performance degradation that would otherwise occur [7] (Chap. 5, pg.250). All the same, performance is not good except for

the coarsest granularity (embarrassingly parallel) applications.

2.2.2 No shared memory multi-processing

The flavour of a parallel system that does not share memory across the processes generally is a result of collecting together a number of independent processors. For example, multiple uni-processors can be amalgamated into quasi parallel systems commonly referred to as loosely-coupled clusters. In these systems communications between processors is facilitated by the network connection. No facility for remote memory access (RMA) is available so a process cannot access the private address space of a process executing on a remote machine. This type of architecture is broadly referred to as a distributed memory machine, or ensemble, or *No Remote Memory Access* (NORMA) architecture. This type of system is inherently more scalable in terms of cost, but at the cost of increased programming burden, as data sharing must be explicitly facilitated by the developer by explicitly transferring it from one process' memory to another.
The extreme case for distributed memory architectures is when geographically dispersed processors are connected by a wide-area network. For a small number of processors (typically high-end supercomputers) this is known as meta-computing [24]. When an infrastructure is created that includes many geographically dispersed sites, this has become known as a Grid (although the exact definition of a Grid is still in dispute). Again, performance is not good except for the embarrassingly parallel class of applications.

2.2.3 Hybrid multi-processing

Some systems support limited sharing of memory generally exploit locality, such as sharing with the nearest neighbour in the network topology. This is an effective stratagem, but yields a restrictive programming model.
The future of parallel computing architecture is tending towards hybrid architectures with systems composed of thousands of processors, termed Massively Parallel Processors (MPP) or systems composed of clusters of SMPs (constellations) as depicted in Figure 2.5. Memory can be shared between processors within a node, while no memory sharing exists between nodes. Again, these clusters can in turn be coupled with those in other geographically distributed sites to form computational grids.

2.3 Parallel Programming Models

The method of developing parallel applications varies depending on the architecture to be employed. In the following sections the three main programming models are discussed, plus some of the implementations. Although many models (in theory) can be implemented for the underlying hardware, some lend themselves better to some platforms than others. There is no one decisive model for all possible architectures.
The choice of programming model can be influenced by the method in which the application is decomposed into tasks that can be performed in parallel. The two general approaches are:

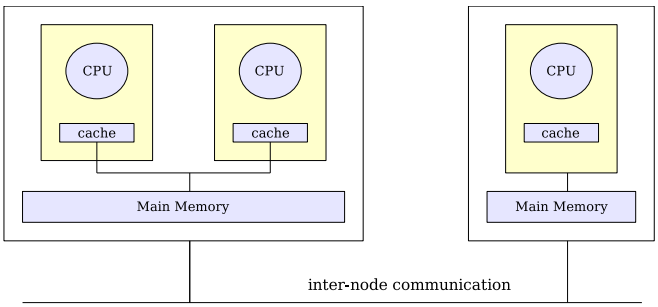


Figure 2.5: *Shared & Distributed Memory Hybrids*

- *Domain partitioning*: the input data is partitioned among the processes which perform the work on their allotted portion, and the partial results are merged together to produce the result. This type of job is normally found in single program multiple data (SPMD) tasks where all the parallel processes execute the same program. This enables space-data parallelism (Same principal to SIMD parallelism discussed in Section 2.2).
- *Functional partitioning*: the application is partitioned according to different tasks that need to be performed on input data. The output of one task forms the input for another (a pipe-line approach). Parallelism arises as this task is done repeatedly. The multiple program multiple data (MPMD) paradigm is often associated with distributed memory programming models. This scheme enables task-time parallelism (Analogous to the MIMD machine taxonomy in Section 2.2).

Some programming models are often preferred over others due to the associated semantics, as one model's semantics can be more demanding of the programmer than another. This additional burden can result in increased code volumes/development time, increased difficulty in algorithm implementation due to increased complexity in testing and debugging of the application.

2.3.1 Loop parallelism model

This type of model has traditionally been found on multiprocessor systems, but in principle is more generalised. The most popular implementations are High Performance FORTRAN (HPF) [25], Co-Array FORTRAN (CAF) [26], Unified Parallel C (UPC) [27], and OpenMP [4]. This approach requires minimal effort from the developer with respect to other models described below, that require the explicit distribution of data. This code is then compiled by a parallelising compiler that transforms the code into a multi-threaded application that is most suitable for execution on a shared memory processor. There have been some efforts to target distributed memory machines by utilising messaging for intra-process communication, but these have resulted in poor performance due to excessive communication overheads [28].

This programming model requires the developer to instrument the code with parallel directives, which can be done in an incremental manner, indicating to the compiler where parallel functionality is required. An example of a parallel directive is the *#pragma omp for* directive in the following OpenMP code snippet. This identifies the areas of code (that can be performed in parallel) to the compiler. The following code snippet implements matrix multiplication. The directive indicates that the subsequent structured *for* loop can be performed by a number of threads concurrently. This may be done because there are no data dependencies within the structured loop.

```
1       #pragma omp for schedule (static, chunk)
        for (i = 0; i < NRA; k++)
3         for(k = 0; k < NCB; i++){
            c[i][k] = 0;
5           for (j=0; j < NCA; j++)
                c[i][k] = c[i][k] + a[i][j] * b[j][k];
7         } /*** End of parallel region ***/
```

This model places little burden on the programmer as the compiler takes responsibility for the placement of data, partitioning of work, and thread scheduling (however the user can override default parameters in OpenMP by specifying scheduling clauses such as *static* & *chunk*). The main burden is the identification of the regions to be parallelised, and where necessary, to direct the scheduling of the threads. In certain circumstances the compiler may not produce the most optimised solution when compared to a hand coded approach where the developer takes responsibility for allocation of work.
The model performs best when the underlying hardware shares memory across the processors. It is most applicable to the class of domain partitioning. Irregular computation is a poor candidate. With the advent of multi-core processors that may share cache in addition to memory and that are primarily designed for multi-threaded applications, then such a programming model will undoubtedly become more popular in the future.

2.3.2 Shared Memory Model

This model explicitly assumes that system memory is shared among processors, thus allowing for application data to be shared transparently among processes executing concurrently. To provide this facility one of the following must be supported by the operating system of the multi-processor machine: (a) sharing of memory regions between processes using *shmem* routines, or (b) multiple threads of execution per process (multi-threading). Recently the latter has begun to dominate as the preferred way to program multiprocessor systems due to the availability of portable libraries such as *pthreads*, which remove the burden on the programmer to explicitly declare & map shared memory regions since this is now achieved automatically (all threads share the same address space anyway).
The following code implements the same matrix-multiply functionality of the previous section using a shared-memory multi-threading model. The increased burden placed on

the programmer can be observed, which involves the initial creation of threads, work provisioning, and the requirement to ensure cleanup of threads.

```
1       // Initialise a & b matrices
        ...
3       // Create threads, these perform a matrix multiply
        for(n = 1; n <= NUM_THREADS; n++){
5         tid[n] = thread_create(&matrix_mult_fn);
        }
7       ...
        // Body of matrix multiply function
9       my_start = NRA/NUM_THREADS * (my_tid);
        my_end   = NRA/NUM_THREADS * (my_tid + 1);
11      for (i = my_start; i < my_end; i++)
          for (k = 0; k < NCB; k++){
13          c[i][k] = 0;
            for (j=0; j < NCA; j++)
15            c[i][k] = c[i][k] + a[i][j] * b[j][k];
          }
17      ...
        // join threads
19      for(n = 0; n <= NUM_THREADS; n++)
          thread_join();
```

The main difference between the previous model and this one is the explicit assumption of shared memory, consequently the developer is additionally responsible for creating and administering the parallelism (the compiler performs this work in the loop parallelism model). This model is suitable for domain or functional decomposition of applications, whereas loop-parallelism is, in general, most applicable to domain partitioning. The primary downsides (or maybe benefits) are the need for explicit management of threads, synchronisation of threads when accessing shared data, and the allocation of work to each individual thread.

To date the previous model and this one have only been truly used where system memory can be shared among the processors. Sharing does involve a common view of memory, and when more processors are involved the physical distances between them increases, and hence inter-process latency increases, eventually becoming highly visible. Once this happens, formerly simple actions such as synchronisation become excessively complex. Hiding the latency requires specialised hardware, and its costs rise sharply as the number of processors increases. However, at small scales the model is very effective. The tools required to program in this model can be found on most platforms and architectures, and as such is the most familiar and the most natural model for developers of parallel applications. In addition, it offers more flexibility to decompose the task in whatever manner is best applicable. As discussed previously, the availability of multi-core processors will greatly increase interest in this area of multi-threaded applications.

2.3.3 Message Passing Model

The message passing model of parallel computing has a natural affinity with distributed memory machines. Messages are sent between processes via a communication channel created using a software library. In the traditional mechanism, processes coordinate themselves into send/receive pairs, so a process will send a message to a named process, more than likely on a remote node. Current message passing implementations include additional functionality such as collective communication routines such as a broadcast operation. Although such extra functionality is helpful, a high level of programmer burden persists with this model.
Not only does the programmer have the same burden to contend with as with the previously mentioned models, but additional responsibility rests on the programmer for managing all communication. This can result in considerable extra effort by the programmer, often manifested in the greater complexity in the parallelisation of an algorithm, and hence this becomes a source of errors [7].

```
        MPI_Init(&argc, &argv);
2       MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
        MPI_Comm_size(MPI_COMM_WORLD, &sys_size);
4       ...
        // Init a & b matrices, variables my_start & my_end
6       my_start = NRA/sys_size * my_rank);
        my_end   = NRA/sys_size * my_rank+1);
8       ...
        MPI_Bcast(&a, (DIM * DIM), MPI_INT, 0, MPI_COMM);
10      MPI_Bcast(&b, (DIM * DIM), MPI_INT, 0, MPI_COMM);
        ...
12      for (i = my_start; i < my_end; i++)
          for (k = 0; k < NCB; k++){
14          c[i][k] = 0;
            for (j=0; j < NCA; j++)
16            c[i][k] = c[i][k] + a[i][j] * b[j][k];
          }
18      ...
        // amalgamate results from all processes at rank 0
20      if(my_rank == 0){
          MPI_recv(&c[my_start][0],(NRA * rows), MPI_INT, 0,
22                 Rank, MPI_COMM);
        }else{
24        MPI_Send(&c[my_start][0],(DIM * rows), MPI_INT, 0,
                   0, MPI_COMM);
26      }
        MPI_Finalize();
```

Again, the programming model is demonstrated with the same matrix multiplication algorithm implemented in previous sections. Not only is there an increase in the lines of code (LOC), but also the derivation of communication logic can be complicated, even for a near-trivial task such as that shown above. Transferring the incident matrices' data (Lines 09-10), and the resultant data (Lines 22-27) requires the sender & receiver to be explicitly defined. Nevertheless it is exactly this requirement that makes for more efficient programs than other models. It is potentially more efficient simply because there is more requirement for the developer's intelligence to be brought to bear on the problem.
One benefit that can be gained from message passing is that where blocking calls are used, synchronisation can be implied. This is a small factor, but is very helpful nonetheless in providing implicit synchronisation between the competing threads. The message passing model is amenable to both domain and functional partitioning.

2.4 Parallel Synchronisation

In parallel programming, situations occur where there is a data dependence between the order of program statement execution. Ensuring such situations (often termed data races) are absent is vital. In order to achieve this, synchronisation primitives are required. In addition to these explicit primitives, implicit means are also available. In the message-passing model discussed above it was seen that there is also the possibility to implement synchronisation using synchronous messaging routines. However, only those tasks that are participating in the communication operation are synchronised.
With any shared-memory-style of programming paradigm (or variant) there is a need for exclusive access to shared memory, i.e. mutually exclusive access, and for the synchronisation of threads of execution in the application. The most common occurring synchronisation variables are locks and barriers.

2.4.1 Lock Synchronisation

A lock, or a MUTual EXclusion (mutex) device, is used for protecting shared data structures from conflicting modifications by multiple processes by protecting sections of code that actually modify them. These code areas are termed critical sections. Locks are also used for implementation of higher-level abstractions such as monitors. There are four requirements of a system that provide the use of critical sections through the use of locks. The first three requirements are the responsibility of the system, while a developer/algorithm will be responsible for ensuring the last:

i. *Mutual exclusion*: at most one thread of execution may execute the critical section at any one time

ii. *Eventual Entry*: in there are multiple threads trying to access a critical section at the same time, then one of them must succeed in acquiring the lock

iii. *Absence of Delay*: a thread should get access to the critical section if no other thread is already doing so

iv. *Absence of Starvation*: lack of deadlock/livelock; a thread that is attempting to access a critical section will eventually be allowed

In traditional multi-processor settings these primitives consist of a shared memory location that can be set to a certain value indicating its state (lock/unlocked). This structure is often termed a spin-lock and is equivalent to the code fragment below. Mutual exclusion is supported by all modern CPUs through the provision of an atomic *test_and_set* (or equivalent) instruction that can test the lock, and depending on this value set it or do nothing, without been preempted by another processor. Such a situation could occur if the (crude) test-and-spin code section below was being concurrently executed by two separate threads of execution. It must be noted that more efficient solutions exist for the implementation of critical section on shared memory machines, such as Peterson's Algorithm, that results in reduced memory contention [29].

```
while(lock != SET);
lock = SET;
```

The implementation of distributed locks has associated with it a number of problems. Most importantly there is no distributed atomic *read-and-modify* instruction, and memory with the required consistency for the above code fragment is not available, and in particular, defects with the required properties (ii & iii) above are highly exacerbated. Additional challenges associated with implementing distributed locks are that every lock primitive must be able to be uniquely identified system-wide, and that asynchronous techniques used in shared memory locks are not suitable for distributed memory machines due to scarce inter-machine communication resources.
In general, a lock can be in three main states: exclusive, non-exclusive and free. The modes that a lock can be held in can be classified in two categories, exclusive or non-exclusive mode (also termed read & write modes). The number of threads that can possess a lock in these modes simultaneously will be governed by the shared memory access modes supported by the system (discussed in Section 4.2).
A lock held in non-exclusive mode can be acquired by multiple nodes simultaneously in this mode (i.e. rule (i) above does not apply). Once in this state it cannot be promoted to exclusive mode without first releasing it and reacquiring it. This type of mode can also be referred to as read access. In order for a thread of execution to acquire a lock variable in non-exclusive mode it must obtain permission from either the owner, or another thread that has already been granted non-exclusive access. This other thread may reside within the same process as the requesting thread, or in a process on a remote node.
Exclusive lock access differs slightly from non-exclusive locks as only one thread, the owner, can possess the lock and therefore perform write accesses inside the critical section. In order for a lock to be granted in exclusive mode, no other process can be in possession of the lock in exclusive or non-exclusive mode.

2.4.2 Barrier Synchronisation

A barrier is a mechanism that provides for the synchronisation of a number of processes in a parallel application. It requires all threads participating in the operation to call a barrier routine and wait until all other processes have also done so. Once this has occurred all threads may proceed. The simplest method to implement a barrier in a shared memory system is to use a memory location as a shared counter. This counter is incremented (atomically) when a thread arrives at the barrier. The thread subsequently waits until the required count (quorum) is reached. Such an algorithm results in high memory contention [30]. A combining tree barrier algorithm reduces contention by introducing sub-counters that record arrivals of a subset of the threads (Figure 2.6(a)). When the sub-count is reached the parent count is incremented. When this count reaches its quorum the thread may proceed. Other (symmetric) algorithms exist that attempt to reduce the load so that all threads wait for the same amount of time. ‘
Barrier primitive implementations have always been inefficient [31], their use in distributed-memory applications should be minimised. However, often their presence is mandatory in applications, particularly in iterative applications. Whenever this occurs the situation can be exploited, as shared state information can be distributed globally piggybacked in the barrier messages. For distributed systems some notable barrier implementations have included [32, 30].

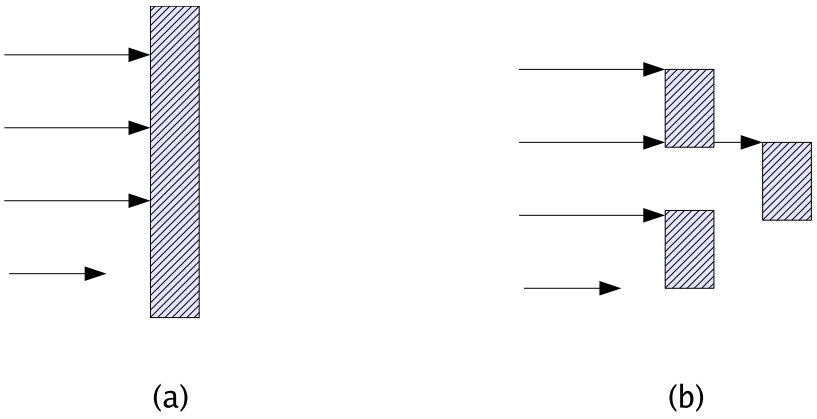


Figure 2.6: *Barrier primitives. (a)Central server waiting for all threads to arrive. (b) tree-based barrier all thread proceeding*

The simplest distributed barrier implementation, the central server, is similar in methodology to the shared counter. It consists of a central barrier administrator that maintains a count and accepts arrival notices from the processes partaking in the barrier, incrementing the count with each new arrival. Once all arrival notices have been received, including the local notice, proceed notices are issued. Such an implementation suffers from a lack of scalability as the barrier administrator becomes the bottleneck, with N-1 nodes contacting the master at arrival, and it having to reciprocate N-1 times.

A modification of the previous scheme involves the adoption a tree algorithm where barrier sub-administrators relieve some of the burden from the administrator. A node partaking in a barrier operation may have antecedents and a consequent. The node can only issue a barrier arrival notification to its consequent once its local quorum has been satisfied. This quorum is composed of the local requirements (have all local threads arrived?), and that of the antecedents. Once the requirements have been met the process can issue an arrival notice to its consequent. At the top of the tree there exists the barrier administrator. Once the quorum of the administrator has been reached it can issue proceed notices. Based on the LogP model [33], the minimum wait latency for a barrier using such an implementation is $2log_2N \times Average\ message\ latency$.

2.5 Other Considerations

Constructing parallel applications can involve many, and often conflicting, considerations. Some of these have already been mentioned, such as whether the functional (task) or domain (data) approach is taken to problem decomposition, which is in turn dependent on the underlying algorithm. Other factors that need consideration are:

- *Granularity:* which is a qualitative measure of the ratio of computation to communication and synchronisation, in essence the frequency at which data dependencies arise. Applications that exhibit a coarse granularity are more suitable for parallelisation on distributed memory architectures. Consider a simple application of summing an array of N integers. It is possible for the application to be parallelised where each process in a process pool, of size M, performs the required calculation of a sub-array of size N/M. If M = N/2, then each process will initially perform one addition, in which case the communication and/or synchronisation overhead would result in a significant slowdown, as the time for a modern processors to execute one addition is negligible with respect to communication cost.

- *Data Access Patterns:* there have been a number of attempts to classify parallel applications and their sharing patterns. The developers of the Munin DSM [34] identified distinct categories of shared data. Data sharing that is responsible for inter-process communication contributes to lowering the granularity and thus is responsible for a degradation in performance and/or scalability of parallelisation. Poor algorithm design can also be a contributing factor, since accessing data in a non-structured manner will generate more traffic than structured access where optimisations can be taken advantage of.

- *Load balancing:* if a distributed memory cluster is very loosely composed, e.g. from a network of workstations (NOW) [7], then it is likely to be composed of different architectures and platforms, and so different threads of a parallel application will have different performance characteristics. Even machines with the same architecture and platform can have different performance metrics. This is an important fact to consider as in many applications the performance will be governed by that

of the slowest processor. Load balancing attempts to match tasks to processors such that overall performance is maximised. For example, if in the matrix example of the previous sections, this same job is divided among M processors, where the performance of one is greater than the other, then as there is a data dependency between the M sub tasks the performance of the system is determined by the last thread to finish its task.
On symmetric-memory architectures this is less important so long as the developer partitions the work evenly among the threads, as each thread will normally get the same resources at the same performance level. Any asymmetry exacerbates the problem and so requires load balancing.

- *Data Distribution:* it is often necessary for the programmer to consider data distribution when developing the algorithm for the application. Employing a data provisioning strategy will allow for the caching principle of data locality to persist across distributed machines; additionally, a reduction in the potential load imbalance will occur. Data parallel languages, such as HPF [35], allow for the programmer to annotate variables, so that a strategy can be followed (or at least suggested to the compiler) for data distribution that employs one of the traditional categories, such as: block, cyclic, block-cyclic, replicated, and local [8].

- *Fault Tolerance:* fault tolerance is the ability of a system to recover from a situation that would otherwise result in failure, possibly necessitating the complete restart of an application. In parallel computing one errant resource could be responsible for the aborting of a job involving thousands of processors. When such an event occurs the subsequent action is determined by the availability of a fault tolerant recovery mechanism e.g. re-start from a saved/checkpoint-ed state. Efforts have been made to address this area such as fault tolerant message passing implementations exist such as [36] described in Section D.
Applying fault tolerance to parallel programming is another important factor that is often not considered by the application developer as extra effort is introduced into the development process, so transparent fault-tolerance is a desirable attribute. Fault tolerance can be supported in DSM systems, however some direction from the programmer will still be necessary, as the application would need to be checkpointed at global synchronisation points.

- *Data Divergence:* A parallel implementation of an algorithm may obtain a different result than a comparable serial implementation [37]. Such a situation can arise as floating-point arithmetic is not associative, or in a situation that may arise from the rounding accuracy associated with differences in internal representation of floating point variables, leading to a variance in precision when performing floating point arithmetic (e.g. Intel processors represent IEEE 754 double precision values internally as 80-bit values, while PowerPC has a direct 64-bit representation).

- *Multi-thread per process support:* Section 2.2 identified the likely future direction of computer architecture. Multi-core processors are now standard and many observers

predict that the number of cores will rise sharply in the near future. Support for multi-threaded systems will be vital in order to leverage this. Even with copy-on-write and shared code features that are now standard with modern operating systems it is not feasible to allow many processes with multiple threads of execution to share the increasingly scarce resource that is memory bandwidth. To maximise resources one should ensure that there the number of applications threads be at least equal, if not greater than the number of processing cores, i.e. 1 Process : N user threads : M cores per processing node ($N \geq M$).

2.6 Review

Shared memory and loop-parallel programming are the favoured methods of implementing parallel applications due to the lower burden placed on the programmer. Explicit data movement does not occur with shared memory models, whereas with message passing it must.

Message-passing is used extensively on distributed memory machines, allowing the execution of applications written in this model to execute on almost all current platforms, including those with shared memory. Often a recompile is all that's required for this to be achieved. So why not just use message passing exclusively? It always comes down to the extra burden placed on the programmer.

The future trends of parallel computer architecture will dictate the need for hybrid programming models that fully exploit the potential offered by constellations of SMPs (particularly those with multi-core CPUs). Already research exists in exploring the use of mixed-mode programming models with combinations of message passing and loop-parallel/shared memory [38, 39, 40]. Although considered by many to employ a loop-parallelism model, OpenMP can be viewed very much as a hybrid. Development involving both OpenMP and message passing applications can be found in the most powerful supercomputers (such as the Earth Simulator [19]), which are in effect a constellation of SMPs. Message passing is used to communicate between nodes, while shared memory can be used internally in the SMP.

There are new problems that arise when dealing with a hybrid programming model: the developer needs to be knowledgeable in two programming models. The readability of code is significantly reduced, especially for third-parties (those who did not write the program); this will be an issue with maintenance of such code, and importantly how easy will it be to debug?

CHAPTER 3

Wide-Area Parallel Computing

In the previous chapter the motivations for parallel computing were explored. To recap, it essentially allows for an application that was compute bound to complete in a timely manner when executed on a system with increased processing capacity (Amdahl' Law), and/or allows for increasing the accuracy of the application results by increasing the resolution at which the solution is computed (essentially Gustafson's Observation). In this chapter we explore what will take parallel computing to a wide-area environment.
Previous parallel computing research focused on the benefits of federating resources from the point of view of the computational instruction execution rate. The increase of aggregate memory resources was a lesser issue. There are anticipated applications [24], that will benefit enormously from the increased parallelisation offered by such an increased memory: image rendering, parallel search, speech and visual recognition, genetic sequencing, data synthesis, and data-mining. These applications will not only have increased computational requirements, but will also have a commensurate memory resource requirement. The challenge will be to harness both increases in the memory and computational resources.
To support this new class of parallel application we must be able to effectively pool these resources into a unified resource. Consider an application where the required memory resource is greater than the available resources, as the application accesses virtual memory pages the corresponding physical pages may need to be migrated to the physical resource. Pages that are deemed less necessary will begin to be swapped out to a swap space that is usually located on a dedicated partition on the local secondary storage system. This storage is slow enough [41], that it is feasible to buffer the page in the memory of an adjacent (in network terms) machine. Increasing the total number of such machines will increase the total memory pool available for application use. If all the available machines have similar demands, then this approach is pointless, although machines might be dedicated to this purpose.
Increased parallelism can most easily be achieved by hiding latency, and the the most promising way to achieve this will be a weakening in memory consistency [42]. However, in the context of the claims made in the above paragraph, a latent effect of doing this will

be a commensurate increase in the memory resource usage. To access the benefits of the parallelism with grid computing, two principal metrics must be favourable: scalability and efficiency.

3.1 Wide Area Parallel Computing Platforms

The natural extension of the cluster paradigm described in Section 2.2.2 is to combine resources at multiple sites into a coupled system as depicted in Figure 3.1, referred to as a meta-computer, or when many sites are involved in a formal manner, a grid. Making such disparate entities appear a unified resource to a user requires a middleware support 'glue' layer to solve problems involving areas such as administration, security, etc. The grid middleware aims to abstract the resources to provide standard methods and interfaces to access the resources at each site. Multiple grid middlewares exist, each developed by different projects: Legion [43], Unicore [44], and more recently EGEE (Enabling Grids for E-sciencE) [45].

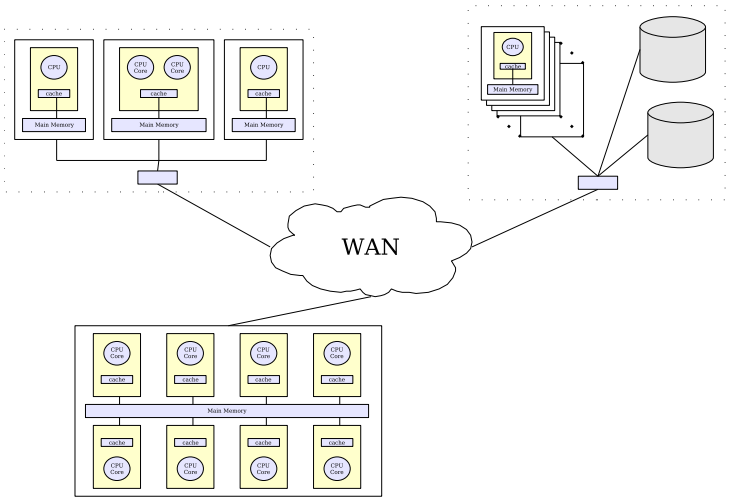


Figure 3.1: *Grid: wide-area distributed computing*

Attempting to use such collections of resources will result in the effects of physical laws governing communication such as bandwidth and latency (see Section 5.1 for more information) becoming more apparent. A number of grid projects are examining wide area computing theory by providing high-bandwidth, low-latency and reasonably deterministic connections between sites, e.g. Naregi [46] in Japan and DAS-3 in Holland, both linking high performance centres to create a giant supercomputer [47]. Naregi is designed for a maximum latency of 25ms (800 km). DAS-3 reduces this further to 3ms (the latency across Holland, and coincidentally the latency for an access to a hard-disk [41]).

As in smaller-scale systems, the available parallel computing architectures for such wide-area environments are still:

- **wide-area shared memory:** such an architecture is barely feasible unless the memory consistency is sufficiently weak to hide high latencies. It is conceivable, but highly impractical to construct a wide area interconnect with hardware support for shared memory. Shared memory could only be realistically supported by virtually sharing memory resources while hiding latencies, effectively via a DSM on a grid. Latencies as low as those for DAS-3 do improve the situation.
- **wide-area distributed memory:** essentially this a Grid. MPI software implementations exist, e.g. PAC-X [48], MPICH-G2 [49], and GridMPI [50], that allow multiple compute elements to appear as a single unified resource, i.e. 'a cluster of clusters', by adding a transport mechanism between the native MPI implementations used on each cluster. For PAC-X there is a gateway process at each site that proxies all messages for a non-local process to the relevant gateway process in the remote cluster. Such software is discussed further in Appendix D.

The function of the middleware is to provide the functionality to authenticate the user and to perform job and data management functions. Using a Grid requires interaction with this middleware that manages the differences of the underlying architecture and platforms. For example, if a user possesses appropriate grid credentials they can sign onto the EGEE *gLite* grid at a gateway machine and then utilise any EGEE resources (compute/storage) that are available to them, and perform other operations such as using the information systems (see Appendix D, page 226). Compute jobs are submitted using a command *gLite-job-submit*. Each submitted job will be assigned a unique job identifier that can be used to check status of the job. Once the job has completed the output may be retrieved using the command *gLite-job-get-output*. Similar abstractions are going to be needed to run parallel jobs. Other commands exist for data management operations.

3.2 Programming Models for Wide Area Parallel Computing

The parallel programming models discussed in Section 2.3 assume an execution target that is a homogeneous distributed memory machine such as a cluster, not the grid discussed above. These programming models may not be adequate for the grid.

- **wide-area shared memory:** Providing a shared memory model in a wide area environment suffers from the impracticability of hardware support. The only realistic solution is to provide virtual sharing of memory.
- **wide-area loop parallelism:** As loop parallelism is ultimately a higher form of shared memory programming, providing this programming model on the grid poses all the same problems associated with shared memory programming. Much of the parallelising work is performed by the compiler, so these problems can be solved

transparently to the user, but the compiler will have to be aware of the impact on performance, and accordingly must schedule/partition the problem accordingly.

- **wide-area message-passing:** In the previous section it was seen that multi-site MPI implementations exist. These allow for grid resources from multiple sites to appear as a unified execution environment. No alteration to user code is required in order to run in such an environment. Optimisations to the underlying messaging passing software have been implemented that allow for certain operations, such as collective operations, to run more efficiently [51, 52]. These packages use hierarchy information to implement the operations by creating a topology map. As mentioned above, the implementations already exist for this. The problem is the added programming burden relative to shared memory models.

According to the seminal book on Grid computing [24], if the shared memory paradigm could be made available to developers, then grid programming would be reduced to optimising the assignment and use of threads, and the communication system. It is the optimisation of the the latter that is of interest in this thesis.
It should be noted at this point that Grids come in forms other than those geared towards processing the highly-parallel computationally-intensive tasks that have been discussed so far in this thesis. Another important class are data parallel applications involving tasks that require significant amounts of CPU time. However this is not the only alternative use case; in addition one should consider applications requiring processing large amounts of input data, as I/O latencies are large, and often the only feasible option is to store the data, and consequently the processing of it, in a distributed manner. Some of the earliest examples of this approach have been the Seti@Home [3] and Folding@Home [6] projects, but notable experiments that require a formal grid infrastructure include the four main experiments involving the Large Hadron Collider (LHC) at CERN [53]: ALICE, ATLAS, CMS, and LHCb.

3.3 Evaluating Parallelism on the Grid

The vast majority of applications when submitted to the grid will be scheduled to where the necessary resources are available, hopefully within a single site. There may be applications that are scheduled across multiple grid sites due to circumstances such as insufficient compute resources being available at any one site, or the necessary job data being distributed among a number of sites. In such a scenario it may be impractical to transfer very large data volumes, or not permissible for ownership or legal reasons to bring the data from remote sites to the compute resource. All these situations require parallel solutions. The question is how well they perform? The scalability and efficiency metrics outlined in Section 2.1 assume homogeneous resources.
A naive approach is to extrapolate the potential scalability of a parallel application in a grid from Equation 2.4. According to this equation, it was assumed that the overhead times, t_{opi} & t_{osi}, were all uniform, but in a grid platform this is not a valid assumption. In addition the various compute resources will have differing performance, so the time

to compute each parallel component, t_{pi}, needs to be calculated for each resource in accordance with its capability, i.e. t_{pi} should not necessarily be the same for all i. The total parallel section can now expressed by the term $\sum_{i=0}^{n} t_{pi}$. Expanding Equation 2.4 to take cognisance of these factors we arrive at Equation 3.1:

$$S(n) = \frac{t_s + \sum_{i=0}^{n} t_{pi}}{t_s + \sum_{i=0}^{n} t_{osi} + max(t_{pi}) + \sum_{i=0}^{n} t_{opi}} \tag{3.1}$$

By inspection, one can see that it is possible to predict whether a certain application is suitable for execution in this environment. In order for the application execution to be feasible the overhead components t_{osi} (serial section overhead) and t_{opi} (the parallel overhead) will need to be reduced so that they are insignificant. The effort required to make the necessary modifications will have to be justified by an increase in performance. [54] suggests a method by which this can be quantified. Previous work has shown that application-specific knowledge can aid in this area [55, 56, 57].
A different approach was taken in [58], which defined a scalability metric for the efficiency of parallel computing in a multi-site grid. Unsurprisingly it demonstrates that for a small number of sites adequate speedup is only attainable if the amount of the total workload of the application, W, attributed to each site is significant enough. The time for this application to execute in a homogeneous computational grid (HCG) is given by Equation 3.2. This equation assumes that the parallel application is hierarchically decomposed. It is assumed that the HCG is composed of C Compute Elements (CE), where each has the same number of processors (p), which in turn have similar compute power (Δ), as depicted in Figure 3.2.

$$T_{C,p}(W) = \frac{W_p}{pC\Delta} + Q_1(W_p, C) + Q_2(W_p/C, p) \tag{3.2}$$

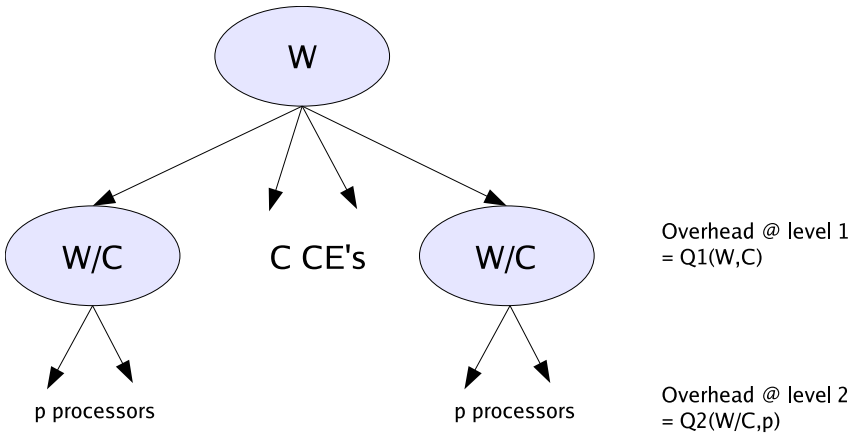


Figure 3.2: *Overhead in hierarchical decomposition*

The first term represents the workload when it is divided evenly among the CEs, each having processing power $p\Delta$. The components Q_1 and Q_2 represent the overheads in decomposing the application. These terms are highly dependent upon the application; the former term is the overhead in decomposing the application among the CEs, while the latter is that incurred while decomposing among the processors in each CE. Where only one CE is available, C = 1, then $Q_1(W, 1) = 0$.
In developing a grid speed-up metric it is necessary to state what the metric actually means. Typically speedup is calculated with respect to execution on one processor; this is essentially the case for Equation 2.4, but in this case the base reference needs to be taken in the context of the performance obtainable from one CE. Equation 3.3 gives the speedup ($S_{C,p}$) and efficiency ($\varepsilon_{C,p}$) when using C CEs instead of 1. By evaluating the overheads and the workloads one can determine if it is feasible to run the application across multiple grid CEs.

$$S_{C,p} = \frac{T_{1,p}(W)}{T_{C,p}(W)}, \qquad \varepsilon_{C,p} = \frac{T_{1,p}(W)}{CT_{C,p}(W)} \tag{3.3}$$

The above equations for speedup (S), and efficiency (ε) make no distinction regarding the programming model. It can be seen that performance will be degraded unless the serial overheads (i.e. communication) are surmounted. This can be achieved by using better communication infrastructure [59, 47], and/or reducing communication.

3.4 Exploration

The question this thesis explores is whether there is a methodology whereby applications can efficiently share data, transparently to the programmer, and across a number of distributed sites in a grid environment that exhibits (relatively) large latencies. It is investigated whether there are applications that can employ latency-hiding techniques, such as relaxing memory consistency, overlapping communication and computation, and user-driven multi-threading. To do this a DSM system, SMG (Shared Memory for Grids), is implemented as a tool for the investigation.
The traditional DSM implementations have to date been focused on the cluster environment. While garnering lots of research focus in the 1990s, they were not a success, as the embedded overhead was difficult to amortise over the duration of the application. Coarser-grained applications that involve processing large amounts of data relative to the communication demands may be more latency-tolerant and should prove a good fit for executing in a grid environment.
In addition, it is asked whether grid-enabled applications can benefit from a hybrid approach that combines two programming models, i.e. message passing and shared memory (provided by DSM), within a program. The inherent drawbacks of one model might be leveraged by the benefits of another, thus lowering the programming burden and costs. Mixed-mode programming models have already been employed successfully in systems consisting of a large number of multiprocessors [19], where a shared memory approach

is used locally for each shared-memory multiprocessing node, while message passing is used for the communication between nodes.
Such a hybrid approach could promote the use of a shared memory paradigm on a grid where otherwise the inefficiencies would prohibit it. Applications written in a shared memory style are easier to write, and subsequently they might be ported to the grid by identifying the shared object and/or code locations responsible for communication hot-spots and converting these to explicit message passing. Potentially, the process of changing from one model to another need only be done in a localised fashion, i.e. the only sections of a program to be converted to message passing would be those that would exhibit a significant performance gain. This might even be done in an incremental fashion dictated by the laws of diminishing returns.

CHAPTER 4

DSM

To recap, Distributed Shared Memory (DSM) is a concept that attempts to combine the advantages of shared memory and distributed memory systems into one parallel programming paradigm. It combines the single address space of the shared memory model into a simplified programming environment with the scalability and cost effectiveness of multicomputer systems. In essence the goal of any DSM is to extend the shared memory paradigm into distributed-memory platforms so as to provide the illusion that a shared variable is physically shared by all threads of execution in a multi-process multi-threaded application. This illusion comes at the cost of reduced performance compared with a message-passing implementation due to the computation and communication overhead associated with the DSM system. This will discussed in detail in Chapter 6. There are, however, some applications where DSM is more applicable than an equivalent message-passing version as the algorithm may lend itself to be more efficiently implemented.
Different approaches to solving the problems associated with constructing DSMs involve a trade-off between the hardware and software facilities available. One extreme, hardware-only DSMs (H-DSM), employ the use of specialised hardware interconnect for latency hiding and/or data consistency, while software-only DSM (S-DSM) uses only software techniques. There are hybrid schemes that attempt to mix both. Appendix B lists previous DSM implementations according to these classifications. As the thesis relates to grid environments we consider software-only approaches.
The implementation of a software-only DSM (S-DSM)[1] system involves many, and potential conflicting, considerations [42]. This chapter examines such implementation issues. There are four primary questions to be answered in order to implement DSM. The first is what underlying protocol will be used to allow access to shared state. Second, how to ensure that shared state is kept consistent across all nodes when there are competing accesses to a shared variable. Third, how is the location of a shared data item to be found when it is required but not available in the local cache. Last, how can communication between nodes be best minimised in the implementation of the previous issues.

[1] From here on the terms S-DSM and DSM will be used interchangeably

4.1 Shared Memory Access Patterns

DSM allows for a variable to be accessed by many threads of execution that may not reside on the same physical machine, while at the same time these accesses should be transparent to the application developer. When a read or write occurs memory should be consistent according to the consistency model (rules) of the system (see Section 4.5). Providing this functionality is the job of the DSM system. With some applications this can place a severe burden on the resources (operating system traps, inter-node communication, DSM system handling). It is important therefore for a developer to consider data access patterns and related actions such as locality of reference. To ease the task, shared memory is often allocated in chunks or regions.
Access patterns to shared regions are an important consideration as better performance will be achieved through optimal use of the available resources. The implementers of the Munin DSM project found that there are characteristic types of shared data that occur in a shared memory application [34]:

- **Read-only:** This type of variable is first initialised, and subsequently only read accesses occur. The incident matrices in the matrix multiply application of Chapter 2 are such an example.
- **Migratory:** this type of variable is accessed by one thread, modified, and then accessed by a different one. This pattern is repeated for other threads in the system. Caching or replicating copies results in little benefit. This type of variable is found in the classical travelling salesman problem (TSP) that is examined in Section 10.1.1.
- **Write-shared:** multiple writers access the shared area concurrently between synchronisation points, but the individual threads modify different sections. Successive Over Relaxation (SOR) (Section 10.1.1) or Jacobi Iteration are classes of problems where a variable of this type may occur.
- **Conventional:** there is no discernible access pattern associated with this type of shared variable. Access is irregular so techniques such as prefetching[2] of shared data are not beneficial. This type of variable occurs in numerical simulation applications such as the N-Body Water benchmark (Section 10.1.1).
- **Synchronisation:** these variables related to synchronisation actions that are required to synchronise threads. Often synchronisation accesses are viewed in isolation to normal shared memory accesses.

The varied nature of these access types indicates the memory-access flexibility required by the DSM system. The flexibility must be supported by the shared memory modes that govern the access to the shared data.

[2]Prefetching is a consumer-initiated technique, as opposed to *remote write* which is producer initiated, that moves data close to the process before being actually required.

4.2 Shared Memory Access modes

A shared memory access mode specifies the number of threads of execution that can read or write to a shared variable at a given time for a given coherence unit size. Accessing a shared memory region is conditional on the consistency model governing the shared variable. The most general classification of access modes bears a strong relation to Flynn's Taxonomy of computer architecture (discussed in Section 2.2). The four types of mode are:

- **SRSW (Single-Reader, Single-Writer):** there can be only one reader or writer accessing a shared area at any one time no matter how large the shared area.
- **MRSW (Multiple-Reader, Single-Writer):** multiple threads can read access the shared variable concurrently, but only one can write to it. Replicated data copies can exist in each of process' address space, but only one writer can access the shared region.
- **SRMW (Single-Reader, Multiple-Writer):**. only a single reader is allowed access, but multiple writers are allowed to concurrently write to the shared memory area, providing that no two threads write to the same location in shared space concurrently, i.e. all writes must be non-conflicting.
- **MRMW (Multiple-Reader, Multiple-Writer):**. multiple readers and writers are allowed to access the shared variable concurrently. The non-conflicting write proviso of the previous mode applies in this case also. This mode is required for the implementation of generalised parallelism that involves access to the shared data locations by many processes.

Having an access mode that only allows a single thread to access a shared area results in a loss of parallelism, although it is possible for processes to buffer computations and access shared memory serially. If only a single reader is supported, and there are N potential readers, the read access must be done in a serialised fashion, thereby increasing the duration of the parallel operation. The same situation arises for non-conflicting parallel write operations. An example where this situation occurs is given in Appendix F where the resultant data for matrix multiplication is gathered from other processes in a serial fashion by a single writer that is allowed access.
It will be seen later in the thesis that the provision of a multiple access mode significantly complicates the design and implementation of a DSM. This is, however, is a base requirement, as not providing modes that provide concurrent access diminishes the potential for parallelism [60], especially so in a grid environment. Often the mode is dictated by the chosen memory consistency model (see Section 4.5) or vice-versa. Some modes are incompatible with a given consistency model, e.g. a multiple-writer mode is incompatible with the strict consistency model.
In previous DSM implementations the provision of multiple DSM modes that can adapt dynamically at run-time were shown to result in significant performance gains [61], but such a scenario is difficult to implement [62].

4.3 DSM Data Distribution Algorithms

The DSM data distribution algorithm coupled with a DSM ownership management algorithm (next section) defines the base functionality required for implementing a distributed shared memory. These were originally derived from the cache coherency protocols of early hardware shared memory multiprocessor systems [63]. The ownership management algorithm specifies how to find the owner of a shared data item, and the DSM data management algorithm specifies how the shared data is distributed. Each is an important consideration as it affects the number of control messages that are generated.

The four basic DSM data distribution algorithms [63] are described below. There are also modified versions not described here. The algorithms can be categorised by whether or not they (a) migrate ownership of data, and (b) replicate data. The associated cost functions for each algorithm consist of two components $C_{ost} = a \times b$, where a is the probability of the access to remotely located data, and b is the average cost of accessing the remote data item. The parameters for the following cost functions are defined in Table 4.1 below. These basic algorithms have been extended to allow for fault tolerance [64], with little additional overhead for the central-server & full-replication algorithm.

- **Central-server:** with this algorithm the owner of shared data never changes. With every read/write to shared data a request is sent to a central server. The server responds with the valid data. Thus two messages are required for each request. The primary problem with this approach is that the server becomes a bottleneck, having to service requests from all processes. A potential solution is to statically distribute the shared data among a number of servers, but a requesting process will then need to know the location of the data. The cost for the central server algorithms is:

$$C_c = \left(1 - \frac{1}{S}\right) \times 4p \tag{4.1}$$

- **Migration:** the ownership of the data is transferred upon receiving a request for the data item. When a process relinquishes ownership of a shared memory item the identity of the process that it transfers ownership to is recorded. In this way it is always possible to ascertain what process is the owner of the item. Data is transferred among processes in blocks of a defined granularity. This scheme is most advantageous when a data block is used predominantly accessed by a single process. If, however, it is accessed by a number of processes then 'thrashing' of the block will occur. One additional requirement is that as the ownership of a block is transient an efficient algorithm is required in order to find the current owner.

$$C_m = f \times (2P + 4p) \tag{4.2}$$

- **Read replication:** the main disadvantage with the previous algorithms is that only one thread of execution may access data at any one time, i.e. they implement

SRSW modes. With this algorithm data is replicated at different nodes allowing different threads to read concurrently, eliminating much of the communication overhead associated with the previous algorithms.
When a read to a shared data item occurs and it cannot be satisfied locally, then a copy is sent to the requester; at this point ownership may or may not be transferred. When a write occurs data consistency must be maintained according to the consistency model (see Section 4.5). This algorithm implements the MRSW mode. The management of shared data can be distributed across multiple nodes in order to eliminate any potential bottlenecks.

$$C_{rr} = f' \times \left(2P + 4p + \frac{Sp}{r+1}\right) \tag{4.3}$$

- **Full-replication:** the full-replication algorithm implements a MRMW mode, whereby unlike read-replication, full-replication allows for data to be replicated while written to, with the proviso being that only non-competing writes can occur. Reads accesses occur in a similar manner to the previous algorithm, while write accesses are broadcast to other nodes. The order of sequencing writes in order for data consistency to be maintained is left to the consistency model (see Section 4.5).
  Examining the cost function below shows that there are $S + 2$ messages for every write, where S is the number of remote caches of the variable. Instead of performing this action for every write operation an optimisation is for all writes to be logged, then the shared memory is only updated in a node when a write occurs locally.

$$C_{fr} = \frac{1}{r+1} \times (S+2)p \tag{4.4}$$

Parameter	Definition
p	The cost of a zero-size packet event (latency)
P	The cost of a large packet event (latency & bandwidth)
S	The number of nodes sharing the data
r	Read/Write ratio,or access pattern to a granularity unit
f	Probability of an access fault on a non-replicated data block
f'	Probability of an access fault on a replicated data block

Table 4.1: *Parameters for DSM algorithm cost functions*

4.3.1 DSM Ownership Management Algorithm

Depending on whether or not data migrates and/or is replicated, the owner of the shared data master version, and copies if they exist, must be located when required. The DSM

ownership management algorithm is responsible for doing this and is closely related to the DSM data distribution algorithm. A number of ownership management algorithms for implementing DSM were identified by Li [65]. The two main classifications arise from the decision whether to centralise or distribute management. The main approaches are:

- **Centralised Manager:** A central ownership manager is responsible for synchronising all accesses to shared memory. It must maintain records of the existence of all replicated copies of data, i.e. a copy-set. When a process requires data it will direct its request to the manager, who in turn will forward it to the current owner of the data.
- **Improved Centralised Manager:** Differs to the previous algorithm in that it doesn't synchronise access to the data but maintains only a copy-set and a record of the current owner. All requests are still directed to the central manager.
- **Fixed Distributed Manager:** To reduce the potential for a bottleneck that may arise from centralised management, multiple managers are established, each with responsibility for a subset of the shared memory address space. A hashing function is normally used to provide the mapping between processes and shared memory [66]. When a process requires access to a shared memory area, the request will be directed towards the appropriate manager.
- **Broadcast Distributed Manager:** Here each process manages the pages that it owns. A message is broadcast when access to a shared memory is required, and the current owner will then respond. A write broadcast results in all nodes invalidating their copies and ownership being transferred to the requesting process, while a read broadcast results in a copy of the data being sent to the requester and the copy-set being updated.
- **Dynamic Distributed Manager:** Here each process maintains a probable owner (*prob_owner*) field which is updated upon every transfer of ownership. When a process requires access to a shared memory area it will direct its request to the process it believes is the current owner, i.e. to the *prob_owner*. The copy-set only exists on the process that owns the shared location. When ownership is transferred the copy-set is also transferred.

These ownership management functions are interdependent with the DSM data distribution algorithms. If the ownership of a shared block does not change, as occurs with the central server algorithm, then the management function is trivial, and the owner can be identified immediately when required. However if migration occurs, then there is a possibility of a bottleneck at the ownership manager, as is the case with the full-replicated DSM data distribution algorithm, then a distributed ownership manager is required. Fixed ownership is an expensive solution (communication occurs on every write) and due to this fact it is a constraint on parallel computation, so much that it renders it an unattractive solution [65] for DSM.

4.4 System Ordering

According to [67], a distributed system is any set of processes that communicate by message passing and carry out desired actions over time. The notion of time is a fundamental concept for any sort of distributed system. In a DSM certain events need to be ordered so some concept of a sequencing relationship is necessary, i.e. a lock can only be acquired *after* it has been released, or an update to a shared memory location cannot be propagated to remote caches *before* the updating write access occurs.
However, discrepancies can occur between the local clocks of a distributed system for a multitude of reasons [68], as clocks cannot be synchronised perfectly across a distributed system [69]. Hence it cannot be assumed that events are ordered uniquely by time by reference to the local clocks. A solution was proposed by Lamport [70], the *happened-before* relation, whereby:

> *If a process p_i, observes two events then the order in which they occurred will be the order in which they were observed to occur.*
> *An event must occur before the event can be responded to, i.e. a message must be sent before it can be received.*

Distributed locks and shared memory require 'Lamort' logical clocks to identify that an event has occurred. In some of the DSM implementations described below a shared memory object, or page, have an associated logical time-stamp that was incremented when a certain type of event occurs.

4.5 Memory Consistency Models

The memory consistency model used by the developer is effectively a contract between the application and the DSM system, whereby the DSM guarantees that if the software conforms to the agreement then the shared memory is correct, or consistent, in the event of parallel accesses [71]. In other words the consistency model specifies when the shared memory is valid, and the application and DSM agree to adhere to that.
There are two main categories of memory consistency model, namely those that utilise synchronisation points/operations to specify when shared data becomes consistent (relaxed) and those that don't (strict). There are a number of formally defined consistency models across the memory consistency spectrum between the strict and relaxed consistency extremes. As consistency is relaxed the elapsed time between resolution of potential inconsistencies between copies of the shared data is extended. By permitting these temporary inconsistencies the more relaxed methods increase performance due to a reduction in inter process communication [72]. There are weaker consistency protocols available than those covered here, namely Sections 4.5.6 & 4.5.7. These models (where consistency is guaranteed for a bounded period) are unsuitable for use with distributed memory, but are used extensively maintaining consistency in file and web servers [73].
The following sections describe a number of the formally defined consistency models that have been defined in previous research.

4.5.1 Strict Consistency

The most rigorous of all consistency models is strict consistency. It is the model assumed by serial programs for the trivial case of uniprocessor systems. In a multiprocessor all write operations must be immediately visible to all processes. The formal definition of strict consistency is:

> *Any read to a memory location x returns the value stored by the most recent write operation to that same variable*

The implementation of strict consistency is all but impossible in a distributed system as the notion of Newtonian global time applies [74], where an access at any process is required to be seen instantaneously by all other processes. When a read operation occurs the correct value at that exact point in time must be returned no matter how quickly a write may be subsequently performed.
This is illustrated in Figure 4.1. The variable x is initially located at process 1 (P1), which initialises it, W(x), with the value 1, then subsequently updates the value to 2. Process 0 (P0) initiates a read operation on x, R(x), and a request is directed to P1. In order for strict consistency to be maintained P0 must be returned the value 2. Due to the high latencies of communication between the processes this may not be and possible the read access may return the value 1.

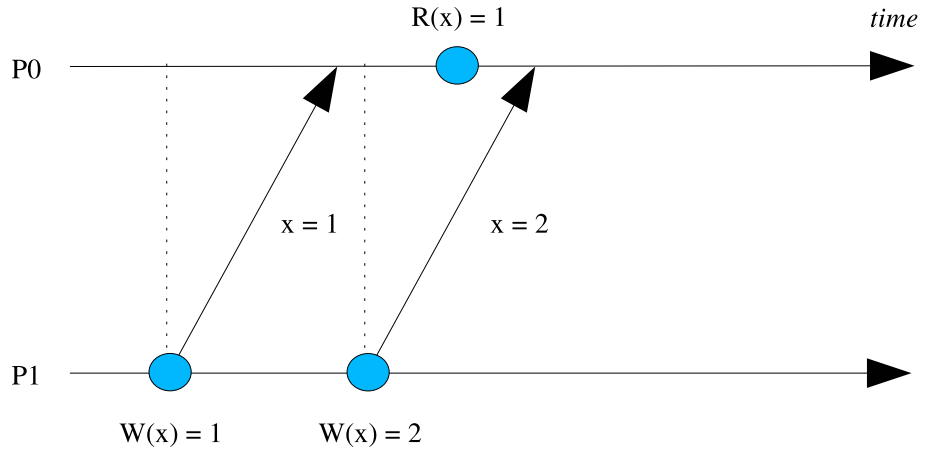


Figure 4.1: *Failure to adhere to Strict Consistency due to communication delays*

4.5.2 Sequential Consistency

Sequential consistency is a weaker model that does not assume Newtonian global time, and mostly provides enough consistency for general usage. Programmers, if properly trained in parallel application development, can easily adapt to a situation whereby

statement execution order is irrelevant. However, it is still in the category of strict models. First defined by Lamport [70], a system is sequentially consistent if:

> *The result of any operation is the same as if the operations of all processes were executed in some sequential order, and the operations of each individual process appear in the order specified by its program.*

A system is defined as sequentially consistent if the result of any execution is the same as if the operations were interleaved so long as all processes see the same sequence of memory accesses [74]. A sequentially consistent system does not guarantee to return a value consistent with its state conforming to Newtonian global time, but guarantees to process memory accesses in a sequential order. An example is shown in Figure 4.2.

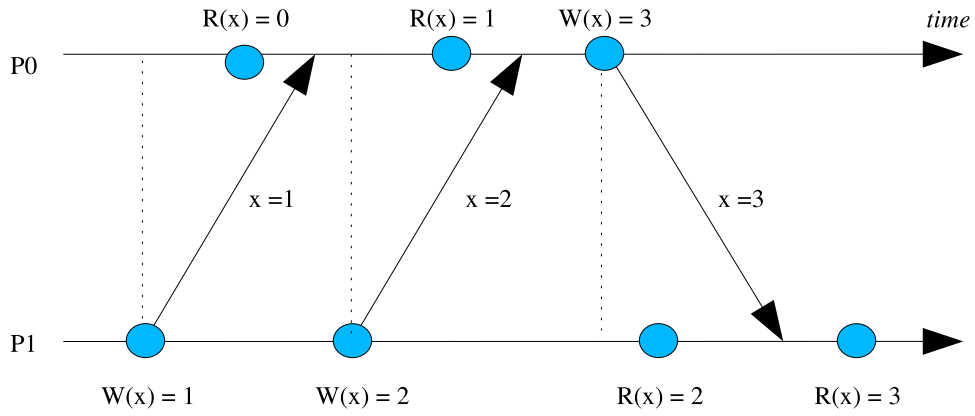


Figure 4.2: *Sequential consistency*

4.5.3 PRAM and Processor Consistency

PRAM (Pipelined RAM) and processor consistency models are similar enough that they are often regarded as equivalent [74]. These consistency models allow concurrent writes from different processors to be seen in different orders by different processes. Time-dependent accesses can also be seen in a different order by different processes. Writes from the same process must be identically and correctly ordered (pipelined) by all processes. These models are also categorised as strict models. The formal definition of PRAM consistency is [75]:

> *The write operations performed by a single process are observed by other processes in the order that they were performed, but the order in which write operations from multiple processes occur can be seen differently*

In effect a writer does not have to wait for all modification to reach other processes before it initiates another write operation. In Figure 4.3 it can be seen that writes observed at

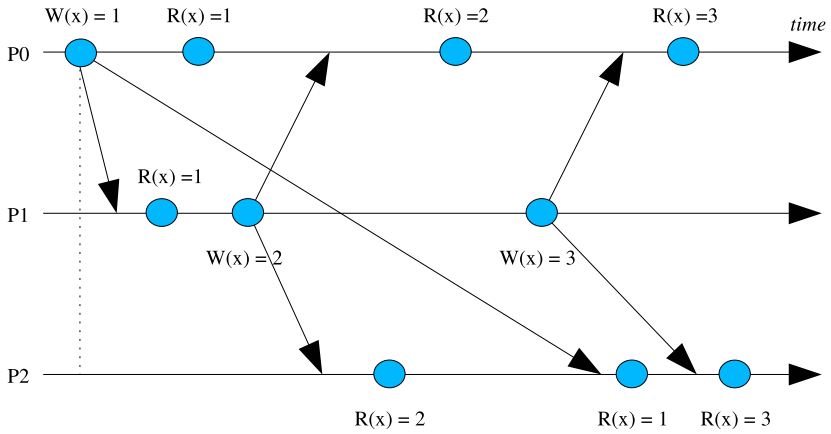


Figure 4.3: *PRAM consistency*

Process 2 (P2) may be inconsistent with stricter models when compared with what is observed at other processes at the same time, but all writes from the same process are observed in the order that they occur.

4.5.4 Weak Consistency

The previously mentioned consistency models are quiet restrictive in that they require all writes from a single process to be ordered and viewable by other processes [74], resulting in excess communication. Weak consistency[3] assumes that if all writes can be propagated to all remote processes at a certain *synchronisation* point then this restriction may be diminished. Weak consistency is categorised as a relaxed model. Relaxed consistency models require the programmer to access shared data in a more structured fashion, thus reducing the volume of network traffic generated, and increasing performance [66].
With weak consistency the task of making memory globally consistent is tied to the use of synchronisation primitives. When a synchronisation operation occurs all writes performed by a process are propagated to remote processes, and all remote writes are applied locally. Hence, there is a clear distinction between ordinary memory accesses and synchronisation accesses. Weak consistency has the following formally defined properties [76]:

1. *Accesses to synchronisation variables must be sequentially consistent.*
2. *No accesses to a synchronisation variable is allowed to be performed until all previous writes have completed everywhere.*
3. *No data accesses are allowed to be performed until all previous accesses to synchronisation variables have been performed.*

[3]Weak consistency has also been used to refer to any consistency weaker than strict consistency

As depicted in Figure 4.4 all processes see synchronisation accesses in the same order. When a process is accessing a synchronisation variable then no other process can access it. Before a process is allowed access to the synchronisation variable all preceding writes must have completed, so by the time access is granted all the writes are guaranteed to have been completed. The final condition means that before an ordinary access is allowed to occur then the preceding synchronisation accesses must have been completed. Shared memory is only brought up to date when a synchronisation variable is accessed.

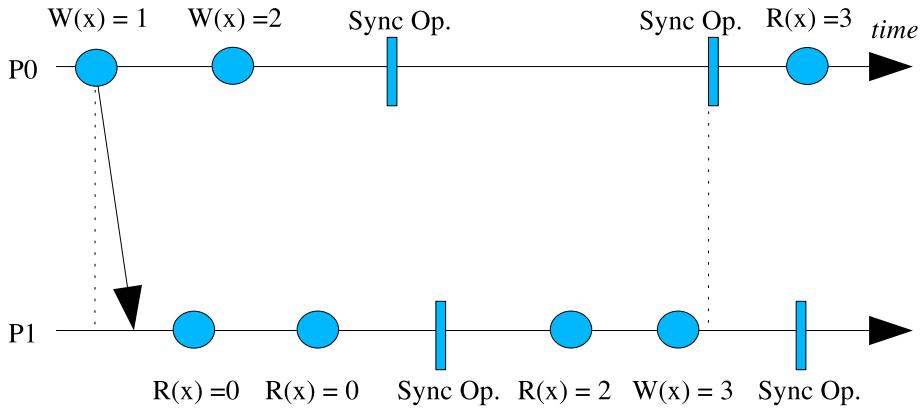


Figure 4.4: *Weak consistency*

4.5.5 Release Consistency

The main drawback with weak consistency is that there is uncertainty concerning the status of shared memory when a synchronisation access occurs. Is it about to be written to, or has it just occurred? Due to this uncertainty all actions must occur: all local writes must be flushed to remote processes if they exist, and all external writes must be applied locally. Thus the synchronisation operation has a global effect for all shared variables.
With release consistency (RC) this problem is removed by identifying synchronisation operations as being either the entrance or exit of critical sections, within which shared data is accessed, although the operations still have a global effect. These actions were termed *acquire* and *release* by the first implementers of release consistency [77]. Acquire actions define the entering of critical sections, while release actions specify the leaving of a critical section. It is the job of the programmer to instrument the application code with these synchronisation operations, be it ordinary operations on special variables or as special operations [74]. The formal definition of release consistency is:

1. *Before an ordinary access to a shared variable is performed, all previous acquires done by the process must have completed successfully.*

2. *Before a release is allowed to be performed, all previous reads and writes performed by the process must have been completed.*
3. *The acquire and release accesses must be processor consistent.*

In effect accesses to shared data are batched, with acquire signalling the start of the batching, and release its end. Figure 4.5 shows the sequence of events that demonstrates the action of a release-consistent system. A typical release consistent DSM system would be constructed using shared distributed locks, where the locking process is equivalent to acquire, and unlocking to release. Global barrier primitives may also be used whereby arrival at the barrier is equivalent to a release operation, and the departure from the barrier to an acquire.

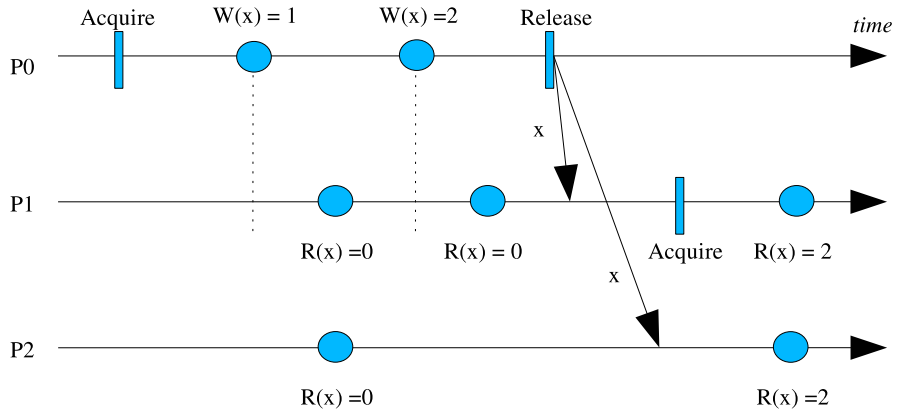


Figure 4.5: *Release Consistency*

4.5.6 Lazy-Release Consistency

A negative aspect of Release Consistency is that upon calling a release all updates are sent to all processes with a cached copy of any modified shared data. However, not all of these processes may require the invalidate/update notice (they may not be actively reading the data), thus there is potentially superfluous overhead. Lazy-Release (LRC) extends the principles of release consistency by delaying the pushing of invalidate/update information until it is actually required. When a release occurs no communication is generated. At a subsequent acquire the modifications necessary are directed to the acquiring process [78]. The conditions that must be met to guarantee lazy release consistency [79] are as follows:

1. *Before an ordinary access to a shared variable is performed with respect to another process, all previous acquires by the process must have completed successfully with respect to that process.*

2. *Before a release is allowed to be performed, all previous reads and writes performed by the process must have been completed.*
3. *All synchronisation operations must be sequentially consistent with respect to one another.*

Figure 4.6 illustrates the difference between the two types of release consistency. When the release operation occurs no communication occurs until the subsequent acquire, in contrast to the previous model where it does.
Some inefficiencies are introduced as an acquiring process with an out-of-date copy of the data must fetch the data from the current owner. This introduces a stall before computation can begin. Prefetching has been used to attempt to overcome this [80], this is instigated by an operation programmed into the application. Such actions must be application driven, as prefetching and lazy release are opposites, so any attempt to automate the prefetching is likely to cancel much of the benefit accruing from lazy release. A possible solution is for the system to adapt to the data usage pattern and so only actively used data will be prefetched.

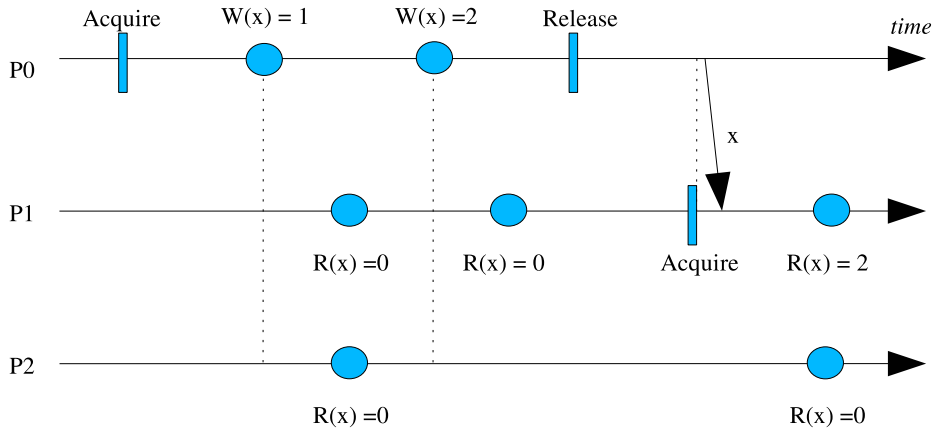


Figure 4.6: *Lazy-Release Consistency*

4.5.7 Entry Consistency

Even an efficient model such as LRC still generates a large volume of individual messages, much of this is due to the global effect of the synchronisation operations. Since an acquire intended to enforce consistency on one variable will also have the latent effect of enforcing consistency on all other variables that have been modified, even if that is not necessary. This is a result of having, in effect, only one global synchronisation variable.
Entry Consistency (EC) is one attempt at reducing this problem by closely binding a shared memory region/block to a specific synchronisation variable, i.e. by allowing more than one synchronisation variable, each covering a subset of the shared variables. In

a similar fashion to the previous models, that render shared memory consistent when synchronisation operations occur, EC will do likewise, however, only shared memory bound to the synchronisation variable is made consistent. The numbers of messages are also reduced as the update data can be piggybacked upon synchronisation messages. Stalling of the application, due to waiting for memory consistency to be enforced, is reduced as well. The explicit association of shared data with synchronisation variables creates extra burden for the programmer. This is the downside for the reduction in coherence message generated.
A memory system is Entry Consistent if the following conditions can be met [81]:

1. *Before an acquire access to a synchronisation variable s is allowed to perform with respect to any process p_i all updates to shared variables guarded by* s *must be performed with respect to that process.*
2. *Before an exclusive access to a synchronisation variable* s *by a process p_i, then no other processor may hold s in non-exclusive mode.*
3. *After an exclusive access to* s *has been performed, any processor's next nonexclusive mode access to that synchronisation variable may not be performed until it has been performed with respect to the current owner of the synchronisation variable* s.

Entry Consistency only guarantees that when an acquire operation on a synchronisation variable occurs, the data bound to that variable is made consistent. Figure 4.7 demonstrates this, where a synchronisation variable, z, is acquired and released by process P1. The data variable x that is bound to z is updated upon an acquire of z at a remote process, while the modifications to a variable that is not bound, y, but modified in the same interval as x, is not (under RC or LRC it would be).

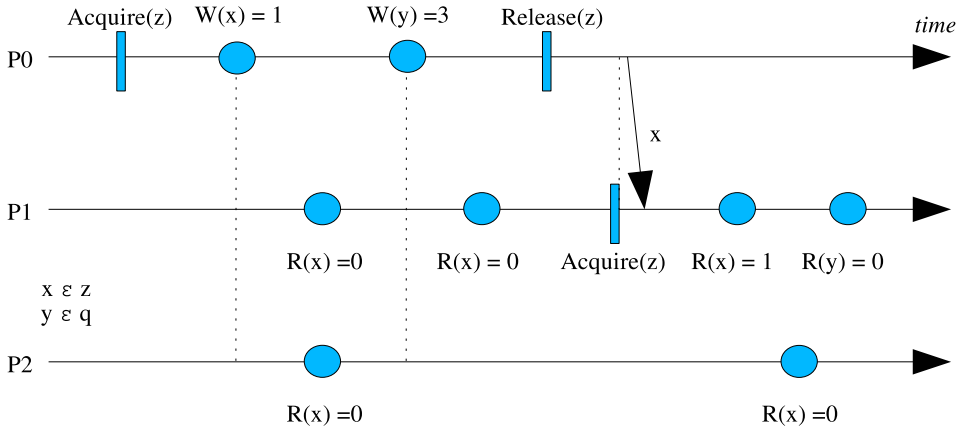


Figure 4.7: *Entry Consistency*

4.6 Memory Coherence Protocols

When a read occurs, a copy of the shared region may be sent to the requester without affecting the status of the data at the responder. However a write results in potentially inconsistent data being resident at remote processes. Thus a data consistency enforcement method to maintain consistency is required, whereby after a write any subsequent accesses act on consistent data. A memory coherence protocol defines how this is done (it does not say when or why it is done). With distributed shared memory there are two main classifications of coherence protocols [82, 66]:

- **Write invalidate** When shared memory copies located in remote processes are no longer valid, they are invalidated by the regions' owner, i.e. their status is changed to 'invalid'. This requires one invalidation message to be sent to all processes that hold the relevant data. When a remote process subsequently accesses the data an access fault occurs, and a valid copy of the data must be fetched from the current owner.

- **Write update** In contrast, the update protocol propagates the writes to all remote copies of the data. Potentially fewer messages result, but the message payload will be greater. This protocol is best suited to situations where the granularity of a shared variable is small (typically equal to the MTU of the underlying transport system), or where the probability of access by a remote node is high. When a remote process accesses its copy of the data, it finds that the variable is valid and no access fault occurs.

Although both memory coherence protocols may be utilised, in certain circumstances one may be more suitable than the other. The decision on which to employ depends on a number of factors: memory access pattern, latency and bandwidth of the interconnect (see Section 5.1, in general low-latency small messages favour the invalidate protocol, while high-latency large messages favour update); required consistency model, and application demands. Some of memory consistency models, described previously, may have a greater affinity to one coherence protocol and this is often a mitigating factor, e.g. Entry Consistency is more suited to an update coherence protocol, with the caveat that when the shared memory mode is MRMW this may not be the case.

4.6.1 Home/Home-less Coherence

There can be slight variations of the above protocols, such as delayed-write update [83], but the most accepted classification of the coherence protocols mentioned above is based upon whether or not each shared unit (see Section 4.7) of the shared memory has an assigned home[84]. In the latter case, termed *home-less*, and is only really applicable to the invalidate protocol; there is no fixed home process (the protocols as described above are examples of this case). In the *home-based* approach, each unit (in page-based systems this will be on a per-page basis) has a predefined home, where modifications are always flushed on the shared region's consistency-related release event. Unless the home is

chosen carefully, or dynamic in nature, poor performance will result, since modifications are always propagated. Of the advantages of home-based protocols enumerated in [85] only one is beneficial when compared to an update protocol, i.e. home nodes are not required to make write notices (diffs). The AllCache System [86] posed another: that it is helpful to have a *home of last refuge* that a block may be flushed to in order to free memory for more urgent use.
Certainly there are situations where this approach will perform better than another [87, 88], but the challenge is to find the one that best fits the general case. Any home based flavour of one of the above protocols really just imposes a fixed ownership scheme (see Section 4.3.1) where all coherence actions are always directed towards the home location of the memory block. Efficient schemes exist for lock synchronisation primitives using home based consistency models [89].

4.7 Coherence Granularity

All storage is usually perceived to be in a hierarchy, ranging from very fast but small CPU cache storage to very slow but enormous bulk storage. DSM is a more general form of storage than CPU cache, but the design of a DSM system echoes that of a CPU cache. In a DSM, when an access occurs to a shared memory variable (location) that the process does not have a copy of, then the variable must be fetched from its current location. In traditional CPUs, when a read occurs and a cache miss occurs, the data must be fetched from its current location. Due to the phenomenon of locality of reference a block of memory locations (a cache-line) are treated as a unit, and are read in at the same time (cache line size size varies depending the CPU architecture). DSM granularity, similar to cache-line size, specifies how much data is treated as a unit, and involves a complex trade-off between locality and coherence overhead [65].
The quantum of this data sharing unit (Granularity) is a major determinant on the overall system. Small granularity may result in smaller message payloads, but the volume of messages may increase, and more overhead occurs in order to witness and respond to an access. The granularity is also the minimum level at which an access can be registered. There will be one DSM access event for every accessed unit of granularity within the shared area.
The granularity units that have been implemented in previous hardware and software DSM implementations are given in Table 4.2 below.

In general one of the first two granularities have been chosen for hardware DSMs where the interconnect efficiently provides for large volumes of small coherence transactions, while software implementations have used the latter three with less specialised interconnects. Tables B.1, B.2, B.3 in Appendix B effectively illustrate this point. Some of the important issues involved that have a significant effect on DSM computation overhead [83], and so influence the decision on the granularity size are:

- **False sharing:** Having a large granularity can result in false sharing, where more than one process writes to the same unit of granularity concurrently (even with the

System-Word	A 32-bit or 64-bit architecture-dependant system word.
Cache-Line	A multiple of the system-word size. (64 bytes with current CPU architectures)
Page	The size of the native operating system's virtual memory page, normally 4KB or 8KB
Variable	A user defined size that can be in the range from a byte to whatever maximum size the system resources/operating system can support
Object	Again this user defined, but with semantic meaning determined through use (data structures) [90].

Table 4.2: *Possible DSM Granularity Sizes*

writes are non-conflicting) there is a ping-pong effect with exchange of ownership between the writers, reducing the potential concurrency. The solution is to support a multi-writer protocol and/or to have a smaller granularity unit, thereby reducing the probable number of writers per unit.

- **Access detection:** Detecting accesses to shared memory locations can be a tremendous overhead on operating system resources, as in most cases page faults are used to detect accesses in S-DSMs, so if the unit of granularity is larger, fewer access events occur. *Write Trapping* is the term given to detecting writes, for every individual unit of granularity where writes occur there will be a detection event.
- **Write collection:** When writes have been detected, remote processes with a copy must be notified according to the consistency model and coherence protocol.
- **Communication:** As is the case with access detection the smaller the granular unit level the greater the usage of system resources, in this case larger volumes of individual communication transfers. Ultimately there is a trade-off between message payload size and message volumes.

Chapter 7 lists some metrics that can be used to determine the best fit for the granularity level. Some of these costs are: page-fault handling; DSM system access latency; communication metrics such as latency and bandwidth; system page size (in a heterogeneous environment this may be a problem as not all platforms share the same page size). The choice of granularity may not be present as it may be dictated by other criteria. It is worth noting that a coarser grained unit such as a page offers a model which is similar to physical shared memory with the downside of a performance penalty [82].

4.8 DSM Case Studies

Systems composed of stricter consistency models tend to impose less burden on the application developer, as the programming semantics are less convoluted. The trade-off,

however, is that there is a significant increase in the volume of control messages and a greater amount of operating system resources are consumed. Stricter consistency models also tend to reduce potential concurrency as only single writer protocols have been implemented. Multiple writer protocols can be utilised to offset this [56, 61].
In this section some previous DSM implementations will be briefly explored with regard to the various approaches taken to the topics discussed in the previous sections, particularly in relation to weak consistency models. A summary of the features of other DSM implementations is given in Appendix B. The salient DSM design decisions are given. A description of the APIs for some of the DSMs are also given in Appendix C. Although there are some DSM implementation that are more recent, it can be argued that those outlined below are still the seminal works in the field.

4.8.1 Munin

The primary goal of the Munin DSM project was to allow applications written for a multiprocessor system to execute efficiently in a distributed memory system with minimal modifications to the source code [34]. Additionally the programming interface should have similar semantics to that used when using true shared memory multiprocessors, while at the same time not restricting the programming language.
Munin was one of the first DSMs to implement Release Consistency [91] (discussed in Section 4.5.5). The developers of Munin implemented eight tunable parameters that govern the nature of the consistency protocol. (i)Invalidate or update, (ii)Replicas allowed, (iii)Delayed operations allowed, (iv) fixed ownership, (v) writable, (vi) multiple concurrent writers, (vii) stable sharing patterns, and (viii) changes flushed to owner.
The main Munin DSM components consist of: the run-time engine which handles synchronisation and consistency events such as page faults, the object directory which managed the shared data that is in use locally, and the Delayed Update Queue (DUQ) which is responsible for maintaining consistency, in this case release consistency.
Munin was written in C, and although one of the primary design goals was to avoid using language, hardware or OS features not readily available on a wide variety of systems [92], in addition to a custom compiler, an OS with custom Munin extensions is also a requirement. The extensions made to the default development System V platform included the ability to handle segmentation (SEGV) faults and other protection violations in user rather than kernel space, the ability for a user process to manipulate arbitrary virtual memory mappings, and the ability to create and destroy processes on remote nodes so that a cloned image of the calling process (initialised data etc.) can be replicated in the new process. Additionally System V's IPC mechanisms are heavily relied upon. All communication in the prototype system occurred over Ethernet.
The main modification necessary in order to port an application to use the Munin DSM is the annotation of the declaration of shared variables using the *shared* keyword, and additionally the type of coherence required should be specified (one of the typical shared memory access patterns discussed in Section 4.1: *write-shared, conventional, migratory and read-only*). Preprocessor support is required to process these user annotations to produce an auxiliary file that lists all shared memory regions in an application. At link

time the variables can be identified from this auxiliary file and are placed on individual system pages, or on pages with the same anticipated access pattern. During run-time when a variable is accessed the system can examine the variable type and determine the coherence action to take. For a write-shared item a delayed update protocol is employed, while for the conventional and migratory sharing schemes an invalidate approach is used. Only statically allocated variables are supported. No routines similar to a global *malloc* exists as with many other DSM implementations. Only variables of the same type may be placed on the same page as decisions taken by the fault handler mechanism are closely related to the consistency model, which is checked at each memory fault. In a similar manner to hardware DSM implementations Munin used directory tables to locate the variable meta-data when an access fault at a particular location occurs. This meta-data comprises information such as variable size, type, locally present, and probable owner. The programming model that is presented to the programmer is similar to that when writing multi-threaded applications for shared memory machines. API calls exist for the creation and deletion of these extra-process threads. Synchronisation accesses are distinguished from variable accesses through the use of library routines, e.g. global barrier and lock/unlock functions are available to access these special primitives.

4.8.2 Midway

The Midway DSM tried to achieve a similar goal to Munin, to ease porting a multiprocessor application to a distributed memory system with minimal changes [74].
The Midway DSM was the first to implement Entry Consistency [81]. Midway included support for multiple consistency models to be employed concurrently (basic support for release and processor consistency models is also mentioned). Parallel applications were developed using a threads model, whereby all threads shared the same address space. The duties of the run-time system included the maintenance of consistency. The Midway API is given in Appendix C. EC requires that all shared data must be declared and explicitly associated with at least one synchronisation object [42]. Binding functions were provided to achieve this (*midway_bind_synch_object*), and also functions to rebind shared memory areas to other synchronisation primitives (*midway_rebind_synch_object*). For a lock synchronisation primitive, Midway distinguishes between two modes in which it could be held: non-exclusive and exclusive. The former allows multiple threads of execution to concurrently hold the lock, while in the latter case only one is permitted. A lock held in non-exclusive access can only be promoted to exclusive access if only the requesting thread holds the lock. However, a lock can always be demoted to non-exclusive mode. Midway allowed a significant optimisation whereby once a thread holds a lock in non-exclusive mode, it can then delegate non-exclusive access to other threads without being the owner of the lock.
Midway could use one of two approaches to detect access to shared regions: the first was to use a compiler that identified a modification to the shared memory region and recorded such an event in an auxiliary table that had an entry for all shared variables. If this first approach was not available, say the compiler didn't support the data-type, or the special Midway compiler was itself not available/supported, then the second ap-

proach was to use the MMU to detect accesses to shared data. The former approach could result in increased performance by reducing the DSM run-time overhead as access to shared memory was detected without the need of a page fault, however, each shared memory type needed to be explicitly supported [93]. The Midway compiler only supported EC, so the other stated consistency models supported (PC and RC) used the MMU method. When the time arose to enforce consistency the auxiliary table was checked for the presence of 'dirty' data. The update coherence protocol was employed as it has a natural affinity with the default Entry Consistency model. One serious deficit with the Midway DSM was its lack of any real support for multiple concurrent writers, thus Midway suffered tremendously from the effects of false sharing.
Midway used a special optimisation to reduce the amount of coherence data transferred at consistency points. Only the modifications performed on a shared variable in the time interval $\{t_a, t_b\}$ were sent to the acquiring process (which had a copy with a modification date of t_a) from the process with the most recent copy (mod date t_b). Midway employed a form of garbage collection, whereby once a shared variable was no longer being accessed locally (indicated by the lack of ownership of the associated synchronisation primitive), it could be discarded. Upon being referenced again the shared data could be refreshed from a remote process.

4.8.3 CRL

The C Region Library (CRL) [94] was an attempt to construct a DSM system that would be highly portable, language and system independent, while at the same time being efficient though minimising the depth of the DSM layer.
The programming model allows the programmer to define an arbitrarily sized contiguous memory region. Following this the region must be mapped into the local process address space using a rgn_map routine. CRL will make no guarantee where in the process' address space a shared region will be allocated, so a region may be located in different locations in different processes. The granularity of writes is whatever size of region that is created. The CRL API is given in Appendix C. It has functions to clearly delineate the beginning (rgn_start_op) and end (rgn_end_op) actions for shared memory regions. No explicit synchronisation operations are used by the programmer to enforce consistency, but in essence the Entry Consistency model is employed.
The coherence granularity is that of the allocated region's size. CRL borrows many of the approaches that a hardware DSM would employ (for a comparison see Table B.1), such as fixed-home invalidate protocol, resulting in high communication costs (potentially three sets of messages per access). A typical hardware DSM directory mechanism is used to locate shared memory home. Support is included for explicit flushing of the local cached copy back to the home process using the rgn_flush call.
CRL requires an underlying messaging system implementation for communication. The CM-5 [95] and MIT Alewife machines were supported by making use of Active Messages implementations and in these implementations proprietary features were also used. A message passing implementation, PVM (described in Section D) is used for start-up of system, with CRL's custom messaging API used thereafter.

The programming model offered by CRL is very arduous to develop with, particularly when the applications tend to have multiple shared regions accessed concurrently. In addition concurrency is reduced as there is a lack of support for concurrent writes. It is efficient in regard to OS overhead in write detection as the programmer has to do this. CRL was the first DSM to run on a relatively large number of processes (128). Conversely, it demonstrated the limitations of DSM scalability for some types of application. However, the actual implementation cannot be said to be a software-only DSM as reliance is made upon specialised coherent interconnects.

4.8.4 Treadmarks

The Treadmarks DSM [96] was the first DSM to implement Lazy-Release consistency. Its designers chose this model for their implementation as they considered that this model resided in the 'sweet-spot' of the consistency-model/control-message trade-off, and would still offer an intuitive programming model with good performance.
The Treadmarks system requires no special kernel or compilers, and as such has been ported to a myriad of platforms, and is itself still in limited use. The Treadmarks API is simple and efficient and has been the basis for many other projects. The API is given in Appendix C(page 221). Shared memory is declared using the *Tmk_alloc* call, which allocates memory from a reserved pool of memory. This memory pool is allocated at run-time and is located at the same memory location in the address space across all Treadmarks processes, which is difficult to achieve in a hetrogenous system. A pointer to a unstructured byte array is returned. The *Tmk_distribute* is subsequently used to broadcast the appropriate data to the other processes in the application.
In Treadmarks synchronisation is implemented using barriers, locks, and condition variables. The barrier primitive is global in nature, and the actual algorithm implemented is the central barrier algorithm. Barrier managers are assigned in a round-robin fashion. As stated above Treadmarks employs an invalidate approach to coherence so the workload is much reduced. The locking routines only provide exclusive access modes. Condition variables are implemented on top of the lock variable. An acquire operation on a synchronisation variable signals the start of an 'interval'. Modifications to shared data for the duration of this interval, the end of which is designated by a release operation, are valid.
A write notice, if required by the coherence mechanism, is sent to other processes upon release consisting of a list of shared data items modified in the interval, and a diff is generated (a write collection method, see Section 4.7) that only encodes the modifications to a shared unit by comparing the run-length of the original and modified versions of a coherence unit (a page in Treadmarks' case). Diffs are only merged upon demand, and it is necessary that this is performed in the proper causal order (by interval vector timestamp).
Treadmarks LRC implementation supports Multiple-Writer protocols [97], allowing multiple threads of execution to concurrently modify data in a shared unit, thereby solving the problem of false sharing. This is supported so long as the modifications are distinguishable from one another, i.e. each memory location (4 bytes in Treadmarks) within

the sharing unit can have only one writer (writes are non-conflicting). As can be seen in Figure 4.8, where multiple writers are in use, modifications to variables within the same shared page are identified at the synchronisation point using write notices. Potential conflicts are determined when the diffs from all writers are merged following a subsequent acquire. Any conflicts result in an access violation.
Treadmarks uses a number of novel features to optimise the DSM run-time. Use of adaptive coherence protocols [98] is one such approach, whereby coherence protocols can *dynamically adapt* between single and multiple writers, thus allowing for a reduction in coherence information whenever possible. Treadmarks also allows for a lazy-diff creation scheme, which allows the process to be delayed until the time the diff is required at a remote node. This has the effect that superfluous diffs are not generated, reducing overhead on the DSM run-time [62].
Communication between Treadmarks processes is achieved using sockets. At start-up of a N process application, the root process will first spawn the remaining N-1 child tasks via a remote shell on remote nodes specified in a machine file. Each Treadmarks DSM process creates N sockets. The maximum number of processes that is allowed in a Treadmarks is fixed at 32, and the total shared memory is also limited to 16384 virtual memory pages (this limit can be modified by recompiling the DSM distribution) that must be located at the same virtual address in all processes. Treadmarks can allocate multiple shared variables on a single page, while offering potential memory space savings the consistency model can be broken unbeknown to the user. The Intel C/C++ compiler has been coupled with the Treadmarks DSM system to provide distributed execution of OpenMP applications [99], see Section 5.3.2.

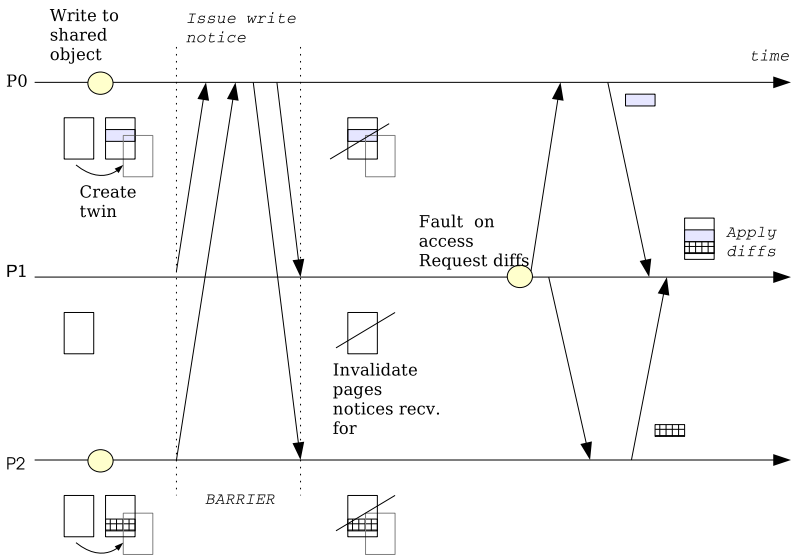


Figure 4.8: *Treadmarks Multi-Writer support*

4.8.5 CVM

With previous DSM implementations the choice of coherence protocol/consistency model was built into the DSM with no variation possible without modifications to the DSM core. The Coherent Virtual Machine or CVM was developed with the intention of being a platform for protocol development and experimentation [100]. CVM was developed by Pete Keleher, one of those responsible for Treadmarks, and is written as a user-level library that supports most UNIX style systems. It is written in C++ and defines interfaces for which different protocols and consistency models can be implemented.
With CVM no further modifications to the DSM core is required for the extension and implementation of new protocols, allowing for rapid development and deployment of new protocol implementations. This is achieved by deriving new classes from the *Page* and *Protocol* base classes. Implementations exist for sequential and lazy release consistencies, and the home and homeless variants of the invalidate coherence protocol.
CVM employs many of the same approaches that were taken in the more recent projects that utilised Treadmarks, such as the underlying end-to-end communication protocol built upon UDP sockets, start-up of remote processes by a remote shell process, and the same limit of 32 processes. However, new communication layers can be implemented by implementing the *Msg* interface. The programmer's API is given in Appendix C, and is also practically identical to that of Treadmarks.
Dynamic CVM (D-CVM) [101] is an extension that allows for the dynamic migration of threads from one process to another, allowing for better load balancing of applications.

4.8.6 Brazos

The Brazos DSM system [102] implemented Scope consistency. Although not described in this thesis, this is a hybrid of release and entry consistencies [103]. This consistency model aims to provide the latter's potential performance advantages with the superior usability of the former, allowing for the reduction in false sharing.
The Brazos DSM was implemented to take advantage of the proliferation of cost effective networks of multiprocessors. The DSM was developed for the Windows NT operating system, and sought to leverage some of its features. The DSM had native support for multi-threading, which is facilitated in the OS with support for preemptive multi-threading, something that had been lacking at that time in the Linux OS. The DSM allowed dynamic run-time performance tuning by reducing the shared memory pages' copy-set sizes by selecting the most appropriate coherence protocol.
Communication between processes was enabled through the use of the WinSock API, and the Brazos DSM made use of novel features such as selective multi-cast sockets for communication, resulting in a reduction of the number of consistency-related messages.
The Brazos DSM was used as the target for an OpenMP compiler, which enabled applications developed with the increasingly popular OpenMP standards to execute on a group of distributed shared memory machines [104]. This project also offered a combined 'common' programming model composed of OpenMP and MPI.

4.9 Review

Chapter 2 developed the rationale (or *why*) of DSM. This chapter dealt with issues surrounding the trade-off in the *when* and *how* it is actually achieved. The SMG DSM is a fresh approach at implementing a DSM, by learning from the inherent defects of previous DSM implementations that have prevented their use in a grid environment (which is reasonable considering most were implemented before the Grid paradigm). Many of the seminal works in the field mentioned in the case studies suffer from trivial design deficits such as support for a (relatively) small global memory space, small process pool sizes, or an awkward API.

When: writes become globally visible as dictated by the consistency model. According to [105, 106] aggressive implementations of the weaker consistency models are capable of delivering higher performance because they better tolerate network delays and limited bandwidth. Figure 4.9 attempts to illustrate that the choice of consistency model is a trade-off in the volume of resulting control messages versus the programming effort. The main protagonists that emerge, LRC and EC, have their drawbacks - LRC: the critical section times are increased as all local caches require the remote modifications (caused by a cache invalidate), EC: the potentially significant and superfluous communication generated.

How: via coherence protocols. The invalidate protocol is best used where a shared object is very large, and where sharing per granular unit is small so the probability of a remote access is small, or the sharing access pattern cannot be established. This protocol is normally employed when the consistency protocol is stricter and a low latency interconnect is present. The protocol has been utilised in a number of DSM implementations that have demonstrated that it is adequate in certain circumstances [66]. Home based protocols are an attempt to solve the inherent lack of scalability of the invalidate protocol.

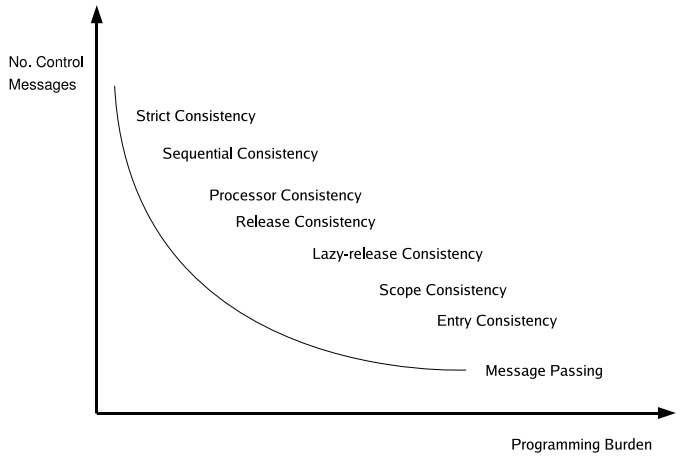


Figure 4.9: *Programming Burden vs Control Messages*

CHAPTER 5

Relevant Issues

The implementation of a shared memory management system that operates in a distributed environment requires the provision of essential services, notably a facility for inter-process communication, as illustrated in Figure 1. Additional issues arise when a distributed application operates in a wide-area-network environment such as a grid. The additional latencies encountered will further exacerbate the overheads, so it is a necessity to have additional services, such as information & monitoring, to aid their reduction.
In this chapter an overview of the enabling distributed technologies and principles are presented. The main topics for discussion are the communication mechanism used between the system threads within the DSM, the method by which environmental information is accessed, and how monitoring data may be efficiently produced. This is followed by an overview of the main alternatives for a parallel programming API that employs shared memory.

5.1 Distributed Communication

The characteristics of the underlying communication mechanism are of prime importance. In order to implement a DSM system, serious consideration needs to be given to the performance of the communication channels between the DSM system threads. There are many factors that need to be taken into account, but the the two primary metrics that characterise communication are bandwidth and latency. They govern the overall scalability of the DSM system:

1. **Latency** is the fixed cost to transfer data, and is independent of the volume of message data being transferred. It is the time taken to transfer a zero byte payload between endpoints (in reality with all interconnects there will be a minimum transfer unit). It is composed of hardware and software delays. The unit of bandwidth is the time taken to perform the transfer, in seconds.

2. **Bandwidth** is the rate of transfer of data. It governs the upper-bound on the total amount of data that may be transferred between two nodes per unit time. The unit of bandwidth is bits or bytes per second.

The values for Latency and Bandwidth will determine the best approach to take for message transfer: a higher message count with a smaller payload versus a lower message count and a larger payload. The nominal time to transfer a message with a payload of n bytes can be derived from:

$$TransitTime = Latency + (n/Bandwidth)$$

Bandwidth, and more significantly latency, will affect the scalability that is attainable by a DSM system. As the number of processes (that an application executes on) scales up, so will the amount of communication, thereby reducing the computation to communication ratio. As was mentioned previously in Section 2.1, if the job is scaled up this effect can be mitigated [11]. When choosing the communication mechanism, some factors to consider are:

- **Overlap:** The ability of a system to overlap communication and computation where possible. This allows user computation to proceed while communication is occurring by allowing a thread to initiate communication while not having to wait for notice of delivery. This functionality is mainly provided to the user through the availability of non-blocking communication calls and autonomous communications.

- **Message/Buffer copy count:** The number of times the message payload is copied between buffers during the overall transfer. In a typical inter-process data transfer, a message has to be copied into communication buffers before it can be transmitted over the wire, with a similar sequence at the receiving side.

- **Multiplex:** The ability for multiple threads to send/receive messages concurrently. This is achieved by multiple threads interleaving fragments of their messages 'over the wire'.

- **Data formatting:** message data may need to be typed by marshalling the data with its corresponding type map before it can be sent/received over the communication channel, since different data representations occur on different architectures.

- **Usability in a Grid:** The ease of enabling the use of the communication library for a grid environment. The use of the DSM should not overly inconvenience the user in having to establish communication channels for every environment.

- **Robustness:** Some communication systems will not guarantee message delivery. If such a scenario occurs, it must be noticed. Extra communication buffer space is required for the data to be re-transmittal if the initial attempt was not successful.

The following sections describe various communication libraries. Although not directly comparative, they illustrate the trade-offs associated with the underlying principles that can be applied to the DSM communication layer.

5.1.1 Sockets

A socket is an abstraction of one end-point of a two-way communication link between two programs potentially running over a network. A socket is normally identified by a small integer which may be called the socket descriptor. The socket mechanism was first introduced in the 4.2 BSD Unix system in 1983 in conjunction with the TCP/IP protocols that first appeared in the 4.1 BSD Unix system in late 1981.
If communication is between two processes located on the same machine then UNIX sockets are one mechanism by which they can communicate. As depicted in Figure 5.1, where communication is required to take place between two processes on distributed machines, then Internet (INET) sockets will be provide a communication channel using a network protocol such as TCP (which in turn is provided by the hardware protocol). TCP provides a reliable service in contrast to the potentially unreliable UDP. However a performance penalty is incurred by TCP in order to guarantee reliable delivery as each packet must be acknowledged by the receiver. Even though there is an arrangement where such acknowledgements can be 'piggybacked' in the same transfer as user data, there are still latency overheads associated with the initial setup of the transfer when compared to the 'fire-and-forget' principle of UDP. If interconnect latency is high but of a lesser importance, such as for file transfer in a grid [107], then bandwidth can sometimes be used to compensate. With TCP, messages must be cached on the sender side until they are acknowledged by the receiver. An extra burden arises from the need to check for delivery.
The advantages of composing the messaging layer from sockets are:

- The flexibility of being able to specify the underlying transport protocol, increasing the potential for local optimisations.
- At this layer native support for multi-cast is provided, thus multiple instances of the same message (one for each destination) need not be generated.
- Platform independence. The Socket API is architecture neutral, and there are implementations for all major platforms.

The disadvantages of composing the messaging layer from sockets are:

- There is a significant amount of extra work involved in the implementation of a socket communication library, e.g. management of socket handles, buffers, and security.
- Connections have to be explicitly established, so implementing communication, this way, may not be as easy as with other messaging layers.
- Support is not provided for sharing multi-byte values across different architectures. Unless a facility for data typing and marshaling exists, or the user explicitly performs conversion, the system is limited to homogeneous nodes.

Socket programming is a primitive method for the construction of a messaging layer, but implementations exist for the majority of interconnects and platforms, and it is viewed as (and is) a building block of the higher level communication methods described below. For a Grid environment additional security implications arise, however there are socket implementations, such as Grid Security Infrastructure (GSI) sockets [108][1], that attempt to overcome these issues.

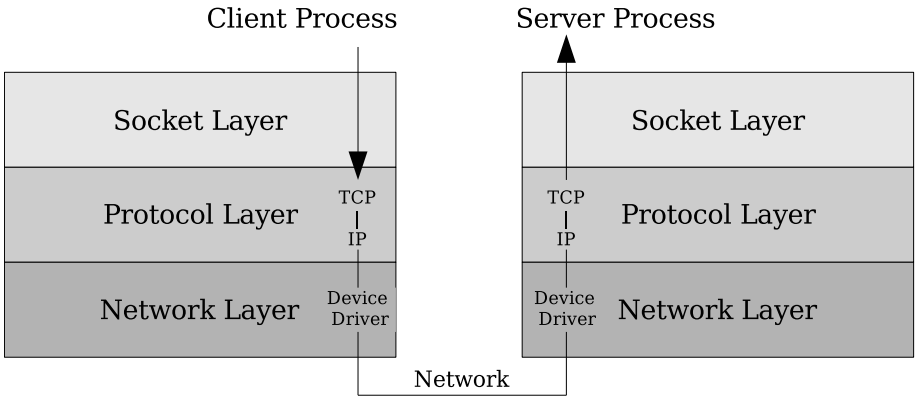


Figure 5.1: *Socket Layer Communication*

[1]The Grid Security Infrastructure (GSI) is based on IETF standard TLS (formerly SSL, the Secure Sockets Layer.

5.1.2 Message Passing

Message passing is a higher level communication paradigm allowing for two processes located on two distributed nodes to communicate at defined instances using simple naming semantics. Communication is usually two-sided, where the sender and receiver(s) must participate in a specified operation, such as the sending of a unit of data from a transmit buffer defined in one process to a receive buffer of another. The underlying transport protocol is often provided by a lower level communication mechanism such as sockets. The goal of a message passing implementation is to offer the developer a simpler communications API by the obfuscation of the difficulties imposed by a lower level communication library. Where previously a machine address and socket pair were required, message passing semantics usually only require a numerical process identifier. Problems arise if the message passing implementation is to be used between heterogeneous architectures and multi-byte primitives are transferred. Such a situation does not arise in clusters of homogeneous units, however it may arise in a Grid where the pool of available processors may be composed of different architectures. In such cases the user must strictly 'type' the data to be sent.
Advantages of composing the communication layer using message passing include:

- Generic functionality is provided that need not be reinvented for every application, for example the initialisation and finalisation routines provided by the message passing implementation.
- Increased usability over a lower level communication implementation such as sockets. Routines are provided for collective tasks such as global synchronisation barriers.
- There is a vibrant ecosystem surrounding some the most popular message passing implementations. This can include user support and a myriad of tools such for debugging, profiling, and visualisation.
- There are well defined standards and APIs, implemented for a myriad of platforms, architectures, and interconnects
- Applications utilising message passing libraries will experience an increase in performance due to current and ongoing work in the optimisation of these libraries.

The disadvantages of composing the communication layer using message passing include:

- The lack of support for transparent fault-tolerance. Often when a problem occurs on one node then the whole application fails. However, there are ongoing efforts to provide a solution to this problem [109, 36, 110, 111].
- Little support currently exists for multiple user threads calling message passing functions simultaneously.
- Dynamic process creation is not implemented (there are specifications in place).

Many different message-passing implementations exist. In the past high performance computing (HPC) hardware vendors had their own proprietary implementation, so an application would often have to be completely rewritten in order to be ported to a new platform. This led to a standardisation process within the HPC community to develop a cross platform message passing standard. The result of this effort was the Message Passing Interface (MPI) standard, which is explored below.
The Parallel Virtual Machine (PVM) library was a popular, yet non-standards-based alternative message passing implementation that preceded the MPI standards. The main advantages of PVM compared to earlier MPI implementations [112] were advanced techniques such as support for fault management and dynamic process creation. Later MPI standards rectified this and is now considered to be a better option, but PVM still remains in use among a minority of the academic community [113]. The current implementation of the PVM pseudo-standard is described in Appendix D.

The Message Passing Interface Standard: MPI

The Message Passing Interface (MPI) is the de-facto standard for message passing implementations. According to [114] some of its primary advantages over other message passing implementations include: MPI has more than one freely available, quality implementation; other proprietary implementations also exist; MPI defines an interface for a third party profiling mechanism, thus allowing for external tools to access profiling information; MPI has full asynchronous communication, thus allowing overlap of communication and computation; MPI efficiently manages message buffers; MPI synchronisation protects 3rd party software; MPI can efficiently program clusters, MPP, and Grids [48, 49]. Some of the current MPI implementations are given in Appendix D.
MPI is also formally specified [115], it has evolved through a number of standards. MPI-1 defined the basic message passing principles and function call definitions, such as send/receive, collective operations, and input-output routines. MPI-2 adds to previous standards with new features, most noticeably support for one sided communications, dynamic processes creation, and process management.
MPI is intended to be totally portable. Language bindings currently exist for C, C++, and FORTRAN. The MPI API, which is considerable, can be divided into logical sections: environmental(initialisation & finalisation), point-to-point (send/receive), and collective operations (barrier & broadcast). The six routines in Listing 5.1 are the most basic, and are required for any message passing application:

```
MPI_Init        // Initialisation of the MPI run-time system
MPI_Finalize    // Finalisation of the MPI run-time system
MPI_size        // Return the total number of processes
MPI_rank        // Return the identifier of this process
MPI_send        // Send a message to a specified process
MPI_recv        // Receive a message from another process
```

Listing 5.1: *Basic MPI routines*

5.1.3 Remote Procedure Call

Remote Procedure Call (RPC) is a methodology for constructing distributed applications. Unlike message passing, RPC provides a client/server model where the programming model is one-sided. It provides a means for a client process to invoke a function call on a remote (server) node by transparently sending a message encapsulating the function name and the required parameters to the remote node where the function is implemented. The function can either be blocking or non-blocking, in the latter case allowing for a situation where multiple calls may be executed in parallel. The result of the function invocation is subsequently transferred to the client machine.
RPC requires that a strict definition of the required functionality (in the form of a method interface, or stubs and skeletons in RPC terminology) must exist. The implementation of this functionality is provided by the 'server' process. Better heterogeneity support results, including transparent data typing and marshaling between processes participating in the system.
The advantages of composing the communication layer using RPC are:

- This one-sided programming model makes it easier to develop parallel applications, where the data and the associated function to operate on it are specified by the programmer. The principal benefit of this approach is that a simple user programming perspective is provided.
- Support exists for different architectures through the use of stubs and skeletons, allowing a developer to implement the mechanism by which data is correctly marshalled between processes.

Some of the disadvantages of composing the communication layer using RPC are:

- The increased programmer burden associated with the generation of defined interfaces. This is exacerbated by the the lack of a coherent or a formally defined standard, hence lack of portability in comparison to sockets or message passing.
- The RPC library can introduce significant latency (100s microseconds).
- The developer is responsible for implementing security on the server side
- Unless the process address space is kept consistent between the client and server by the programmer, global variables cannot be used, and pointers cannot be passed as function parameters or returned.

Like sockets, RPC can use either UDP or TCP as the underlying communication transport layer. When UDP is used then the implementation must deal with any reliability issues. Examples of RPC are Sun RPC, used in the implementation of the Network File System (NFS), Java RMI, and Corba (two of the more recent implementations of RPC style, distributed programming). The latter two relieve the programmer of the responsibility for marshaling/unmarshalling of data, while a RPC mechanism has been implemented for the Grid [116]. The main drawback with this technology is that a great deal of software infrastructure is required.

5.2 Information and monitoring systems

With the emergence of grid-based computing through the use of wide area networks it is becoming increasingly important that there are mechanisms in place to (a) obtain reliable information on the state of the grid environment [117], and (b) have the ability to monitor applications executing within it. This can involve access to dynamic information such as processor load, disk storage space, network availability [118], node availability, and (relatively) static data such as system configuration and topology. Clearly this information is important to grid operations staff, but also it may be the key to optimisation algorithms within services and applications.
A grid scheduler needs such information to correctly identify the best available resources to execute a grid job [117]. Subsequently the ability of a developer to access this information, such as the physical topology of the scheduled parallel application execution, can result in increased performance. For these reasons, grid information and monitoring architectures have been intensively examined by the Global Grid Forum (GGF) [119].
There are three primary components in the GGF grid monitoring architecture (GMA): producers, consumers, and directory services [119] [2]. Producers announce to the grid that they have information to publish; this is done by registering the type of data to be published with the directory. Consumers can query the directory for the location of producers that can meet their demand. Requests from consumers can then be directed towards the relevant producers. This interaction between the components is depicted in Figure 5.2. The two main actions that may be performed by the application are:

- **Producing information:** Applications may produce vast quantities of monitoring/logging information. There needs to be an efficient infrastructure for publishing system monitoring and user logging information from multi-process applications.
- **Consuming information:** If the DSM system wishes to generate a topology tree then environmental information must be present, and easily accessible. The monitoring information generated by the application also needs to be accessible for (i) the application as a feedback mechanism for the DSM system, and (ii) for run-time analysis of the application by the application developer. Also, the process of consuming information should be flexible.

Standard information schemas, such as the framework-independent Grid Laboratory Uniform Environment (GLUE) Schema[3], define a common conceptual data model to be used for monitoring and discovery of grid resources [120]. Information published compliant with the GLUE schema include physical machine status, storage element information, and network information. SMG defined schemas are required where the GLUE schema is deficient (does not contain the variables of interest), so the systems used will need to support this demand.

[2] Currently the GGF INFOD-WG working group are preparing the document 'Information Dissemination in the Grid Environment - Base Specifications'., which is intended as the template for next-generation grid information & monitoring systems, much as GMA did for the present generation
[3] Currently Version 1.2, with Version 2 currently under deliberations

Other implementations, such as MDS (part of the Globus toolkit [108]), are outlined in Appendix D. Both MDS and R-GMA adhere to the GLUE schema [120]. Either (or both) are a good basis for fulfilling the requirements of SMG. The following section illustrates the GMA-compliant approach, R-GMA, to an information and monitoring system.

5.2.1 R-GMA

The Relational Grid Monitoring Architecture (R-GMA) is a relational implementation of the GGF GMA [121]. As is the case with the GMA, information is published via producers and accessed by consumers as shown in Figure 5.2 below.
R-GMA is currently built using Java servlet technology (future implementations will be based on web services). Java, C, C++, Perl, and Python APIs exist for the use of R-GMA, allowing ease of creation of highly portable applications. R-GMA has successfully been used to enable the monitoring of MPI applications in a grid environment [122]. R-GMA adheres to the GLUE schema, and applications have been developed to republish data held in other systems (MDS).
The interaction of all its components are illustrated in Figure 9.1 (page 133). When a producer is instantiated it registers itself with the directory service, termed a registry in R-GMA, specifying the view/type of information to be published by declaring a *predicate*. This defines the schema which specifies the structure of the relational table that the producer will be publishing to. The schema definitions are maintained by a schema servlet (an extension of the GMA architecture). The producer is accessed via one of the defined *producer* APIs. This can be either a primary or secondary producer, depending on the type of data being published. The producer will in turn publish via a producer servlet. The registry is responsible for maintaining information about all producers in the system, such as the location and the type of data being published, i.e. the predicate. A consumer will request access to the data via a consumer servlet, which will contact the registry to obtain the locations of data providers (producers). The registry mediates between the producers (if present), and the consumer is then directed towards the relevant producer. Information regarding the consumer request will also be stored in the registry, enabling optimisation where multiple consumers issue identical requests.

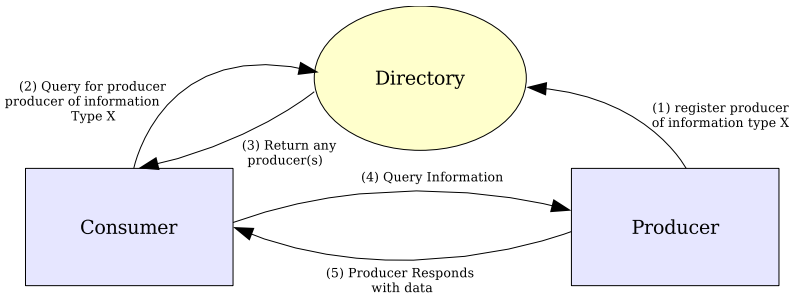


Figure 5.2: *Grid Monitoring Architecture (GMA)*

5.3 The User API: Case Studies

A DSM developer needs to take cognisance of an often quoted reason, aside from poor performance, why DSM failed to gain acceptance among the parallel programming community, i.e. programability. All DSMs implemented their own non-standard Application Programming Interface (API), an API might be simple and intuitive, such as that presented by Treadmarks, but it is still (another) non-standard. Educating a group of application developers for a new API could prove costly for something that could fall out of fashion. It should be noted that another recent effort exists, UPC [27], that define a shared memory programming model that makes provides for architectures with different levels of memory consistency, but as it has yet to gain acceptance it is not covered here. Another, Global Arrays(GA [123]), offers functionality for accessing shared arrays, assume a NUMA-like platform, but has been constructed on top of a message-passing library. It has similar semantics to the put/get operations in MPI-2 outlined below.
The philosophy that will govern the API design of the SMG DSM will ultimately be to support a standard either directly or indirectly. In this section we examine some of the standards that provide some direction. Each programming model imposes a certain level of burden. This is highlighted in the examination of the OpenMP programming API, the MPI-2 one-sided communication routines, and also an implementation of OpenMP for distributed memory machines. An API such as *pthreads* is not considered due to a lack of a defined memory model.

5.3.1 OpenMP

The (OpenMP) standard [4] has as its main focus the parallelisation of structured loops in application code. It employs a fork-join model of parallel execution which is particularly suited to applications involving large iterative operations on array-based structures. This is achieved through the annotation of parallel code areas through the use of compiler directives. In the C version of the specification (C++ and FORTRAN also exist) an OpenMP directive has the following form:

#pragma omp `directive` [*clause,...*] *newline*

An important motivation for using OpenMP is its ability to simultaneously support both serial and parallel variants of programs through the use of these *#pragma omp* compiler directives (that define when parallelisation is required), and so can be turned on/off at compile time, but can also be introduced throughout the appliaction in an incremental fashion. Directives can be parallel, work-sharing, or synchronisation in nature and are discussed in the following sections. Also explored are the APIs and memory consistency required by OpenMP. A more substantive definition of the API is given in Appendix D. Profiling libraries may be implemented using the profiling interface definition, `POMP`, and other work has been done on combining the data obtained using this interface for static analysis to obtain speedups [124], and in a dynamic context for run-time optimisation [125, 126].

Work-Sharing Directives

The `parallel` directive essentially directs the initial user thread to create a team of threads. The parallel directive can also be paired with a `for` directive, that simultaneously creates the team of threads and apportions each a portion of the associated loop workload (See Listing 5.2 below: each thread in the team, of size N, will get allocated approximately X/N iterations of the *for* workload). Allocation strategies may be instigated by specifying a `SCHEDULING` clause (Appendix D). The `sections` directive complements the `for` directive, allowing parallelism to be functionally decomposed, and thereby enabling a number of further discrete `section` constructs to be divided among the threads and executed concurrently.
In the C programming language the *for* loop is parallelised, by dividing the work among the team of threads of execution according to some function (specified by the clause). In the default case each thread is assigned a static chunk (determined by the number of iterations to be performed and the number of threads in the team). Barriers are used implicitly at the start and end of the snippet, where any thread will wait at the end of the structured code block until all threads have arrived, except, for example, where a *nowait* clause has been declared. In order for concurrency to be allowed inside this parallel section, the shared memory regions must be concurrently writable by multiple writers, i.e. a multiple writer protocol must be supported.

```
#pragma omp parallel for        /* Begin parallel section */
for (k = 0; k < X; k++){
      sub_total += a[k];
}                               /* End parallel section */

#pragma omp parallel sections /* Begin parallel section */
 {
    #pragma omp section         /* Methods foo & bar
    {  foo();  }                   executed concurrently by
                                   different threads    */
    #pragma omp section
    {  bar();  }
}                               /* End parallel section */
```

Listing 5.2: *OpenMP Work Sharing Directives*

API calls & Environment variables

The OpenMP specification also includes API calls to enable the programmer to query and set the value of OpenMP environment variables, to use lock synchronisation routines, and timer routines. The most notable of the environmental routines enable the dynamic setting of the default number of threads in a OpenMP team (this value can be specified by a clause to the parallel directive).

Mutual Exclusion Directives

Mutual Exclusion is supported through the use of synchronisation directives (highlighted in Listing 5.3), and additionally though primitives with associated API routines (see next section). The barrier and flush operations, referred to above, have explicit directives. The other mutual exclusion directives that ensure structured access to shared data are as follows.

```
/* Only the master thread will execute code */
#pragma omp master
{...}

/* Only one thread will execute the code */
#pragma omp single
{...}

/* All threads will execute the
             code, but only one at a time */
#pragma omp critical
{...}

/* Atomic update to a shared variable */
#pragma omp atomic
<statement>
```

Listing 5.3: *Format of OMP Mutual Exclusion directives*

These directives must be nested within a `parallel` (or variant) directive. The first two are functionally equivalent, and allow only one thread to execute a block of code. In the `omp` case the master thread will always execute the block, while in general with `single` the first thread to reach the block will execute it. The `critical` directive allows the code block to be executed by all the threads in the team, but ensures that only one will do so at a time. The `atomic` directive is a restricted version of `critical`, allowing a single statement, such as a variable increment e.g. i++, to be executed atomically.

OpenMP Memory consistency model

In the early OpenMP standards no reference was made to the memory consistency model that was needed in order that an OpenMP application would be correct. Version 2.5 of the OpenMP standard somewhat rectified this by specifying that the memory consistency model required is a relaxed consistency model, similar to weak ordering as described in [127]. Various data scoping attribute clauses can also be supplied (see Appendix D, page 230). All shared memory references must be performed with respect to OpenMP `flush` directives. While an explicit `flush` exists they are also implicit

with regard to the work-sharing directives. The following requirements need to be met with respect to flush operations [128]:

- If the intersection of the flush-sets of two flushes performed by two different threads is non-empty then the two flushes must be completed as if in some sequential order, seen by all threads.
- if the intersection of the two flushes performed by one thread is non-empty, then the two flushes must appear to be completed in that thread's program order.
- If the intersection of the flush-sets of two flushes is empty, the threads can observe these flushes in any order.

In relation to this thesis, the specification of a relaxed consistency model was very encouraging, as it strengthened the case for SMG to be designed as a potential target for an OpenMP compiler.

5.3.2 Cluster OpenMP

Cluster OpenMP is a recent optional module of the Intel compiler family (C, C++ & FORTRAN) [129] that allows applications written using the OpenMP interface to execute transparently across distributed memory machines. Although previous efforts strove to provide such functionality, this is the first to be provided commercially, and is fully supported in terms of commercial-grade documentation and support tools (compiler, thread checker, thread profiler). This distributed OpenMP is built upon a modified version of the Treadmarks DSM (see Section 4.8.4), rectifying many of the inherent deficits listed, i.e. larger number of user processes, multiple threads per process, larger quantities of sharable data, and increased support for modern interconnects.
In addition to additional functionality, some deviations from the OpenMP standard have been made. The most noticeable departure is from the memory model, whereby by default, all variables are not not shared among all threads of execution. Shared variables are explicitly declared shared using a new `sharable` directive which is an Intel extension to OpenMP standard. Some compiler support exists (the -clomp-sharable-propagation compiler directive) for identifying variables that need to be declared using this directive [130]. The minimum granularity at which memory consistency is guaranteed is four bytes.
As the `mmap` system call is used internally by Cluster OpenMP, use of this function by user code should be treated with caution; alternative functions are provided for use. OpenMP lock variables `omp_lock_t` must be also explicitly allocated, and dynamic memory must be allocated/freed using API calls instead of using the standard library functions (`malloc/free`). Nested parallelism, i.e. a parallel directive within the scope of another, is not supported.
Like Treadmarks, remote process start-up is done using the basic remote shell (`rsh`), or the secure variant `ssh`. Communication is still via sockets, but the DSM has increased user thread-ability where the number of DSM system threads (termed bottom-half threads) is proportional to the number of user application threads.

5.3.3 MPI-2 Shared Memory

The remote memory access functions included in version 2 of the MPI interface enable one sided communication, while at the same time not requiring a uniform shared address space. This remote memory access, although somewhat removing the communicating pair requirement, is still explicit in nature. MPI-2 functions `MPI_Get`, `MPI_Put`, and `MPI_Accumulate` allow respectively for the initiation of one-sided remote memory read and writing. However, before these can be used, constructions known as 'memory windows' need to be established, using the `MPI_Win_create`, that specify a contiguous address range (memory address and size) where those memory operations are mapped into the local address space.
The above one-sided operations are non-blocking, so the point at which these remote memory actions initiate and ultimately become visible are denoted using synchronisation operations specified using the active (i.e. a collective operation, so all processes are involved) 'Fence' mechanism `MPI_Win_fence`, which is analogous to a barrier. If multiple processes perform a put to the same window location the the result is undefined. The passive routines (i.e. where only one caller is involved), `MPI_Win_lock` and `MPI_Win_unlock`, are available to provide multiple-reader/single-writer functionality, but have yet to be implemented in open-source MPI-2 implementations.
The data being transferred is well defined using the MPI type routines, so when completely implemented, this functionality will allow for data access in heterogeneous systems. The main drawbacks with the MPI-2 RMA operations are the requirement for the user to explicitly specify the read and write routine and to synchronise access. Other work has concluded that MPI-2 does not provide an adequate compilation target for global address space languages [131] or parallelising compilers [85]. Additionally there is a lack of support for fault tolerance, since the error handlers only allow for the cleanup of the process and not adaption to the loss of a process. Where some implementations include such support they do not totally adhere to the MPI standards.

```
MPI_Win_create    // Create a memory access window
MPI_Win_free      // Free a window
MPI_Put           // Write operation on remote memory
MPI_Get           // Read operation on remote memory
MPI_Accumulate    // Perform operation while performing put
MPI_Win_fence     // Collective, barrier-like operation
MPI_Win_lock      // Lock (r/w) the memory access window
MPI_Win_unlock    // Unlock (read/write) the window
MPI_Win_start     // Start an interval to the memory window
MPI_Win_complete  // Complete a memory access interval
MPI_Win_post      // Start an interval locally
MPI_Win_wait      // Signal completion of interval locally
```

Listing 5.4: *MPI one-sided communication*

The code example given in Listing 5.5 demonstrates the one-sided communication abilities as provided by the MPI-2 standard.

```
MPI_Alloc_mem(sizeof(int)*size, MPI_INFO_NULL, &a);
MPI_Alloc_mem(sizeof(int)*size, MPI_INFO_NULL, &b);
MPI_Win_create(a, size, sizeof(int), MPI_INFO_NULL,
                    MPI_COMM_WORLD, &win);
for (i = 0; i < size; i++){
  a[i] = rank * 100 + i;
}

printf("Process %d has the following:", rank);
for (i = 0; i < size; i++){
  printf(" %d", a[i]);
}
printf("\n");

MPI_Win_fence((MPI_MODE_NOPUT | MPI_MODE_NOPRECEDE), win);

if(op == GET){
  for (i = 0; i < size; i++){
    MPI_Get(&b[i], 1, MPI_INT, i, rank, 1, MPI_INT, win);
  }
}else{
  for (i = 0; i < size; i++){
    MPI_Put(&a[i], 1, MPI_INT, i, rank, 1, MPI_INT, win);
  }
}

MPI_Win_fence(MPI_MODE_NOSUCCEED, win);

printf("Process %d obtained the following:", rank);
for (i = 0; i < size; i++){
  printf(" %d", b[i]);
}
printf("\n");

MPI_Win_free(&win);
MPI_Free_mem(a);
MPI_Free_mem(b);
```

Listing 5.5: *Use of MPI one-sided communication*

CHAPTER 6

Shared Memory for Grids (SMG)

The explorations of this thesis were conducted by implementing a DSM called SMG (Shared Memory for Grids). One of the primary goals of this thesis is the exploration of facilities to allow existing parallel applications to execute efficiently on a grid with little, if any modifications required. To achieve this proper attention need to be paid to the requirements of existing parallel programming standards. As OpenMP is the current *de facto* standard for parallel application development on shared memory architectures it is appropriate to ultimately design the DSM to be a target of an OpenMP source-to-source compiler. Some of the OpenMP parallel constructs, covered briefly in Section 5.3.1), will be further examined in order to ascertain such requirements. This will form the starting point for the implementation of DSM topics that were discussed in Chapter 4.

This chapter presents an overview of the internals of the base SMG system such as how the DSM interacts with user application threads, how communication between processes is effectively managed, and the start-up and shutdown stages of a SMG DSM application. This permits both application-level performance optimisation as well as algorithm implementation and problem tracing, and is crucial to facilitating higher performance on a Grid. Chapters 7 and 8 will deal with the relevant aspects of integrating memory management and synchronisation. SMG allows applications to be monitored either by the application itself, by another process, or by the user. Chapter 9 deals with the implementation of the libraries to access information and monitoring systems.

The steps involved in the compilation and execution of a simple Helloworld SMG application are also covered. The chapter concludes by describing some of the system implementation issues that were encountered.

6.1 DSM Requirements

The system must present the programmer with an easy-to-use and intuitive Application Programming Interface (API) in order that the additional burden in the construction of a DSM application is minimal. This requires that the semantics must be as close to that of normal shared memory programming as possible. The SMG project aims to borrow from the successes and learn from the mistakes of previous DSM implementations.
When designing a DSM system, the main decision is to what extent the user will be responsible for maintaining the shared memory consistent. In general, less burden on the programmer results in more work to be performed at the DSM management layer. Ultimately this may result in performance deterioration that will limit the overall scalability of the system. This is further exacerbated when there is little or no hardware support available to the DSM. There are successful implementations of hardware-support using off the shelf components for distributed shared memory, employing interconnects such as SCI [20] and Myrinet [21], but one of the aims of this research was to avoid the need for specialised hardware support.
Apart from a being a popular research topic, DSM has otherwise been a failure, a number of reasons have been suggested [92], but one significant factor is the poor take-up outside research labs due to the lack of any open standards in the area. Any new DSM implementation is usually accompanied by a new programmers API (some are given in Appendix C), and involves a developer learning a new set of programming semantics, with an additional drawback in the lack of portability of the application. Therefore parallel application developers are loathe to adopt any new non-standardised API.
For a DSM to gain acceptance, compliance with an open standard is necessary to encourage use, so it is with this in mind that the DSM must be designed to be potentially a target of a parallelising compiler, such as OpenMP, thereby allowing the use of existing parallel code with support for future application development. In Section 5.3.1 some OpenMP directives were examined. One can identify some of the design requirements of the DSM if it is to form the target of a parallelising compiler.
To implement these directives, a compiler targeting a DSM system only requires a consistency model that ensures shared memory areas are consistent after a synchronisation operation has occurred. In the parallelised *for* example, shared memory sections are required to be consistent at the entrance and exit of the section. This allows any of the more relaxed consistency models to be used, as there is a close affinity with synchronisation primitives and the fact that shared memory can be explicitly declared as such using the OpenMP **shared** clause.
Where a developer targets the DSM, its primary functions are then used to act as proxy between SMG processes and provide the developer with a transparent method of accessing shared memory and performing synchronisation routines. For a Grid DSM the DSM engine at local level should not have to overly contend with the characteristics of remote nodes, e.g. different architectures or platforms that may have different page-sizes, except in cases where this is unavoidable, e.g. to transfer of data between nodes caused by invalid data.

Useful statistics should be gathered (if required) while the DSM system is running and used to detect variable use & code areas that result in poor performance, the most pertinent being page faults, memory allocations and releases, and DSM specific statistics. The user should have access to this information if desired.

6.2 SMG DSM architecture

The overall function of the DSM engine is to provide and manage transparent access by the application developer to shared memory areas. This involves keeping track of the location and state of shared memory areas and synchronisation primitives. The SMG DSM has additional duties such as enabling topology support by processing information about the execution environment to generate a topology tree (in order to create tree-structured barriers). This facilitates taking advantage of topology in order to provide efficient messaging, and utilising the monitoring system to enable a feedback mechanism to allow run-time optimisation of the DSM.
The various components are represented in Figure 6.1. There are clearly defined APIs for most of the relevant decisions in the DSM implementation (these will be discussed in the following chapters).

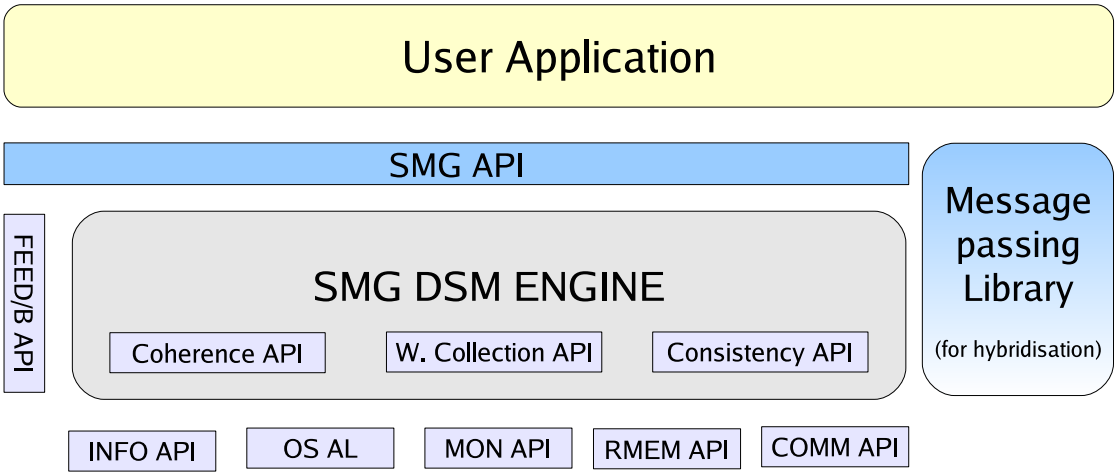


Figure 6.1: *SMG Conceptual Architecture*

The following is a summary of the decisions made in the area of the system architecture:

- Initially, MPI is used for the communication between DSM system threads. Hence the initial communication (comms) API implements a wrapper around the MPI communications infrastructure, but can also be used to support hybrid programming.
- When a user application thread requests a resource, the DSM system will be responsible for satisfying it. This choice, while adding extra latency, is justified from

the desire to efficiently support multi-threaded user applications by ensuring that duplicate requests that involve external communication can be eliminated.

- Initialisation and finalisation routines are required. This deviates from OpenMP standards where no such provision exists, but could be remedied by a compiler.

- In order to support user multi-threaded applications a wrapper around the thread creation routine of the thread library is provided.

All API function calls belong to one of the three following groups.

1. **DSM Management** (Initialisation/finalization/Environment) will be dealt with in the remainder of this chapter.

2. **Shared Memory Management** such as allocation in Chapter 7, and support for hybridisation in Chapter 9).

3. **Synchronisation** operations are covered in Chapter 8

6.3 Internal DSM Engine Operation

The operation of the DSM system requires that it be initialised and finalised in a structured manner as there are initial tasks required to be performed such as the initialisation of internal data structures and synchronisation primitives, and the starting of the DSM engine thread(s). All DSM threads in participating processes must be started and coordinated before DSM services can be provided to user application threads.
When the system has been started the user application threads make requests of the DSM system through the defined API, or through events generated by the memory management system. Support for user multi-threaded programs are required so any such requests must be passed to the DSM system thread through an internal queue structure as depicted in Figure 6.2. When a user thread places a request on the queue it currently blocks pending a response from the DSM system. A possible future refinement here is to add support for non-blocking calls, analogous to the fashion in which MPI provides complimentary non-blocking and blocking calls.
The DSM communication sub-system must concurrently probe for requests from remote nodes. When a request is received (it may be coherence, synchronisation, or system related) it is added to the DSM queue to be processed by a DSM system thread. A system thread will also process the request and depending on the request type it can be handled if it can be satisfied, forwarded to another process, or in certain circumstances ignored. While a solution such as this, introduces additional latency to the DSM, it is necessary for supporting both user and DSM system multi-threading.

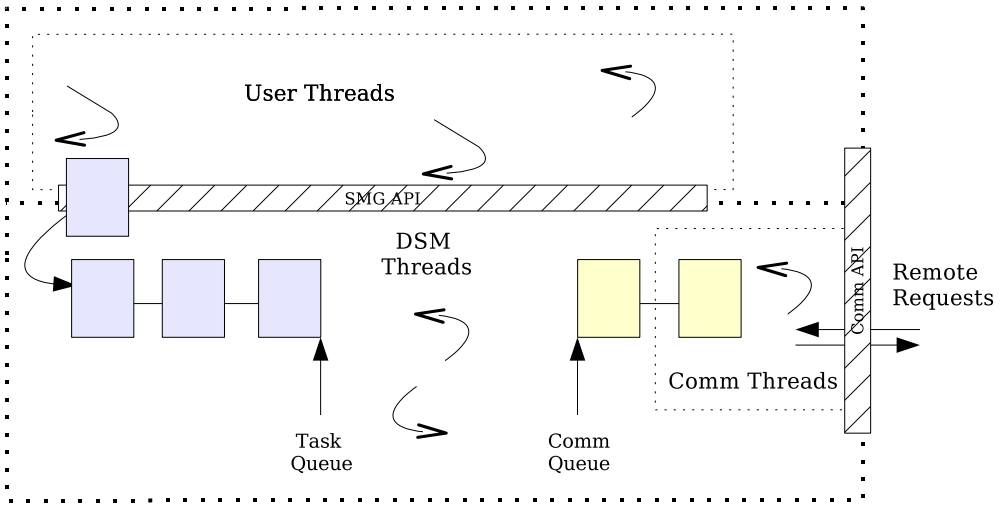


Figure 6.2: *DSM Engine Structure*

6.4 Helloworld using the SMG API

In the distributed shared memory programming model, the level at which the user is responsible for consistency dictates the semantics and style of the user API (currently only a C interface is provided). The API consists of a number of functions that allow the user to exploit the facilities provided by the system, such as locks and barriers. One of the most successful implementations of software-only DSM is Treadmarks, its API is simple and elegant (listed in Appendix C, page 221). The design of the SMG API follows a similar style to this API, and is given in full in Appendix E (page 243).
In the Treadmarks model a thread is responsible for ensuring that it has correct access privileges to a variable through the use of synchronisation primitives. This principle applies even if a thread only requires read access. The SMG DSM introduces the same constraints; in order for a thread to ensure that it is using a consistent shared memory then the appropriate synchronising operation must be performed. A developer will construct an application specifying the control of a thread using flow control statements with reference to their identifiers.
All function calls in the SMG API return error codes indicating the return status of the function call. Typical values returned are **SMG_SUCCESS** or **SMG_FAILURE**. These return codes should be examined when returned to ensure that an application is correctly executing or that the action that was requested has taken place. Input and output parameters are specified as the arguments to the function.

6.4.1 DSM Initialise

The local actions essentially consist of the initialisation of the internal storage, DSM handles and queues. This SMG routine must be invoked before any other API call. The

API initialisation function is given below. The first two arguments, *argc* and *argv*, are the arguments passed to the user application at run-time, which are in turn passed to the MPI initialisation routine. The *flags* argument specifies the environmental requirements such as information & monitoring services. The last argument *type_defaults* defines what the requirements of the application developer are, and to specify what the default memory consistency models and coherence protocols are; this allows internal DSM engine optimisations.

```
int SMG_init(int *argc, char ***argv, int flags,
                    int type_defaults);
```

At the start-up sequence of the SMG DSM engine a number of tasks are performed:

- initialisation of the local internal DSM structures
- the start-up of the DSM system handler thread(s)
- establish the underlying communication channels between all processes. In the MPI implementation, this acts as a wrapper around the underlying initialisation call (i.e. *MPI_Init*/*MPI_thread_Init* in the single and multi threaded versions respectively).
- the installation of the shared memory write trapping mechanisms, currently this is a system page-fault handler, which is described in detail in Section 7.3.1.
- If the information and/or monitoring system is required, then the application must register with it during the initialisation routine. If registration fails because the information and monitoring system is not available then the system will exit.
- The initialisation routine also acts as a global barrier that ensures that all threads of execution, wherever they be, perform such tasks before DSM requests can be invoked by remote processes.

The error code returned will indicate successful initialisation or failure. Sources of failure can include the unavailability of requested information & monitoring services, and failure by the underlying communication system. Upon successful completion the value *SMG_SUCCESS* is returned. At this point the global variables, such as *SMG_proc_rank* and *SMG_proc_size*, are valid, and, if specified, the application has been registered with the information system.

6.4.2 DSM System Environment

Although the global variables that specify the process pool size and a process's rank are provided (*SMG_proc_rank* and *SMG_proc_size* respectively) for use by the developer after the initialisation call has returned, supplementary functions are provided that will dynamically return these values when required. The SMG prototypes for these functions are:

```
int SMG_process_size();
int SMG_process_rank();
```

The first function will return the number of processes in the system, which will be less than or equal to the total number of user threads. Currently, this value is fixed at run-time due to the present lack of support for dynamic processes in MPI. The latter function will return the numerical rank of the calling process, in the range [0..N-1], where N is the total number of processes in the system.
A function to display internal DSM engine information (SMG_print_state) is provided. Additional functions are provided for the getting and setting of system attributes, e.g. these functions provided user access to the topology information if originally enabled. An API call (*SMG_module_load*) is also provided for the loading of additional modules that allow for the extension of the DSM, e.g. in areas such as shared memory management.

```
void SMG_print_state(int stream);
int  SMG_internal_get(int key, void *value);
int  SMG_internal_set(int key, int value);
int  SMG_module_load(int MODULE_TYPE, char *location);
```

6.4.3 DSM Finalise

In order for the system to exit cleanly the finalisation function call must be invoked. This ensures that all processes synchronise on exit, and guarantees that only when all processes are ready to finish they will actually do so, thereby preventing shared memory regions from being freed while they might still be required at a remote node. When all processes have reached this point all remaining shared memory regions are freed. Once all local cleanup routines have been called the information & monitoring systems can be signalled. Finally the underlying communication environment can be finalised. This function may act as a wrapper around the relevant communication function; with the MPI communications implementation, MPI_Finalize called by the DSM system.
The SMG API finalisation call, called at the end of all SMG applications is:

```
int SMG_finalise()
```

This function will block until it has completed successfully. Once this is done no further DSM services can be availed of. This function should only be called once, usually by the master user thread, otherwise the behaviour is unspecified.

6.4.4 DSM Abort

In certain circumstances it is desirable for an application to abort during execution for some reason local to one of the processes. An abort call is provided to allow the application as a whole to degrade gracefully, and perform housekeeping functions such as

the closure of files, deregistering from the information system, and the reporting of errors to the developer. This call may form a wrapper around the underlying communication call (*MPI_Abort*), so once it has been called no further interprocess communication is possible. This call will terminate all processes involved in the DSM application. In all cases the user specifies the error code, *error_code*, to return to the invoking environment. The SMG API call to abort an application is:

```
int SMG_Abort(int error_code)
```

6.4.5 User Multi-threading

Enabling multi-threaded user applications can result in significant performance gains [132]. Support will leverage hardware advances, such as multi-core processors, to better employ multiple user threads per node, allowing the DSM to fully exploit the available resources (say extreme exploitation of overlapping of computation and communication).
When creating user threads it is important that they are registered with the DSM runtime management system. This allows for setting of a thread signal mask in order to catch accesses to shared memory regions. Thread creation is requested using the *SMG_thread_create* API call, essentially a wrapper around the underlying thread library call. Only threads created using the SMG API call will be visible to the DSM system. If the API call is bypassed when the user thread is created and if a shared region is accessed an unintended SEGV fault will occur. More of the latent effects related to this decision are mentioned in Section 8.1.
At thread initialisation the appropriate cleanup handler functions are registered using the *pthread_cleanup_push* call, and so system routines that are required to be called before the user thread exits are registered, thereby avoiding the developer having to do so manually in the application. An option that is allowed is to verify that the exiting thread does not hold any locks; if there is a lock held, then the appropriate measures may be taken. Currently, the event to logged to the monitoring system.
As the SMG API function for the creation of user threads is a wrapper around the the underlying *pthread* library call, it presents the same familiar argument list to the developer. The function parameter *tid* is a thread identifier for the newly created thread; *attr* are the required thread attributes; *start_routine* is the entrance function for the newly created thread; and *arg* is a reference to any input parameters that the user may wish to pass to the thread. The error codes returned by the function are the same as those returned by the underlying thread creation call.

```
int SMG_thread_create(pthread_t *tid, pthread_attr_t *attr,
                      void *start_routine, void *arg);
int SMG_thread_count();
int SMG_thread_count_atrank(int process_id);
int SMG_thread_systemwide();
```

The number of threads that are active in a given process can be obtained using the *SMG_thread_count_atrank* call. The number of current 'alive' user application threads within the local process, created using the *SMG_thread_create* function, can be obtained using the *SMG_thread_count* function[1]. The total number of system threads that are active across all processes at a point in time can be obtained using the *SMG_thread_systemwide* function. Changes in the number of user threads in the system only becomes visible upon a system-wide barrier, so between initialisation and the first global barrier this function will return a value equal to *SMG_proc_size*.

6.5 Engine communication

Nearly all current software DSM systems either only include support for standard UNIX sockets (see Section 5.1.1), or are targeted towards a specific network technology [], and are thus not suitable to porting to a grid environment. The SMG DSM system requires a means to transfer messages between the system threads. Naturally since MPI is being widely integrated into grid infrastructures, then MPI communications could be used. However, so as to not tie the implementation of SMG to a particular communication ideology the decision was taken that the external messaging functionality will be accessed via a well defined API. This decision allows for the use of different messaging transport implementations once an implementation of the SMG communications (comm) library has been provided. A subset of the API is given below in Listing 6.1.
A standard message envelope (header) was defined, comprising fields such as *type*, message size, timestamp, *origin*. All DSM messages are filtered upon the *type* field by the incoming message handler. The *message_size* field will often duplicate what is present in the meta-info of the underlying communication system, but allows multiple individual messages to be multiplexed into a single transaction. The *timestamp* field is present to enable discarding of duplicate messages, while the *origin field* is present to allow messages, under certain circumstances, to be redirected to other processes. If the request cannot be fulfilled by a process, eventually a process that can fulfill the request will respond directly to the requesting process. An incoming message is placed on the *incoming* list (specified at initialisation), and the DSM is notified using the *smg_notify* function.
Unlike other components of the DSM, such as consistency & coherency modules, currently only one active communication implementation is allowed from start-up. While priority queueing schemes for DSM messaging have been shown to result in increased performance [133], no such scheme has been implemented at this time, primarily due to the need to support both a single and a multi-threaded DSM.

[1] (equivalent to `SMG_thread_count_atrank`(*SMG_proc_rank*))

```
int SMG_comm_init(int *argc, char **argv[],int flags,
        Handle_list *incoming, Handle_list *free_handles,
        dsmMessageCallBack smg_notify,
        dsmMessageBufferMalloc SMGMallocFunct,
        dsmMessageBufferFree SMGbufferFreeFunc);
int SMG_comm_initialised();
int SMG_comm_send(void *buffer, int size, int destination,
        int FREE_BUFF, int *status);
int SMG_comm_send_many(void *buffer, int size, int *dests,
        int num_destinations, int FREE_BUFF, int *status);
int SMG_comm_bcast(int origin, void *data, int data_size);
int SMG_comm_finalise();
```

Listing 6.1: *SMG communication interface*

6.5.1 MPI Communication Implementation

The initial SMG communication library uses MPI. It was chosen mainly due to its ubiquitous nature (there are multiple implementations available, proprietary and open-source) and increasing support in a grid environment. The model assumes that a separate MPI_communicator is created for the exclusive use of the DSM communication sub-system. This ensures that conflicting calls cannot be issued by the DSM and a user application thread.
The implementation of the *SMG_comm* interface specified above greatly depends on the MPI implementation. The most salient question is whether multi-thread support is required. In the early versions of the MPI standard there was no specification for multi-threading support. With the evolution of CPU architectures, it became apparent that this would need to be rectified. With MPI implementations supporting the version 2 standard, a user application has the `MPI_Init_thread` function available to ascertain the level of thread support (various levels are described in Section D). It must be noted that currently multi-threaded MPI is not pervasive across Grid sites, and therefore two MPI flavours of the *comm* API have been developed (supporting single-threaded and multi-threaded implementations).

Single-threaded MPI

If only a single user thread is permitted to call the library then use of blocking calls is forbidden. The side effects include that the scope for multi-threading within the DSM engine is greatly reduced, and that hybrid DSM/MPI programming is not permissible. Incoming and outgoing DSM requests are polled for by the DSM communication thread, which is the only thread permitted to access the MPI library. Incoming requests are checked using the `MPI_Probe` call, while outgoing requests must be dispatched via the internal communication queue as depicted in Figure 6.2. Apart from the disadvantages

just mentioned, this single threaded communication channel usurps system CPU resources as requests must be continually polled for. A communication timeout may be specified to alleviate this, but at the expense of increased message latencies.

Multi-threaded MPI

If multi-threaded MPI support is available, indicated by the value `MPI_THREAD_MULTIPLE` being returned by the call to `MPI_Init_thread`, then a far superior DSM experience is obtainable. The DSM engine & communication systems can be highly multi-threaded, and additionally user application code can use the underlying message-passing infrastructure. The DSM system thread can send messages directly without using the communication thread, decreasing the latency for sending a DSM request. Incoming messages are handled by a dedicated thread(s) that blocks on the `MPI_Probe` call. There are additional concerns, primarily with management and shutdown of the system (i.e. the message receiving thread is blocked; to prevent this, extra latency is introduced for communication shutdown).

6.6 SMG Compilation

In order to build a SMG application it is necessary that at least one implementation, or such number as required by the user, of the core modules (Information, Monitoring, and Communication) must exist for a successful build. The required modules are depicted in Figure 10.2, page 156. The default module used for a particular class will be the one that is specified at link time. Unlike extension modules no other core module implementations can be loaded dynamically.
The code sample below in Listing 6.2 shows a simple hello-world program that demonstrates the use of the initialisation and finalisation routines. This code snippet also demonstrates how to turn on the use of the information and monitoring systems by setting of the appropriate flags in the initialisation routine. Although the consistency level is specified as *ENTRY_CONSISTENCY*, as it happens that this is the only consistency level currently supported. The information and monitoring system, if required (specified in the *flags* field of SMG_Init), will be the one linked to the user application at compile time. This is discussed in more detail in Chapter 9.

6.6.1 SMG extension Modules

In the next chapter it will be seen that the various aspects of the shared memory management run-time system (consistency, coherence, write collection) can be augmented using different extension modules (created by implementing defined interfaces). These modules can be loaded dynamically at run-time (using the *SMG_module_load* function specified in Section 6.4.2), so no recompilation of user code is required to utilise a new module. The module should be available as a dynamic library to the system.

```
int main (int argc, char *argv[]){
    int error, flags, default;

    flags    = (INFORMATION_FLAG | MONITORING_FLAG);
    defaults = ENTRY_CONSISTENCY;
    error    = SMG_init(&argc, &argv, flags, defaults);
    if(error != SMG_SUCCESS)
        return -1;

    printf(''Helloworld I am process # %d of %d\n'',
                SMG_rank, SMG_size);
    error = SMG_finalise();

    return error;
}
```

Listing 6.2: *helloworld using SMG*

6.7 Run-time execution

A SMG application must be executed within an environment support by the underlying communication implementation. An application developed with the SMG DSM and compiled using the MPI implementation of the communication API will present itself as a regular MPI job, thus easily runnable on any system (standard cluster or grid with a suitable MPI implementation installed). This is achieved by using the appropriate MPI start-up mechanism (mpirun/mpiexec), together with its associated options and parameters, i.e. shown below the stanadard `-np` is used to specify the number of processes to start. However, a simple script `smgexec` which is generated during the compilation process. Any arguments specified will be passed to the underlying execution mechanism. The code in Listing 6.2 is executed from the command line in the following manner:

```
> mpiexec -np 2 -machinefile machines helloworld_smg
I am process #0 of 2
I am process #1 of 2
```

Once the system initialisation routine has been called the global variables *SMG_proc_size* and *SMG_proc_rank* are initialised by the DSM run-time. The former gives the number of processes participating in the DSM system pool (the initialisation routine does not register the number of user threads in use, which is currently fixed at initialisation). The latter variable gives the unique identifier of the process participating in the ensemble. These values will be determined by the number of processes a user requests, which can be specified as an argument passed to the MPI execution environment, and the order in

which the processes will be started.
To execute the application in a grid setting the same procedure that would be used to execute a similar program is followed. For a SMG application with an underlying message passing execution setting the procedure will be the same as MPI. In an EGEE Grid, with glite middleware, a job description language (jdl) file is required. A `jdl` file is shown below for a SMG job.
A supplementary script to execute the application (basically calling smgexec as above) is required. Listing 6.3 gives a simple jdl file for a MPI type job. The required files are the SMG program executable itself and the execution script.

```
1  Type          = "Job";
   JobType       = "MPICH";
3  NodeNumber    = 4;
   Executable    = "helloworld_smg.sh";
5  Arguments     = "helloworld_smg -i $EDG_WL_JOBID";
   StdOutput     = "helloworld_smg.out";
7  StdError      = "helloworld_smg.err";
   InputSandbox  = {"helloworld_smg.sh","helloworld_smg"};
9  OutputSandbox = {"mpiexec.out", "helloworld_smg.out",
                                        "laplace_smg.err"};
```

Listing 6.3: *Simple SMG helloworld jdl file*

6.8 Other Issues

Important issues that arose were multi-threading in Linux, provision of multi-threading in MPI implementations, DSM system messaging, and how information and monitoring hooks into the overall execution.

- *Development Cost:* what is the overall cost that the grid introduces to the DSM itself? Intrinsic factors have already been mentioned in Chapter 4. Ideally the application developer will be oblivious to the fact that the application is executing on a grid, but exposing the information API to the developer may have benefits, both to the developer and to the application at run-time.

- *Ease of Use:* as the DSM is currently built on top of MPI, the ease of use of SMG will be determined by the ease of use the MPI execution environment. For example as the submission of MPI jobs to the grid becomes more trivial, so too will that of a SMG job.

- *Start-up Cost:* the time for SMG to start and initialise will be (relatively) long, but should be small relative to the total job execution length. Start-up of information & monitoring will introduce additional overheads, however it is hoped that this will be compensated for by performance gains.

- *Scalability of Performance:* the scalability of any parallel application is dependent on many factors, hardware, middleware, application algorithm, etc. The scalability of the SMG DSM is likely to be somewhat comparable with the MPI implementations that it is constructed upon. Additional demands such as DSM management scalability will reduce scalability relative to MPI.

- *Fault Tolerance:* as the grid is non-deterministic, fault tolerance becomes a thorny issue. Fault-tolerant implementations of MPI are currently under development [36, 110], so it follows that a fault-tolerant SMG could be a likely byproduct. Some work has been done on making DSMs fault tolerant [64, 134, 135, 136]. The current approach taken with SMG is to enable the DSM to recover by simply check-pointing the shared memory objects. The most suitable mechanism for a checkpoint (and only one supported in SMG) is a global barrier as this ensures that a consistent shared memory state is check-pointed. Upon restart the user application can re-read the checkpoint files and resume computation. For this strategy to be effective the user application code must also be able to recover, but with respect to the DSM. Currently no support is provided for the user to achieve this.

CHAPTER 7

SMG Shared Memory

The conventional style for shared memory programming uses processes to implement parallelism; locks and barriers are used to synchronise the processes. A software-only DSM consists of a number of processes that can interact via a low level messaging system. One design issue is how shared memory is managed overall in the distributed system. Additional issues include when consistency is actually enforced, how coherence is implemented, the efficient location of a consistent shared memory region and the efficient replication/caching of shared data.

Most of the earlier DSM implementations supported sequential consistency, but it impacts negatively on the performance of distributed systems because of the strict requirements on system wide access to shared memory [79] with support only for memory transactions that are not required to be atomic. Implementation of more efficient relaxed consistency models require a close binding between the shared memory system and the management of synchronisation primitives.

The development of applications using the DSM is required to be as similar to traditional shared memory programming as possible. Shared memory allocation should be trivial, resulting in transparent access by all processes to the shared memory region. This task is made difficult for SMG as use of specialised compilers and non standard libraries is not permitted. Nonetheless, the application developer should be oblivious to how memory is actually allocated.

This chapter identifies the approaches used for implementing the management of global shared memory areas. This includes topics such as: the global creation of shared memory; the normal use of such in SMG applications, with descriptions how memory is viewed across a set of distributed processes; the releasing (freeing) of shared data; and the techniques employed to minimise the volume of data transferred in order to keep memory copies consistent.

Other issues that are explored are the support to be provided for multiple user threads of execution per process, and the benefits that can be gained for the DSM from integrating information & monitoring services with the DSM. The chapter concludes with a review of the design choices and highlights some issues that arose from the implementation.

7.1 SMG Memory Management

Shared memory areas are referenced in application code using a globally unique identifier, which serves as an opaque reference to a local DSM handle (described in Listing 7.1). This handle is used by the developer to refer to a specific shared memory section throughout the execution of a SMG application. A shared memory area is allocated/freed using the appropriate functions detailed in the subsequent sections. When these are used the memory must be somehow managed locally. Internal data structures are responsible for keeping records of where the shared memory section is mapped into the local process' address space, storing information about the shared region such as its state, current location, maintenance information, and useful statistical information. The structure below depicts a handle that holds the associated information for a shared memory area. When an allocation function is called, an entry is added to the local collection of shared memory handles. There are two separate collections, one that holds handles that the local process is responsible for managing, and another for which remote processes are responsible. The handle is removed when the free function is called on a shared memory region.

```
struct handle_t {
  int identifier;
  int type_info;
  int timestamp;
  int owner;
  int size;
  int bound_to;
  void *bound_to_ptr;
  void *data_ptr;
  void *twin_ptr;
  int *coherence_info;
  stat_t *statistics};
```

Listing 7.1: *SMG Handle*

As alluded to in previous chapters a DSM algorithm that allows multiple writers to concurrently access shared locations is required. The management of shared data regions takes a similar approach to the dynamic distributed manager algorithm as first proposed by Li [65]. At any given time, all shared memory objects have both a manager and an owner, although a single process may perform both roles. The manager of a shared object is fixed and determined by some hashing function e.g. *manager = (process_rank % memory_identifier)*. The owner of the object is dynamic and it is this process that has the most authoritative copy of the data. As mentioned previously in Section 4.3.1, the manager of a shared memory section is responsible for keeping track of the current owner of the shared variable. This enables faster look-up of the current owner by a requesting process.

7.1.1 DSM Engine Memory allocation

A shared memory region must be allocated locally by all processes wishing to access it. Shared memory areas are referenced by an application-unique numeric identifier. The shared memory region is mapped into the local address space by the DSM, and once this has completed it can be utilised. All unique shared memory areas must begin on a virtual page boundary.
A shared memory object is explicitly allocated using the call below. Entry Consistency (EC) is assumed, and as such use of shared memory is closely associated with the use of synchronisation primitives. Currently only shared regions bound to lock primitives may be rebound to another primitive during the execution of an application, and only to another lock.

```
int SMG_shmem_malloc(int id, int size, void **pointer,
                     int type, int lock_bind_to});
```

The parameters to the allocation function under EC are: *id* the unique identifier of the shared object; *size* the size (in bytes) of the region to be allocated; *pointer* address of the start of the object; *type* type information associated with the object; *sync_bind_to* the synchronisation primitive to bind to.
When a process allocates a shared memory region, it in effect maps the global virtual shared object into its own address space. When two processes allocate the same shared object it might not be allocated at the same location in both virtual memory address spaces; this occurrence is depicted in Figure 7.1. To help alleviate the issues that may arise for the developer in such situations where references to shared locations are passed between processes, function calls are provided that allow for the generation of global pointers from local memory pointers and vice-versa. These are discussed in more detail in Section 7.1.3.
The following is an outline of the sequence that occurs when a shared memory allocation is requested. If the synchronisation primitive to be bound to is a lock then it must be owned by the calling process.

- A handle is allocated for the object. The handle depicted in the previous section includes management information.
- If the item is to be bound to a synchronisation primitive such as a lock then the process that owns the lock issues the notification to the manager of the object and subsequently binds to the shared area.
- A call to allocate the local shared memory takes place. The *mmap* call is used as it allows the placement of the object to begin on a virtual memory page boundary. This is important for the reasons discussed in Section 7.3.
- If the allocation is successful the protection levels on the shared object is set so that access to the shared region will be trapped.

- Meta-information such as associated coherence dirty flags and timestamps can be allocated and initialised.

The allocated shared memory object only becomes globally visible after the next synchronisation point. If the amount of local memory is insufficient to satisfy an allocation request then some housekeeping tasks are preformed (e.g. the DSM system will expire any variables where it does not have access to the associated synchronisation variable, or twin regions may be released if not in use).
It is intended that the DSM operate in a grid composed of a heterogeneous mix of architectures with potentially different representations of data. This introduces the biggest obstacle to developing grid applications as the user must be mindful of this problem in data exchange. In a SMG application the user is not responsible for data exchange but the DSM system is; this requires that the DSM know the exact format of data that is been transferred. The problems in implementing such a facility are described at the end of this chapter.

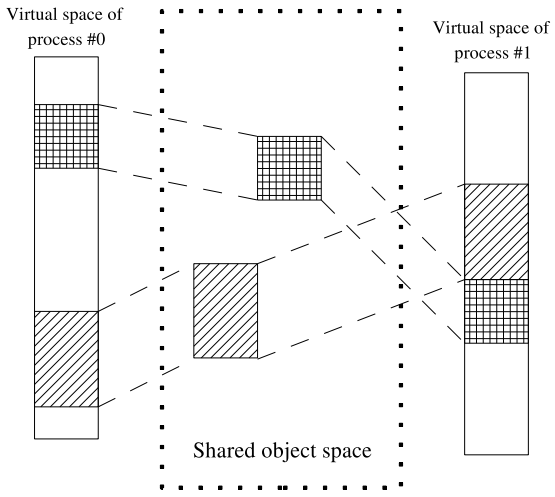


Figure 7.1: *Shared Memory Mapping*

7.1.2 Global memory mmap

To map an existing piece of shared memory into the system one can use the available function for mapping in data. The main proviso of using this function is that the memory has been allocated using the `mmap` function, ensuring that the page is aligned upon a page boundary, thus another shared memory region cannot be allocated on the remainder of a virtual page.

```
int SMG_shmem_mmap(int id, int size, void *pointer,
                   int type, int lock_bind_to});
```

7.1.3 Global memory pointers

As a shared memory area may be mapped into the address spaces of processes at different locations a facility is required that allows for passing of a reference to a shared area between processes. SMG provides functions to accomplish this. These functions are also implemented with a view to supporting heterogeneous environments, where the conversion work will be provided by the DSM, as offsets within shared regions may be different on different platforms.
The first function generates a global reference when provided with a pointer to a shared area mapped into the local address space. The second function provides the reverse operation converting a global reference to a pointer to a local memory address.

```
smg_ptr SMG_local_toglobal(void *local_ptr)
void *SMG_global_tolocal(smg_ptr *global)
```

7.1.4 DSM Engine Memory free

The process of freeing a shared memory region results in the freeing of resources used in the maintenance of the region. This can produce substantial benefits as the local memory resources used by the DSM can be more than twice the size of the actual shared memory area. This is mainly due to the memory occupied by the twin space for a shared object. Like the memory allocation call(s) the freeing of shared memory regions only becomes globally valid following a synchronisation operation. The API call to free the memory region locally is:

```
int SMG_shmem_free(int id)
```

When a process calls this function to free the node the action that results depends on whether the calling process is the owner of the region/bound synchronisation object:

- If the calling process is not the owner or manager then all data is expunged and local resources are released. The handle of the object will be marked for releasing, which is done at the next synchronisation point, and it will have its timestamp is set to zero.
- If the calling process is the owner of the region or binding synchronisation primitive, the manager of the region is first notified of the pending action. All resources can then be released including the object handle. The actual shared region will be released using the appropriate system call, currently *munmap*. At the next synchronisation operation the appropriate notification can be distributed to other processes in the system.

By ensuring that the above protocols are followed it is guaranteed that no process holding the valid version of the shared object region can deallocate it locally while any other process might be (or might begin to be) using it before the next synchronisation point.

7.2 SMG consistency

As shown in the discussion on DSM consistency models (Section 4.5) it is evident that only relaxed consistency models are suitable for software-only DSM implementations. The best choices for a DSM that may potentially be used in a grid environment are the LAZY-Release and Entry consistency. These have been implemented and have demonstrated modest potential in the Treadmarks [96] and Midway [81] DSMs, while the Brazos DSM has been used as the target of an OpenMP compiler [102].
In order that SMG can be easily extended it was important to ensure that clear interfaces are defined for new consistency models and coherency protocols to be implemented. The defined consistency interface (specified in Listing 7.2). All shared variables must declare the consistency model required at time of allocation. This allows for the shared memory region to be registered with the consistency protocol. With a relaxed consistency model there is an explicit association with synchronisation operations. With this in mind, mappings for acquire (*consis_sync_acquire*) and release (*consis_sync_release*) operation are present that map as prescribed in Section 2.4 onto synchronisation operations (more detail in Chapter 8).

```
typedef struct _consistency_blk{
  int identifier; char code[8];
  char description[consistency_desc_size];
  int (*consis_init)();
  int (*consis_finalise)();
  int (*consis_mem_attributes)(int parameter, Handle *obj);
  int (*consis_mem_attributes_free)(Handle *obj);
  int (*consis_sync_acquire)(Handle *barrier_ptr);
  int (*consis_sync_release)(Handle *barrier_ptr);
  void (*consis_remote_response)(int tag, int src, void *msg);
  void (*consis_remote_request)(int tag, int src, void *msg);
  void (*consis_write_trap)(Handle *handle, char *fault);
  void (*consis_coherence_alert)(Handle *obj);
  int (*consis_register_object)(int type,int id,Handle *obj);
  void *handle;
  void *stats;
}consistency_block;
```

Listing 7.2: *Consistency model interface*

It has been shown in studies that entry and lazy-release consistencies perform comparably [137]. However, the same studies show that lazy consistency can generate an order of magnitude greater number of messages than entry consistency. This is the primary reason why EC is the superior consistency model for a grid DSM. Multi-writer Entry Consistency will thus be required in SMG; this is achievable when a shared region's binding object is a barrier.

7.2.1 SMG Entry Consistency

The previously mentioned drawback of increased programmer burden with EC is countered by the fact that this DSM will be primarily used as the target for a parallelising compiler such as OpenMP, where the user will be oblivious to the need to binding shared memory regions to synchronisation primitives. However as previous attempts have shown some deviation from the standard OpenMP is required[1]. Additionally, for synchronisation access entry consistency requires thread consistency while release consistency requires processor consistency [81]. This allows multiple threads of execution to be placed in a single node.
In EC, an explicit link is required between the shared region and a synchronisation variable[2]. This link is achieved at allocation by invoking the *consis_mem_attributes* function of the consistency API (in SMG an object can be bound to all barrier synchronisation primitives).
Upon a write trap event (i.e. a pagefault), the *consis_segv_handler* handler is invoked to notify the bound synchronisation primitive that a bound object has been modified. On the first write to a shared variable under EC, the *consis_coherence_alert* function is called to record that coherence for the object will be required at a release operation (this improves the performance for the release action). On a release event, coherence actions are taken for all shared objects bound to the synchronisation primitive with a coherence alert in place.

7.2.2 SMG granularity

One of the most important decisions in designing a DSM system is the choice of what is the minimum unit of address space that may be shared. Factors in this decision have been mentioned in previous chapters such as the available inter-node bandwidth and latency. The main priority in a DSM that may ultimately be used in a Grid setting is to reduce the number of messages generated; doing so involves the selection of a relaxed consistency model [75].
Other factors involved include the underlying hardware and user requirements expectations. If the grid is a heterogeneous environment, then there may still be a problem as not all platforms share the same page size (for write trapping), so techniques used by Treadmarks where the size of the coherence unit is a system page are not applicable in the case of a grid DSM. As EC has been deemed an appropriate choice for the primary consistency model, the granularity of sharing is not a factor that could be influenced in any great way, as the EC sharing granularity is whatever size the user allocates a shared object to be. In the case of the DSM being used as an OpenMP target then this gives rise to an additional benefit as the granularity implied by the **shared** clause is dependent on the size of the shared memory item, which can be varied.

[1] this long standing presumption [138] was reaffirmed when Intel released Cluster OpenMP in May 2006, deviating from the standard in the process

[2] The terminology $x \epsilon y$, will be used to denote the association for EC objects with synchronisation primitives, i.e. an object x, bound to the use of the synchronisation primitive z.

7.3 Write trapping

The use of relaxed consistency models results in a substantial reduction in the number of messages generated between processes. Nonetheless, the best DSM implementation of an application will never have a lower message count than a comparable message passing implementation, mainly due to the associated overhead of DSM control messages. There is scope, however, for reduction in the payload of individual messages by only sending the shared memory sections that get modified. The process of detecting the areas that get written to is called *write trapping*. It is vitally important that efficient methods be used to detect the locations at which write operations to shared memory occur.
Previous DSM implementations used compiler instrumentation to set a dirty bit per unit of the shared object that was modified [42][3], but for SMG no specialised compiler is permitted. The SMG implementation of write trapping therefore uses the virtual memory management system to detect writes at a system page granularity. This is achieved by making use of the virtual memory protection mechanisms provided by the Linux Operating System (other flavours of Unix provide the same base functionality using a similar methodology)[4]. Techniques that evaluate the dynamic sharing state on a per page basis have been explored in depth [139].
As previously mentioned shared memory objects in the SMG lie on separate virtual memory pages, the size of which is not a primary consideration, but does determine the granularity at which write trapping occurs, and hence impacts on write collection (see Section 7.5). For small shared regions the above write trapping technique proves to be inefficient. Anything less than a page would require an alternative detection method, i.e. a specialised compiler, or a modified kernel segv trap handler that would increment the program counter (PC).

7.3.1 Detecting Writes Using SEGV Faults

The mechanism is based on the software write detection methods described in [140].
The underlying virtual memory management facilities provided by the operating system allow for the identification of an access to shared memory (writes being of most interest) through the setting of virtual page protection flags. The subsequent generation of a page fault results in the execution of the DSM page fault handler. When this occurs certain actions can be performed that allow for the accessed area to be identified at a later point, allowing for the write collection (record of modifications) to be generated to represent the changes to the shared object within the given time duration.
The shared memory page fault handler is registered with the operating system at initialisation through the use of the *sigaction* system function call. This function allows the association of a user defined fault handler with a specific signal (in this case *SEGV*). When a page fault occurs the appropriate signal is generated as indicated by a SEGV value in the *si_signo* field in the *siginfo_t* structure (outlined in Appendix D) that is passed to the declared signal handler. The memory address at which the fault occurs is

[3] this approach is borrowed from Hardware-DSM
[4] one of the goals was that no OS-specific operation was permitted

given in the *si_addr* field.
This faulting location is resolved to the local record of shared memory regions by searching the local data structures that hold the shared memory handles. If the memory address is found then the appropriate memory coherence action may be performed that enables write collection at a later stage. If a record is not found then it is assumed that a local memory access violation has occurred and the appropriate actions can occur; usually the application will exit.
The actions taken by the SMG SEGV (pagefault) implementation of the write trapping handler are depicted in Figure 7.2. Currently there are three variants for multiple page objects that can be employed by SMG in response to the SEGV fault; each one is described below. For all three methods on the first fault access to the object a per-object dirty bit is set. The first two incorporate twinning, which involves making a copy of the shared location before the first write can complete. The first and third strategies will incur concurrent faults for every other page modified, with each one setting a per-page dirty bit.
The decision of which strategy to use can be made on a per shared memory object basis, and may be adapted to the run-time characteristics of the application. The cost equations for each are given for a scenario where an application has a shared object consisting of N pages, and modifies M of them, $(N \geq M)$ in a given interval. The definitions of the component costs are given in Table 7.1.

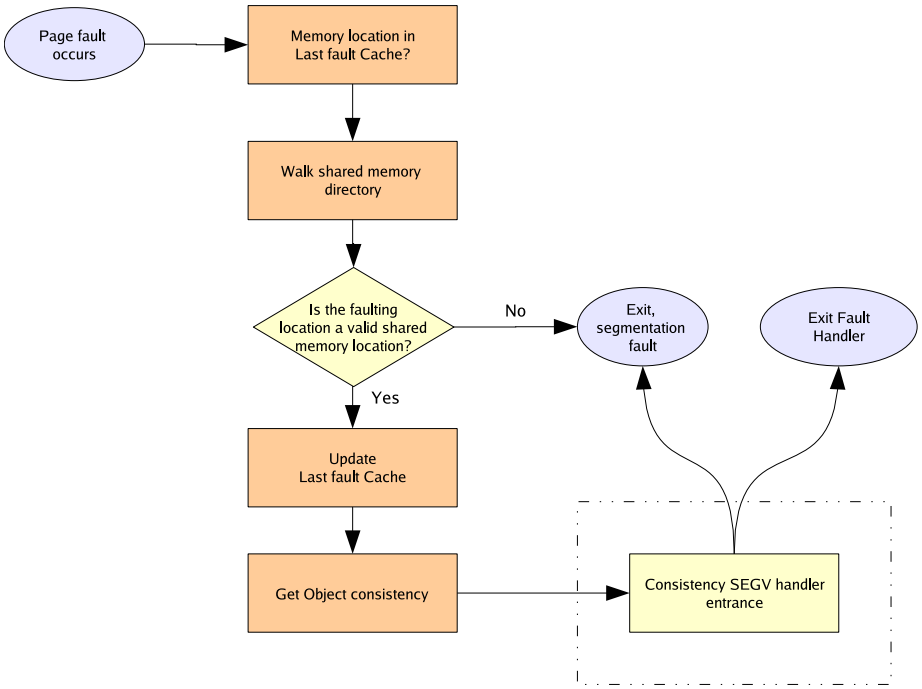


Figure 7.2: *SMG SEGV Handler State Diagram*

Twin-per-page

With this strategy, only the page to which the write occurs gets twinned, as depicted in Figure 7.3(b). As the page-protection settings remain unchanged for the rest of the object, when a write occurs to a location on another page, then that will be twinned too, see Figure 7.3(c). A record of this event is made in the list of pages, termed dirty page list, that has been modified, and a flag for the object as a whole will be set upon the first modification to any page in the object. With this approach it is assumed that the writes issued by the process are very localised, thus memory copying and the amount of memory to generate the write collection against will both be minimised. If writes are not localised there will be more overhead incurred with more page faults being generated. The cost for this strategy is given by Equation 7.1.

$$T_{overhead,tp} = N \times (T_{cp} + T_{wc} + T_{pf} + T_{pp}) \tag{7.1}$$

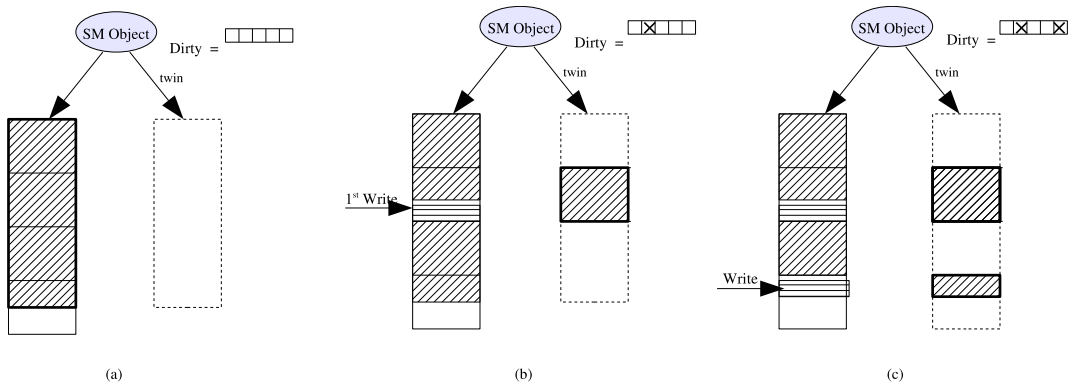


Figure 7.3: *Twin page on write*

Twin-all-on-write

In this scenario the whole object is twinned upon the first page fault. The page protections for all constituent pages of the shared memory object are write-enabled as shown in Figure 7.4 (b). This has the effect of eliminating subsequent page-faults for the duration of the write interval. The downside of this approach is that at the end of the interval the whole object must be examined for write collection. The benefits will accrue where the writes issued by the process to the shared region are non-continuous throughout the object. The overhead for this strategy can be calculated as:

$$T_{overhead,ta} = N \times (T_{cp} + T_{wc}) + T_{pf} + T_{pp} \tag{7.2}$$

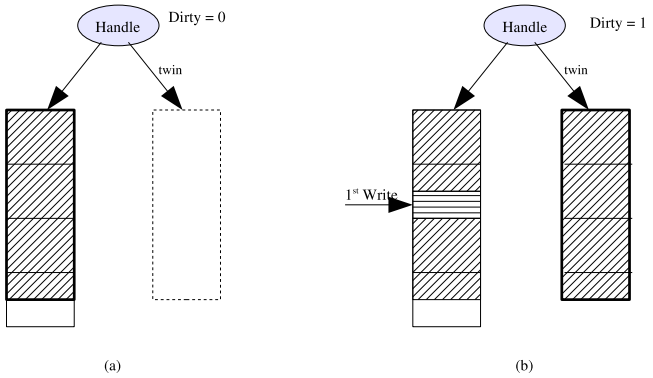


Figure 7.4: *Twin all on write*

Twin None

The above write trapping variations usurp precious memory resources to make a copy of the shared memory page, all in order to support multiple writer shared memory access modes. With this scheme no twin is created; this means that a multi-writer protocol is not supported. The overhead (see Equation 7.3) is then clearly lighter (i.e. there is no T_{cp}), however, this results in the overhead being offloaded to the write collection mechanism. Whether this is acceptable depends on the behaviour of the application. This strategy could be further developed in a similar manner to *twin-all*, where just one 'shared object' dirty bit is set by the only fault for the given interval. Upon release the whole shared region is transferred in the coherence mechanism.

$$T_{overhead,tn} = N \times (T_{wc} + T_{pf} + T_{pp}) \tag{7.3}$$

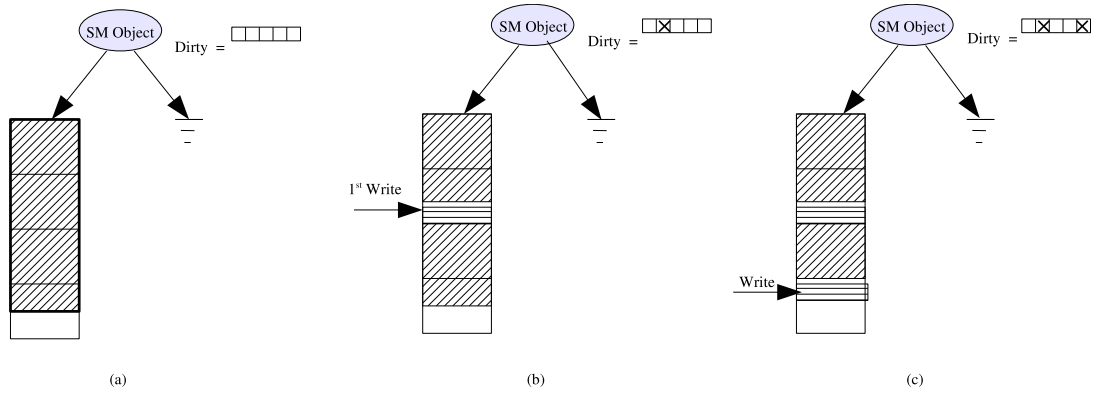


Figure 7.5: *Twin nothing on write*

T_{cp}	Time to make a copy of a page
T_{wc}	Time to generate a write collection across a page
T_{pf}	Time for a page fault
T_{pp}	Time so set page protection for a page used by object.

Table 7.1: *Metrics involved for SEGV Write Trapping Schemes.*

7.4 SMG Coherency

From its inception, it was considered highly desirable that the DSM should enable efficient execution in a multi-site Grid environment. As mentioned in Section 4.6, the main influence (from the DSM engine's viewpoint) is the coherence protocol. As outlined in Section 4.6 there are two broad classes of coherence protocol: invalidate and update. [141] demonstrated the benefits of having support for multiple protocols within a DSM. The *home-based* class of protocols that were discussed in Section 4.6.1 are not considered as their use are patently unsuitable for a Grid DSM.
Weak consistency models are only of interest in this project, this implies that consistency is enforced at appropriate synchronisation acquire and release operations. How SMG supports coherence protocols in relation to this, and how this actually works is discussed in Chapter 8. The default consistency model implemented in SMG, EC, somewhat forces the choice regarding the coherency protocol. As shared memory objects have an associated synchronisation primitive, update coherency has previously been implicitly used (because accessing a sync primitive implies the use of the bound regions, thus signalling the required modifications, so it would be sensible that these are implicitly prefetched; data that may not be required is sent regardless).

```
typedef struct _proto_blk{
  int  identifier; char code[8];
  char description[coherence_desc_size];
  int (*coherence_init)();
  int (*coherence_finalise)();
  int (*coherence_mem_attrs_create)(Handle *obj);
  int (*coherence_mem_attrs_free)(Handle *obj);
  int (*coherence_predict)(Handle *obj, Handle *sync_prim);
  int (*coherence_alarm)(Handle *obj);
  void (*coherence_write_trap)(Handle *obj_handle,
                    char *fault_at);
  update_part *(*coherence_action)(Handle *obj,
                    update_part *barrier_update);
}coherence_block;
```

Listing 7.3: *Coherence protocol interface*

In a similar manner to consistency, a general API for coherence is defined (enabling new protocols to be easily added!), a subset of which is given in Listing 7.3. The more interesting functions, such as that which is called in response to the a write trap event, *coherence_write_trap*, are expanded upon in the implementation descriptions below.
In SMG, an update protocol was first implemented, as it is the most natural protocol for the chosen consistency model, EC. An invalidate scheme was initially rejected due to reasons outlined in Section 4.6. However, it became apparent that an update protocol alone was insufficient to cope with all sharing patterns, so the *dynamic-subscription* protocol was developed to address the deficits, see Section 7.4.2.

7.4.1 Update-based Coherency Protocol

Upon a release event the write collection process will start, and under the *update* protocol the modifications to shared memory regions that (i) occurred during the preceding interval[5], and (ii) dictated by the consistency protocol, will be sent to the DSM engine of appropriate remote processes.
Update coherence requires that modifications be detected; the actual methods were discussed in Section 7.3. Figure 7.6 expands further on the SEGV method depicted in Figure 7.2 to illustrate how the *EC* consistency model and *Update* coherence protocol are integrated. The update protocol can use the output of either of the write collection mechanisms outlined in Section 7.5 without any further processing.
In SMG, every shared memory object has an associated logical clock that is incremented at every release if a modification has occurred. When a process wants access to a consistent shared location it must perform the synchronisation action that is required of it (the particulars are discussed in chapter 8). The circumstances in which this arises are different for barriers and locks.
When a process issues a lock synchronisation request it includes its logical time-stamp for the object in the message to the probable owner. When the true owner of the lock responds to the request it will compare the requester's version of the logical time-stamp with its own (valid) version. If both are the same then no coherence action is required. If they are different then the correct coherence information can be generated. (Section 7.3.1 describes how this is done). With EC and lock synchronisation, if superfluous data is being transferred then the granularity of the shared region should be reviewed (i.e. one should break the shared object into smaller objects).
For a barrier all processes advance the logical clock of an object if there are modifications. Because of this all processes must receive all modifications by all other processes. It is conventional practice to piggyback these modifications on the barrier release/acquire messages. During the release phase all modifications are gathered by one process, the coordinator, and checked for potential concurrent writes to the same location. These can be all sent during the acquire phase. This situation is depicted in Figure 8.3 page 126, where an object a is bound to a barrier synchronisation primitive z ($a \epsilon z$), and the modifications from all writers to a are dispersed during the barrier process.

[5]The interval is delineated to be the time period between the previous release and the next acquire.

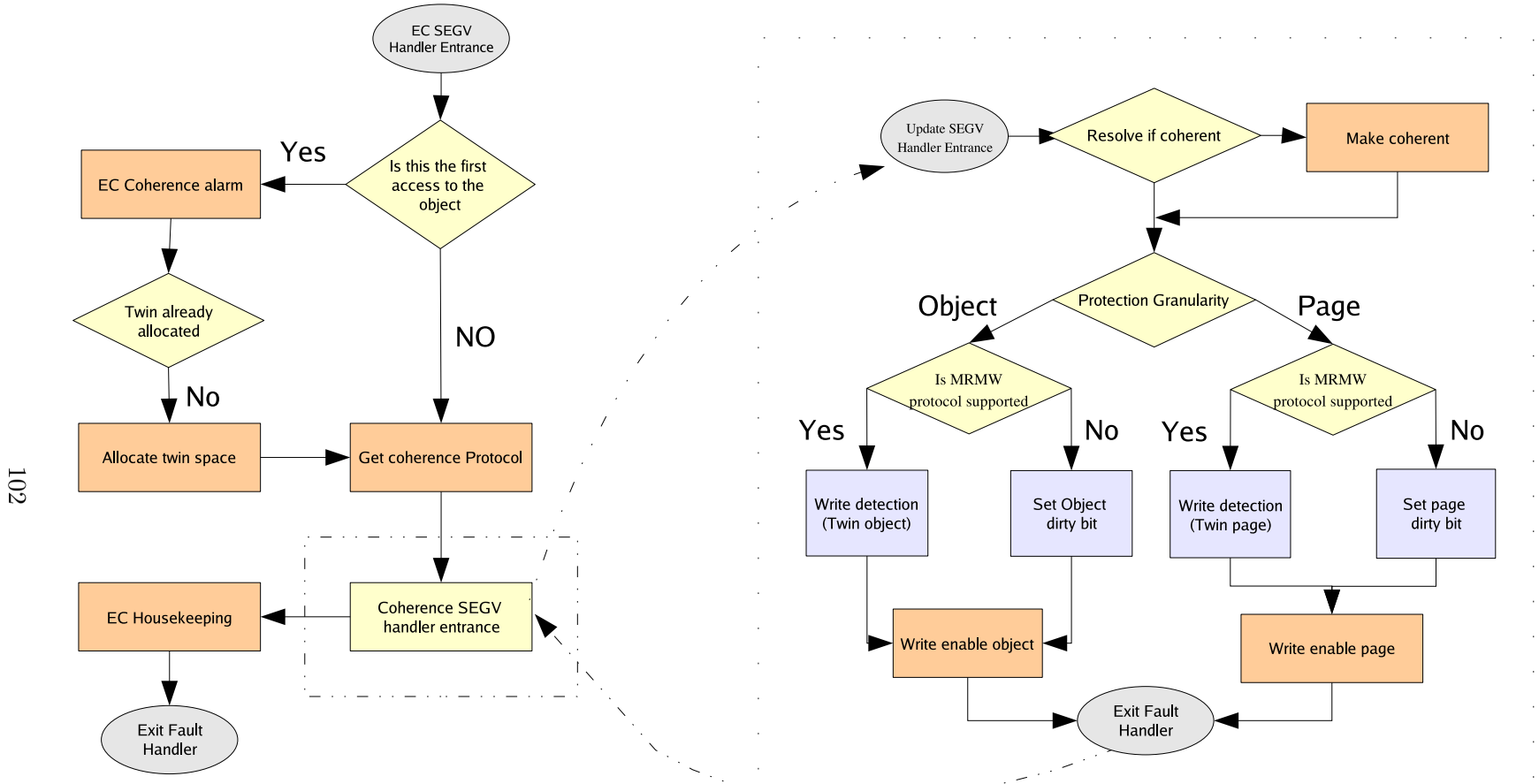


Figure 7.6: *Integration of EC Consistency & Update protocol*

7.4.2 Subscription based Coherency Protocol

The main motivation for developing a new protocol was to address the inherent defects of the invalidate and update protocols:

i. an invalidate protocol will generate a significant number of small messages proportional to the degree of sharing of the object, where a latent effect can be the significant delay in processing while the process waits for its request for write collections (a complete valid copy) for an invalidated object; but this can result in a different type of communication efficiency as only requested data is transferred.

ii. an update protocol is the converse: minimal coherence messages that are piggybacked upon a synchronisation message, minimal stall in the application occurs as required data is implicitly prefetched; but if the object exhibits a fractured view of sharing, then there is considerable potential for superfluous data to be transferred.

The *Dynamic Subscription* (DySub) protocol aims to find the middle ground between them in terms of the balancing act between volume of messages and coherence message payload. All sections of a shared memory object that are accessed by a particular process will have update characteristics attributed to them, i.e. the process subscribes to regions; other locations not accessed, or less likely to, will have the attributes of the invalidate scheme, i.e. the process will not receive updates for those regions. This protocol is dynamic in nature as locations can switch between the two, allowing transient accesses to be supported. If a region is not subscribed to, then upon an access to this invalid region a fault occurs and a valid copy of the region must be obtained, which can be from any process that does already subscribe.
The essence of the DySub protocol is that for a process responding to a request for write collection, the parent decides what is forwarded to the requester by maintaining a subscription of the anticipated requested data. There are downsides of this approach, it involves extra 'filtering' overhead as each coherence message must be examined for non-subscription coherence data. For shared memory with transient access patterns (described as migratory under Section 4.1) the DySub protocol begins to exhibit the characteristics of update protocols.

SMG Subscription Protocol Implementation

Similar concepts have been explored in other projects: such as the *simulation* variant of this solution at the cache line level for hardware shared memory protocols [142], while other work demonstrated that the filtering approach when applied to a *home-based* protocol may perform very well for data parallel decomposed (iterative) applications, where a regular steady-state sharing pattern is present [85]. Where the SMG *implementation* of such a coherence protocol differs is in its dynamic, distributed, and scalable nature and (implicit) support for a hierarchical topology suitable for multi-site grid applications.
All protocols need to provide information for the write collection mechanism to identify what areas have been modified. In the *update* protocol described above this was a single

'dirty' bit per local unit of consistency granularity[6]. The subscription protocol provides the same 'dirty' data, allowing for the same write collection implementations (see Section 7.4.2) to be re-used. However, additional information is required as some blocks of the shared object could be invalid, and a mechanism is also required to identify if a block is subscribed to (hence the term subscription list). This means that there are three sets of information required for every block: *invalid*, *subscription*, and *dirty*. A state transition diagram is given for this protocol in 7.4.2.
The implementation of the subscription protocol for use with weak consistency models requires the use of extra synchronisation meta information. For an EC object bound to a barrier synchronisation primitive, the subscription requirements of the process' parent and children (if relevant) are met by maintaining a local subscription list for each, i.e. the local process is aware of their requirements, and can thus filter any write collections according to these lists. The bulk of the benefits of the subscription protocol stem from this ability of processes being able to filter out coherence data that is not required by its parent or antecedents in the barrier tree. However it does require that some sections of coherence data be kept temporarily that could otherwise be disposed of (under the update protocol).
For lock synchronisation primitives a subscription protocol should be unnecessary with EC (the problems of superfluous data transfer should not occur, otherwise the granularity level of the object should be reduced). However, where it is required the maintenance of a subscription list for all other processes may be impractical due to memory resource requirements. If the process size is small then this is feasible, otherwise a record is only maintained for the process that last requested the object and for the process that transferred ownership of the primitive (similar to the lock owner resolution method, see Section 8.2.2). As the benefits only really occur for barriers the rest of this protocol description will be devoted to their use.

Subscription write trapping

A shared object that operates under the subscription protocol has slightly different write trapping requirements than for the update protocol. The principal difference arises when a block is accessed, but is invalid. This requires a valid copy of the block be fetched from an available source. When a response is received any coherence data can be applied (possibly including subscription information). At this stage only read protections need to be applied to the page. If the original fault was a write then the fault needs to be repeated and then the write permission will be applied. This is the main source of extra overhead in the DSM engine when compared with the update protocol.
When an access fault occurs occurs to a block that is invalid, the DSM must identify a suitable process to direct the request to. The location of a source may not be known directly, but there are hints available in the subscription lists of family members of the node. It must be noted that these will only indicate the best direction in which to issue the request. If a process receives a request for a page and it does not have a valid

[6]throughout this protocol description the local consistency unit will be referred to as a block

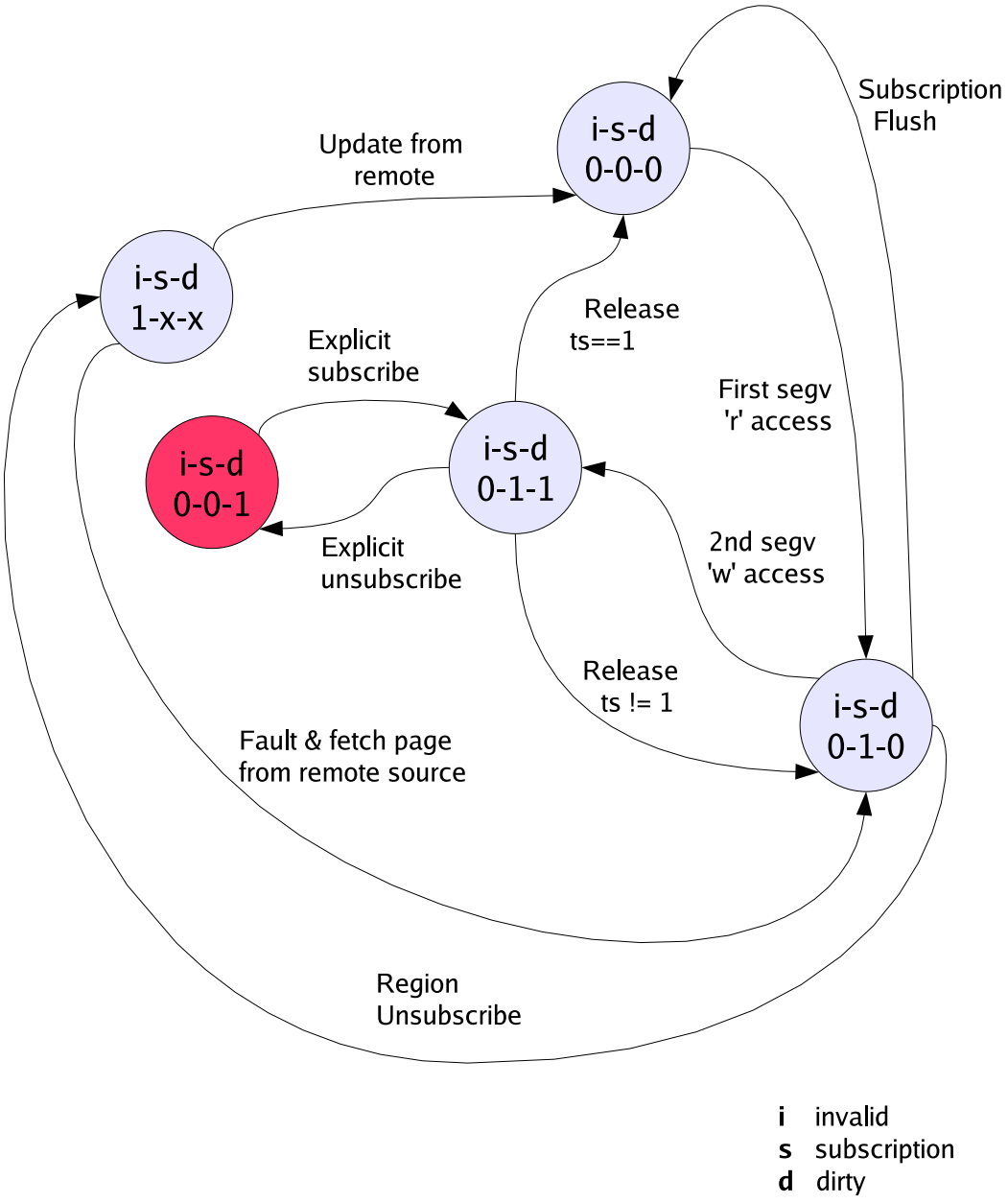


Figure 7.7: *Subscription Protocol State Diagram*

copy then it can forward the request based on its own subscription lists. The maximum number of hops that a request may take before reaching a process with a valid block will be a function of the barrier implementation. For the current SMG (binary) barrier implementation (see Section 8.3) this will be $2(log_2(n)) - 2$.
Section 4.6 highlighted the difficulty that invalidation-based protocols have when operating in a heterogeneous environment where processes may have different page sizes. The Subscription protocols also has this problem. This is solved in the implementation by enabling block size translation[7], allowing for a block request of a different size to be translated into the local equivalent. Issues around the fragmentation of blocks can arise where a process may only be able to transfer a partial request. Currently this situation would be handled by another block request.
In (user) multi-threaded applications the access race issue described in Section 7.6 may arise if not provided for in the manner prescribed.

Subscription Cold start

When an object is first accessed (logical timestamp = 0) the subscription lists for an object, for the node itself, its children, and importantly the parent, may not exist. An effective solution is that on the first release, all modifications are sent to the parent at the first event, i.e. that the parent subscribes to all modifications.
The consequences of this decision to 'cold start' the protocol in this way is that the coherence behaves like an update protocol until a steady state sharing pattern can be established. This also highlights one of the flaws in the protocol: an inability to deal with highly irregular sharing patterns. This demonstrates that benefits are only obtainable in iterative-style applications where other processes can learn the access behaviour of a particular process (and thus know required modifications to send) for previous intervals. However, many parallel applications would have such a discernible access pattern. The situation whereby a transient access can be 'pruned' is discussed in Section 7.5.

Subscription write collection

The write collection process is similar to the *update* protocol, i.e. all blocks with their dirty bit set are write collected. However, before the same write collection implementation can be used with the subscription protocol the dirty bit-set must be cleaned of references to any stale block that may be present, i.e. blocks where the invalid bit is also set. This pre-release clean operation on the dirty information is a simple matter of applying the operation specified in Equation 7.4.
A possible optimisation would be to delay the time for write collection due to the coherence's release operation, thus allowing only blocks that were required to be processed by the write collection mechanism. However, this would introduce additional latencies into the system, and is not provided for at present.

$$Dirty\_list \quad = \quad (Invalid\_List \ \oplus \ Dirty\_list) \ \& \ Dirty\_list. \tag{7.4}$$

[7]The proviso is that block sizes must be multiples of each other 4kB/8kB

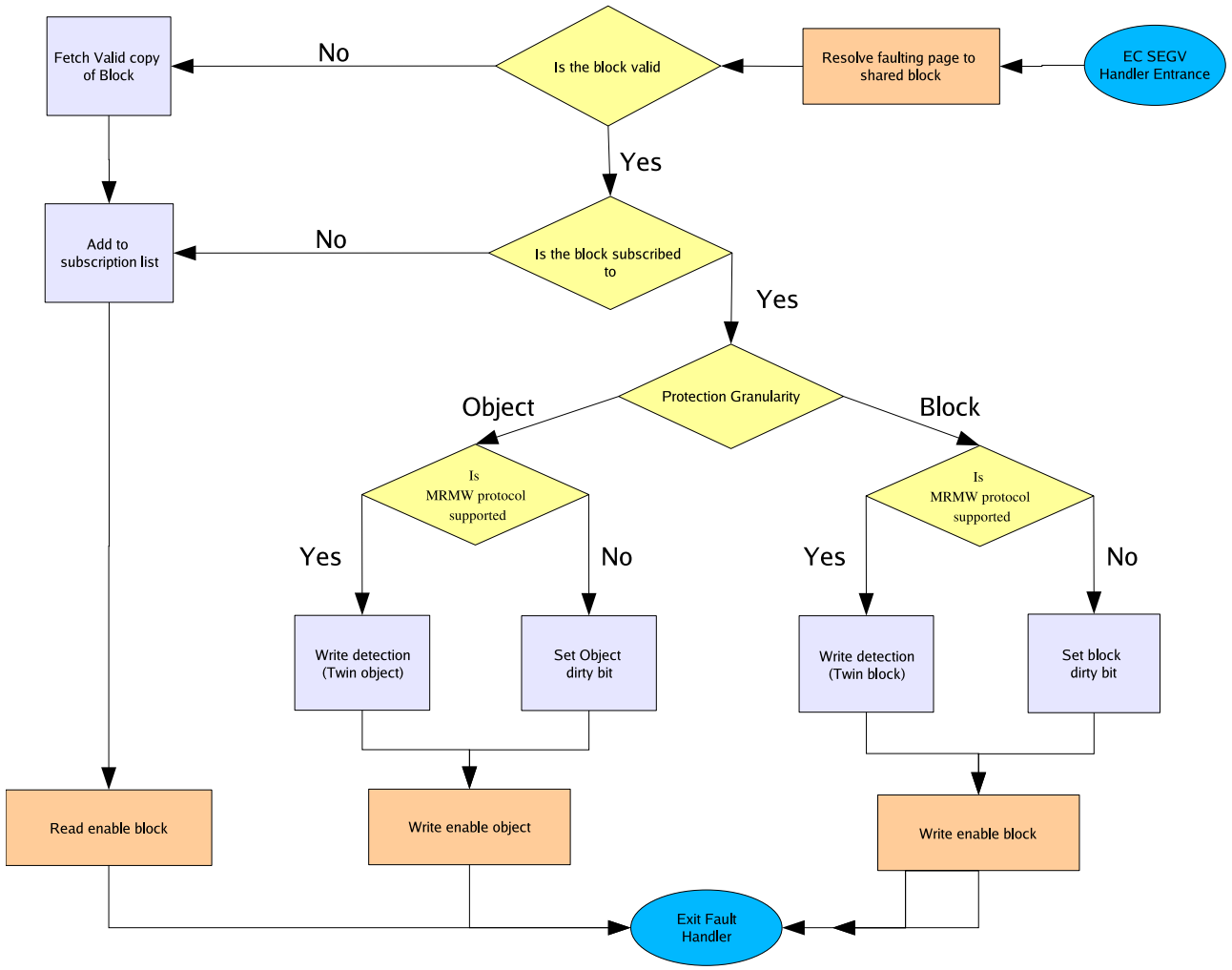


Figure 7.8: *Subscription Protocol Trapping Handler*

Subscription Release-Acquire

Upon a release event a process will take its own write collection for the shared memory region, and together with those of its antecedents will generate a suitable coherence message for the parent based on the available subscription list.
If the subscription list has changed in the preceding interval then this is signalled to the barrier parent at the release stage (notification is packaged in the eventual message generated by the release stage of the protocol), and to the antecedents in the acquire stage. In addition, periodically the process' subscription list can be flushed after a release and before the acquire stages of a barrier. A similar feature demonstrated benefits by reducing the volume of invalidation messages when implemented in shared memory machines [143]. This feature enables the pruning from the subscription list of pages that possibly were 'transient' accesses. These mechanisms provide the degree of dynamism in the subscription protocol.
During the acquire stage the process of filtering the coherence messages must be repeated for all antecedents (based on their subscription). At this stage a number of write collections must be filtered: that contained in 'parent' acquire coherence message, the local write collection, and that of any *other* antecedents. The latter two stages can potentially be done during the wait period between the sending of the release and the receiving of acquire messages.

Subscription Protocol - Scenario

In this worked example a four-process application performs a computation for a number or iterations, similar to that of the Laplace application described on page 150. All processes are allocated an equal share of the total workload to compute. The data set to compute on is six virtual pages in size, so each process will compute 1.5 pages each as shown in Figure 7.9(a) below. At the end of every iteration each process will then require the boundary elements of its neighbour(s). In the example depicted in Figure 7.9(b), Process 1 (P1) requires 0.5 pages from processes P0 & P2), as will those neighbouring processes will require of it.

Given the above scenario, for each process the update coherence protocol would receive all modifications from all other processes for all iterations of the application. Figure 7.10 below depicts the difference in data transfers that occur between processes at different stages of the barrier operation for the application operating at steady state (from iterations 1 onwards) under both update and subscription protocols. The SMG barrier algorithm arranges the binary-tree so that Process 1 is the root (barrier coordinator).

Figure 7.11 shows the state diagram of the subscription protocol meta-data (*invalidate*, *dirty*, and *subscription* bits - I, D, & S) for the shared memory region during the initialisation and the first two iterations of the application described above.

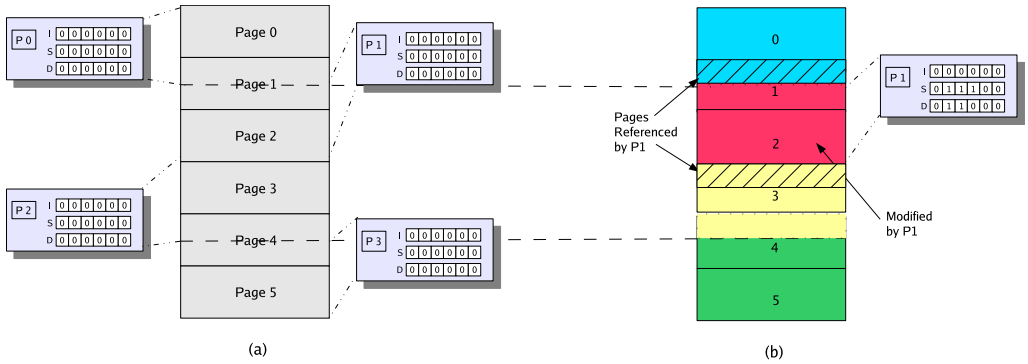


Figure 7.9: *(a) Virtual memory pages modified per process, (b) and the pages subscribed to by Process 1 (P1)*

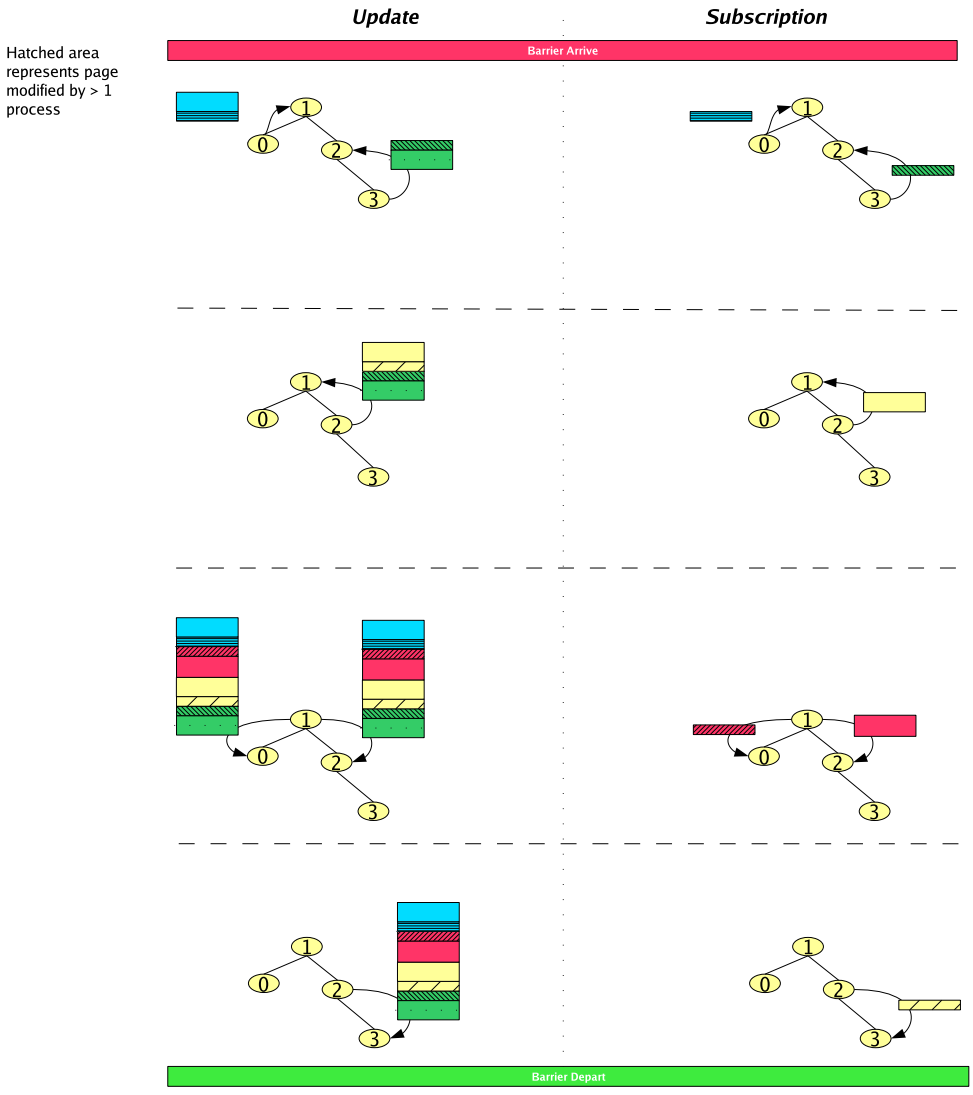
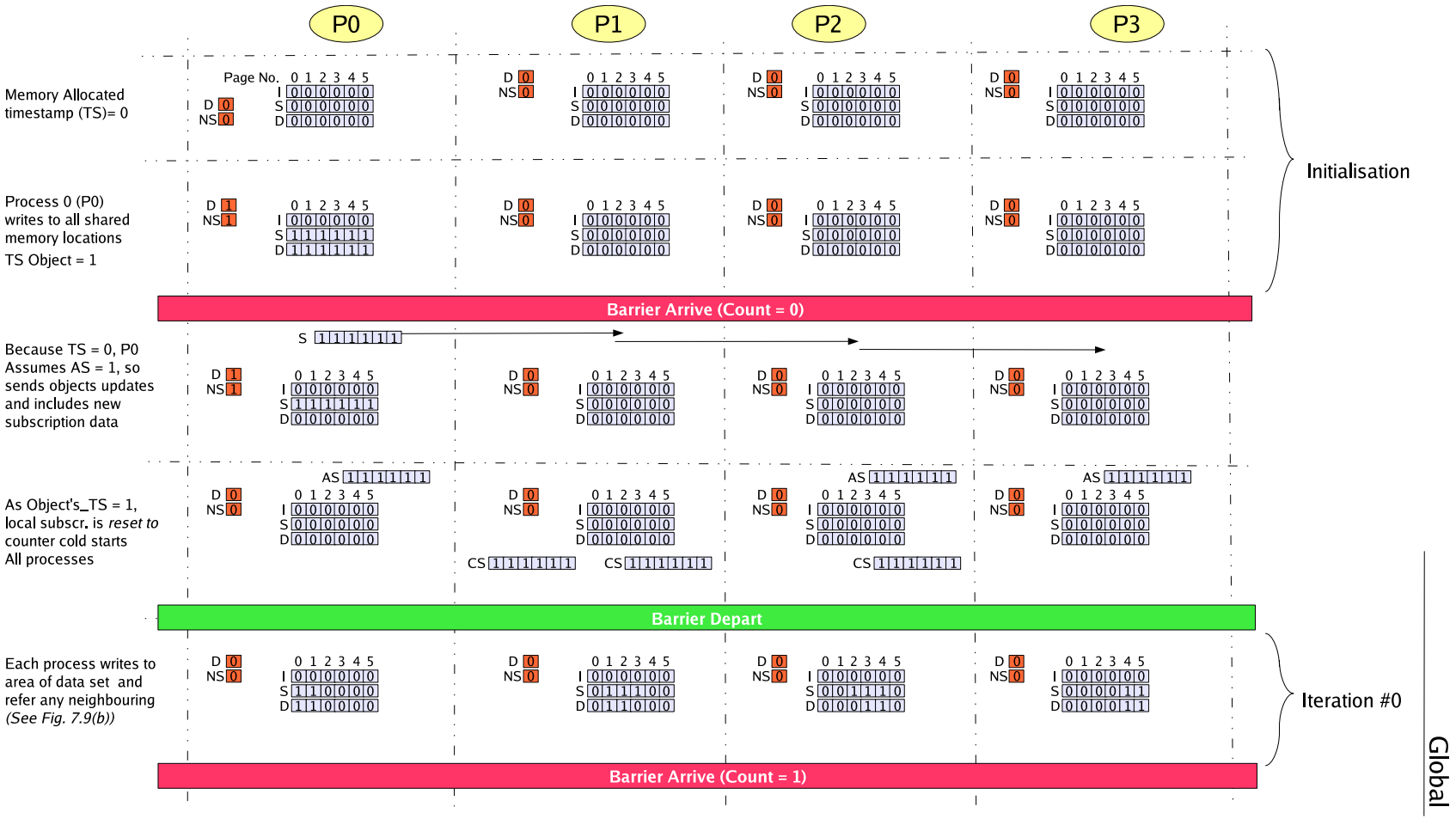


Figure 7.10: *Data transfer under different different protocols*


P0
P1
P2
P3
Memory Allocated timestamp (TS)= 0
Page No. 0 1 2 3 4 5
D 0
NS 0
I 0 0 0 0 0 0
S 0 0 0 0 0 0
D 0 0 0 0 0 0
Process 0 (P0) writes to all shared memory locations
TS Object = 1
D 1
NS 1
S 1 1 1 1 1 1
D 1 1 1 1 1 1
Initialisation
Barrier Arrive (Count = 0)
Because TS = 0, P0 Assumes AS = 1, so sends objects updates and includes new subscription data
As Object's_TS = 1, local subscr. is reset to counter cold starts All processes
AS 1 1 1 1 1 1
CS 1 1 1 1 1 1
Barrier Depart
Each process writes to area of data set and refer any neighbouring (See Fig. 7.9(b))
Iteration #0
Barrier Arrive (Count = 1)
Global

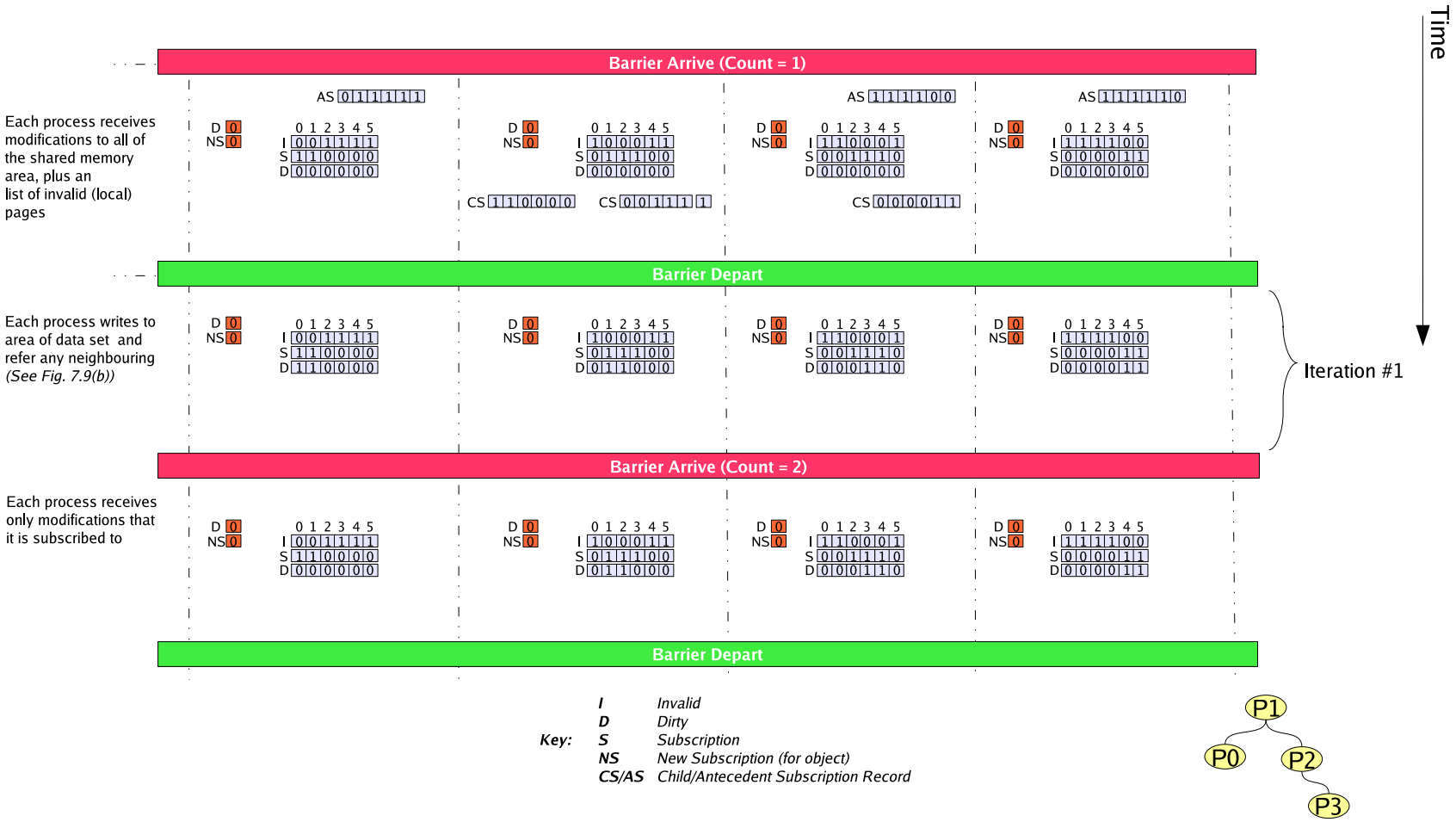


Figure 7.11: *Subscription Example State Change Diagram*

This example demonstrates the advantage that the subscription protocol has over the update protocol in terms of aggregate data transfers between processes during a barrier operation. The data transfers illustrated in Figure 7.10 would be greatly increased in this application if the number of processes was increased (anf the application data-set size increased commensurately) due to the 'multiplier' effect associated with the update protocol.

7.5 Write collection

The process of collecting all modifications to a shared memory object, A, in the preceding interval $x \rightarrow y$ (i.e. since the last synchronisation operation), is known as *write collection*. It is in effect the construction of a patch (A') that can be applied to the original object A_{tx}, such that the resulting shared region A_{ty} is consistent: $A_{tx} + A' \rightarrow A_{ty}$. The method differs depending on the required type of access mode to the shared object and on the strategy taken for write trapping. If in single-writer mode then the whole shared object object can be considered as the write collection. This has the benefit that there is no memory resources are given over to computing the collection. However, if little of the object has been modified then excess communication overhead results from transferring the unmodified data.
There is no method currently available to examine the state of the page protection bits in user space, i.e. there is no complimentary call to the *mprotect* function. Still, if such a function was available it would only provide a method to examine the shared region for modifications at a coarse granularity. To overcome this problem, the DSM provides all shared objects with this information. Section 7.3 explained that when a page is modified the associated coherence 'dirty' bit is set by the DSM page fault handler. This allows the write collection process to avoid processing pages that have not been modified. When write collection is performed with the use of the *Twin-per-page* write trapping, the amount of the shared memory region involved in the computation of the write collection may be significantly reduced.
When is write collection performed? This is dictated by the coherence protocol, usually triggered during the release phase. Some coherence protocols can take a 'lazy' approach and defer write collection until the region is actually requested [96], but this will introduce extra latency. If a diff is created at a release point, and the same region is accessed immediately afterwards by the local node, then the diff must be recomputed which involves superfluous computation overhead. In SMG, this could potentially be performed during the wait period for an acquire operation (latency hiding). If merging of two write collections for a given interval gives rise to a conflict (i.e. concurrent writes clashing to the same location) then a user-specifiable fault handle is executed. The default user-specified fault handler simply exits.
A different write collection mechanism can be utilised by implementing the write collection interface (Listing 7.4). The salient methods allow for creation (*collection_generate*), merging (*collection_merge*), and application of write collections (*collection_apply*). The two implementations for SMG described below balance the processing overhead and the

eventual write collection size differently.

```
typedef struct _collection_blk{
  void *library; int identifier;
  char code[8]; char description[desc_size];
  int (*collection_init)();
  int (*collection_finalise)();
  void (*collection_set_attrib)(Handle *obj_ptr);
  void *(*collection_gener_sim)(Handle *obj_ptr,int flags);
  void *(*collection_gener)(Handle *object_ptr, int flags);
  void (*collection_apply)(int memory_size, int *memory,
                           collection_header *diff);
  void *(*collection_merge)(collection_header *wcoll_a,
                collection_header *wcoll_b, int *error);
  void *(*collection_merge_here)(collection_header *wcoll_a,
                collection_header *wcoll_b,
                collection_header *wcoll_merge, int *error);
  void (*describe_collection)(collection_header *wcoll);
}collection_block;
```

Listing 7.4: *SMG write collection interface*

7.5.1 Diff write collection

The diff method generates a run-length encoding of the modifications to the shared memory region. This is achieved by comparing the modified page with the twinned version of the original (so it can only be used when the twinning variants of write trapping methods are used). Therefore when consistency is required only modifications are transferred by the coherence protocol. This approach has been used in Treadmarks [96].
This method supports multi-writer protocols as conflicting writes, that may occur during an interval, *x-y*, can be easily detected. This will be done by the (*merge*) of two distinct *diff* write collections. There is significant overhead in the processing of diff write collections, but this is the trade-off that occurs to minimise communication usage. There is the potential for problems with this approach, as each contiguous region that is modified (termed a patch in SMG) requires a patch header specifying offset. If writes are interlaced at small frequencies, then the patch header requirements become significant, hence the benefits are reduced. Section 9.4.2 highlights such a scenario.

7.5.2 RAW write collection

The *raw* write collection technique dispenses with the creation (thus the overhead) of the run-length modification encoding of the diff approach outlined above by transferring modified blocks (a block will be the size of a local page as outlined in the section on write trapping). This can be a better solution when the subscription protocol is employed

for an object as less resources are required to process the coherence messages from the antecedents or the ultimate root. Another scenario where it is useful is where the writes to the object are highly interleaved from different writers (a problem with the *diff* approach).
At the time of write collection the object dirty information is examined. Where a block is dirty then it is added in its entirety to the write collection, so processing for potential write conflicts is only done when required. At the merging of two *raw* write collections, blocks relating to the same shared memory region will be compared to an original (twinned) copy; if all three are different at a given offset then a clash has occurred, and the appropriate merge-clash handler is executed.

7.6 User Multi-threading Issues

As support for multi-threaded user applications is required, additional consideration needs to be given to how this will effect the overall operation of the DSM.

- **Access Race Condition:** Consider a scenario where two user threads within the same process modify locations within the same shared object[8] at the same time. A race may arise if both locations are in the same page. The DSM may be updating the page in performing coherence events for the first thread, and to achieve this the DSM must have write access to the shared region. While the DSM is performing the coherence event the second user thread may make modifications unbeknown to the DSM (as no write trap event is generated).
  The solution is depicted in Figure 7.12, and requires that at object allocation the DSM maps the physical memory for the shared object to two different virtual locations (this is termed dual-porting). This allows the DSM to modify the data according to its own access protections, independently of the mapping for the user threads.

- **Signal Delivery:** As mentioned previously the Linux Virtual Memory (VM) system delivers signals on a per-process basis. However, as there may be multiple threads of control it is important that the SEGV write trapping mechanism be aware of this; it is important that mutual exclusion be maintained to the data structures responsible for management of the shared area.
  For Linux 2.6 kernels and later, native threads are now supported, so threads are scheduled rather than processes. The recommended method of handling signals in multi-threaded applications is to mask all signals in all the threads and to create a single thread with the responsibility to handle signals delivered to the process. This can be achieved using the *sigwait* call. However, currently the SEGV signal cannot be blocked on a per thread basis.

[8] the object is covered by an invalidation protocol, or variant like Subscription

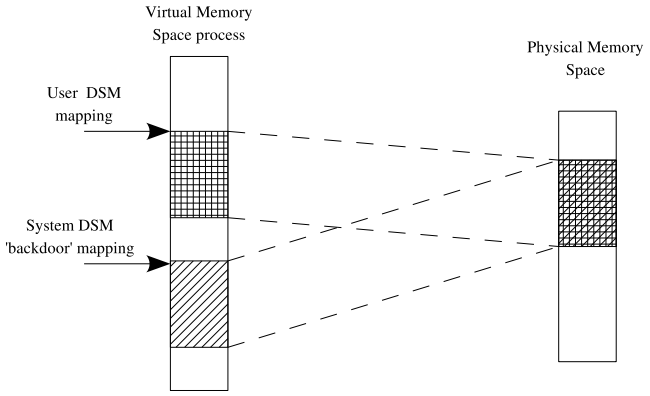


Figure 7.12: *Dual porting shared memory mapping*

7.7 Implementation issues

There are numerous issues that arose then implementing the shared memory management of the DSM. A subset of these are:

- **Kernel support:** One of the issues that arose while developing the shared memory functionality of the DSM is that a SEGV access is unrecognisable, due to lack of support for MAP_ANONYMOUS flag to the *mmap* system call in kernels prior to version 2.4.1. This is also related to the lack of a complementary call to *mprotect*.
- **(Physical Memory Use:** To provide transparent shared memory the DSM incurs *tremendous* overhead in physical memory usage for write trapping & write collection (three times the actual size of the region [data, twin, write collection]). This will only be removed if different write trapping & collection techniques are used such as compiler instrumentation. Home-based coherence protocols can only eliminate the resources required for a twin/diffing on the page's home node.
- **Virtual Address Space:** While not requiring significant system resources, the dual-porting of the shared memory region (see Section 7.6) is achieved at the expense of virtual memory usage. In 32-bit architectures the virtual address space is 4GiB, and when typical OS considerations are factored this is 3GiB. This implies that the maximum shared memory region that can be allocated will be 30% of the remainder[9]. This problem should not be so significant with the large virtual address space provided by 64-bit systems.
- **Heterogeneity:** As computational grids are composed of machines with potentially different architectures this involves differences in the native representation of data on those architectures (big/little-endian). This is a concern when transferring shared data between nodes. Message passing programs overcome this by

[9]In practice this will be much lower due to other requirements e.g. communication

'typing' the data when it is sent between processes for every send/receive call, for example *MPICH-G2* takes explicit actions in this regard.
SMG currently has no support for heterogeneous systems. Memory is viewed as a linear array of bytes. While desirable the lack is somewhat justified from a number of aspects: (i) (Increased burden) this would require the application developer to strongly type the shared data so that the underlying system would have a *type map* when transferring shared data. This would give rise to excessive processing and extra messages. These requirements would add a considerable burden on the programmer [144], and on the DSM system. (ii) the overhead involved would be unfavourable (iii) the vast majority of grid compute resources use the Intel x86 (or 64 bit versions) architecture.

- **Fault Tolerance:** Some fault tolerant schemes are impractical [136] due to the massive overheads involved, others require a holistic approach approach that are difficult to maintain and port, while some use check-pointing and restart. For SMG it was decided that all that would be provided would be to allow persistence of objects at a given point in time (on a global barrier). Further work can easily expand this to the DSM engine.

7.8 Discussion

In the SMG system it was important not to hard-wire defaults such as page sizes as this would reduce future portability potential. These are variables that can be obtained at run-time [28]. This has the effect of increasing slightly the DSM overhead.
The implementation of write-trapping and write collection follows a similar approach to that adopted in Treadmarks [96] with some small modifications. Write-trapping is achieved by setting the protection level of the shared object, where the minimum granularity is at the virtual page level up to the total size of the object if it occupies more than one page. The default behaviour is that a twin is generated upon the first write to a variable in a shared region.
When the synchronisation object that the shared memory object is bound to performs a release a diff is generated by comparing the twin and the current state. This is used to minimise the message traffic for coherence updates. If topology information is available then hierarchy awareness can be applied in barrier operations, i.e. as arrival notifications are received the diffs can be merged at intermediate nodes (as depicted in Figure 8.3), thus reducing the processing bottleneck at the root node level. Otherwise, a traditional tree-structured barrier is employed.
Multi-writer entry consistency can be easily supported for barrier primitives and this enables support for OpenMP *parallel for* constructs. Basic user-specified alteration to write trapping and write collection methods are allowed, and when more integration with the information system occurs, this user control will assist application-level optimisations where access patterns to shared data are irregular [98].

CHAPTER 8

SMG Synchronisation

This chapter will describe the implementation in the SMG DSM system of the two core concepts used in the synchronisation of parallel applications that were discussed in Section 2.4. SMG is a DSM implementation that uses relaxed consistency protocols, and so it is required that the implementation of the synchronisation primitives support this. It is necessary that coherence actions occur at the synchronisation points when required. As EC is the chosen consistency protocol it is necessary to explicitly associate, or bind, shared memory objects with a synchronisation primitive. This increases the burden on a developer, and this has been a source of problems in previous DSM implementations that employed this consistency protocol [137]

In distributed shared memory, accessing shared memory is tightly coupled to any synchronisation variables whose implicit role is to enforce the consistency of that memory. The access mechanisms were discussed in Chapter 7. The API functions that provide access to distributed locks that provide different levels of access to the guarded critical sections, are discussed below. The barrier synchronisation primitive differs in its use in a DSM application as it allows the use of multi-writer protocols. Frequently an object was bound to the incorrect synchronisation object resulting in a degradation in performance; when this occurs the problem must be identified to the developer.

Problems that may occur in normal parallel programming tasks that are difficult to diagnose, such as deadlock and live-lock, mainly arise from poor or inaccurate algorithm implementation by the developer, or a high lock contention rate. If such occurrences can be highlighted to the developer, then that allows the situation to be remedied.

The problems encountered in the implementation are discussed at the end of the chapter.

8.1 Synchronisation

In the discussion on weak consistency models it was seen that weak consistency operations (Acquire & Release) map onto the operations required for synchronisation. The synchronisation API requirements can be summarised as:

- **Scalability & Efficiency:** Synchronisation is an integral part of parallel computing and should be as efficient as possible, so it should incur minimal communication costs. Any scalability issues that are evident from cluster use will be magnified if used in a grid.
- **User Multi-threading:** support for multi-threaded applications is required, so the synchronisation primitives themselves must be capable of supporting access to multiple user threads.
- **API Generic:** Section 5.3 highlighted the desire to support the implementation of higher level APIs, such as OpenMP, and this dictates that the characteristics be as generic as possible.
- **Consistency & coherence agnostic:** As previously stated the initial consistency implementation will be EC. Ultimately to support extensions to the DSM (in the form of new consistency models, coherence protocols etc.) it is necessary that the DSM engine, and hence synchronisation functions, should be agnostic in this respect, so the prototype cannot be 'bound' to EC.

8.2 Lock Primitives

The SMG implementation of a lock primitive aims to meet the general requirements of a mutex device that were outlined in Section 2.4.1, and the extra requirements imposed above. In SMG a lock primitive is referenced using a global numerical identifier, that acts as an opaque reference to a local DSM engine structure similar to the shared memory handle structure (see Listing 7.1), allowing for the transparent use across the system. The lock variable may be held exclusively by one thread of execution (termed *write* mode), or non-exclusively by a number of threads concurrently (*read* mode).
All locking routines are required to be thread-safe with respect to the local process. Where a lock is in exclusive mode it must be able to distinguish between user threads on the local node. This is achieved by the thread identifier being recorded in the internal lock structure when a lock is acquired.
A lock is initialised with the following declaration. The *lock_identifier* field specifies the global lock identifier, while the *flags* field is used to specify required attributes for the lock. In a similar manner to other projects [89], *flags* specify if shared memory is to be associated with the lock, or the behaviour permitted for multi-threaded environments, and if the locking protocol can be optimised to the consistency/coherence protocol.

int **SMG_lock_declare**(int lock_identifier, int flags);

8.2.1 Lock acquisition

When a lock needs to be acquired by a thread it must wait for the request to be satisfied. A process that can grant the required access level is found and it grants the requested access. The conditions by which a process can grant the requested access are outlined below for each relevant mode.
A lock acquire operation can be mapped to the acquire operation in weak consistency models. Upon the lock being acquired all relevant associated consistency model information will be included in the grant notification by the grantee, and the lock acquisition will apply coherence information received, thus when the operation has completed all (relevant) shared memory is consistent. For this reason non-blocking lock acquire operations are not supported.

Non-exclusive (read) lock acquisition

In respect of the DSM where the EC model is used, all shared memory associated with the acquired lock is guaranteed to be consistent until the lock is released. This data cannot be modified as the lock is non-exclusive mode, so the thread has only acquired read permissions. If another thread within the same process incorporating the requesting thread has been granted access, then lock acquisition is a relatively straightforward task involving updating a local counter; otherwise a remote lock granter must be found.
The first location to be found is the probable owner process, although there are other variations as discussed in below on Section 8.2.2. The API call to acquire a lock with the global identifier *lock_identifier* is:

int **SMG_read_lock_acquire**(int lock_identifier);

The return code specifies the status of the operation. *SMG_SUCCESS* indicates that the lock has been successfully acquired in non-exclusive mode; otherwise an error such as the the general error code *SMG_FAILURE* is returned.

Exclusive (write) lock acquisition

The exclusive acquire call has potential for high latency as only the actual owner can grant the request for access. In addition the requester must wait for the current owner to finish using it. Furthermore, there may be prior requests for access that are ahead in the request queue.
As shown in Figure 8.1, the owner of the lock is the only process able to grant exclusive access by transferring ownership to the requesting node. The following is the API call for issuing a request for a lock when the global identifier *lock_identifier* is in exclusive mode:

int **SMG_write_lock_acquire**(int lock_identifier);

Again, the return code *SMG_SUCCESS* indicates that the lock has been successfully acquired in required mode; otherwise an error such as the the general error code *SMG_FAILURE* is returned.

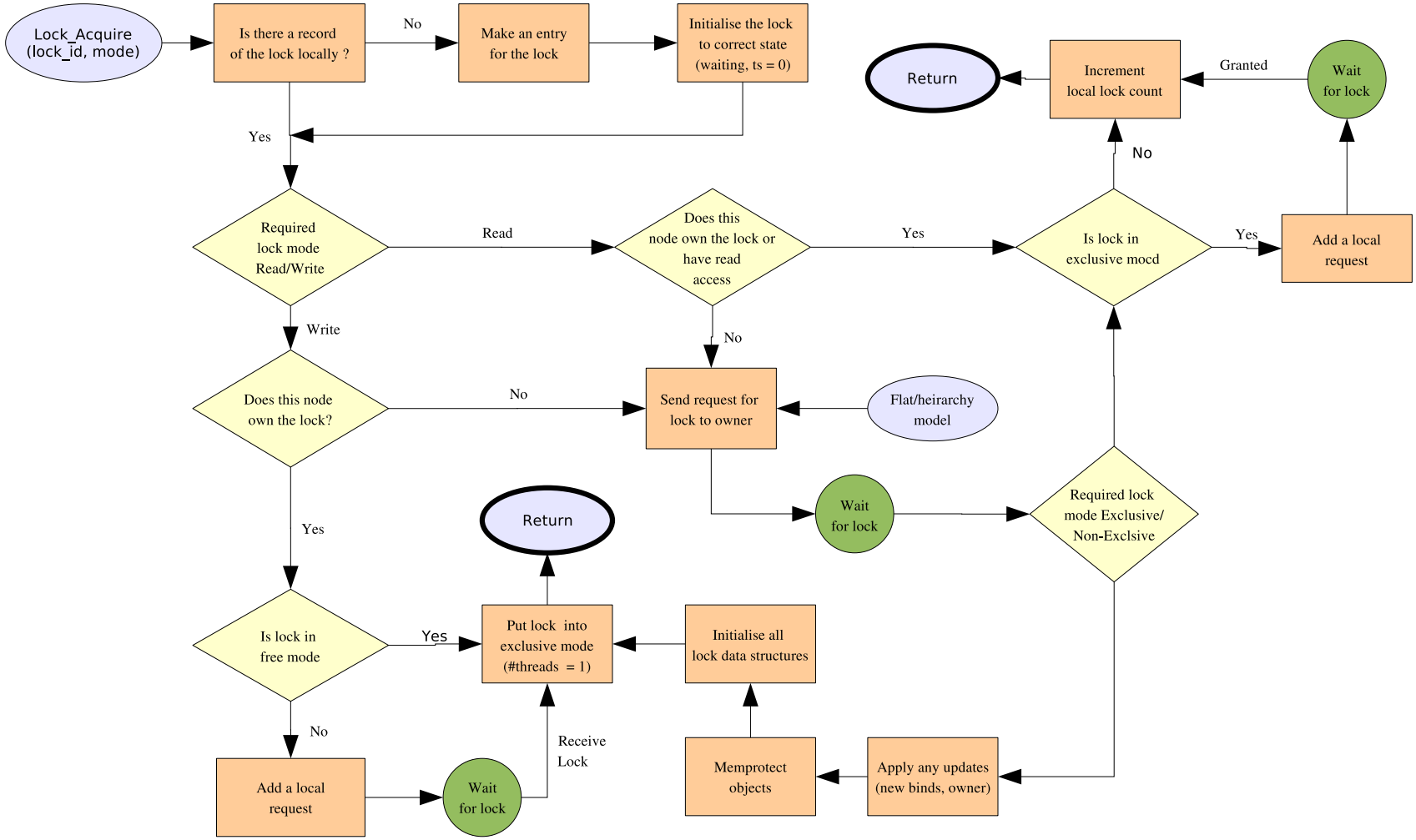


Figure 8.1: *Lock Acquire flow diagram*

8.2.2 Finding the lock owner

When a user thread requests access to a lock primitive in exclusive mode the current owner must be found. With a non-exclusive request any thread holding non-exclusive access will suffice, but the same principle applies. It is important that this is achieved in the most efficient manner, avoiding delays occurring in the subsequent computation to be performed by the user thread in the critical section. If the current owner of a lock cannot be found efficiently, extra latency results, with degradation in performance proportional to the lock contention.
As memory is bound to synchronisation objects the access patterns to locks will be similar to the shared memory access pattern. For this reason the distributed memory management algorithms developed by Li [65], as discussed in Section 4, are highly applicable in the development of a distributed lock management protocol for a DSM. When a lock request is made and it cannot be satisfied locally, then finding the owner of the lock can introduce severe performance penalties into the system. What is required is an efficient procedure so that the requesting thread is not blocked for too long. There is always a route to the owner through a chain of probable owners.
The SMG system employs a *N degrees of separation* approach, where all locks have a static manager that knows the current owner through a chain of probable owners of the lock. This chain of probable owners is composed of the previous owners of a lock. When a lock is first requested the manager will record the identifier of the requester. When this requester passes ownership to another node the identifier is recorded. This trail will always lead to the current owner of a lock. When a process receives a lock request that cannot be satisfied locally it will pass the request to the process it believes is the current owner. If the chain of probable owners is larger than N then a purge message may be sent.
In a grid environment with information available it is possible for the owner chain to be shortened by creating proxy lock managers at sites where the actual lock manager is not located. When a process receives lock ownership from a process in a remote site then the site proxy lock manager is informed that the lock is located on the local site. Any requests that originate from a site can be first directed to the local proxy rather than initiating a high latency request to an probable owner located off-site. If a request is received by a proxy, and the lock is not located on the local site then the request can be forwarded to the probable owner. Messages of this type need only be generated when a lock arrives or leaves a site.

8.2.3 Queueing lock requests

When a request for lock access is received by a thread, and where the lock is still in use (in exclusive mode) and thus conflicts with the status, the request must be queued until it is no longer needed. Where multiple requests are outstanding a request queue must be formed.
The method of queueing the lock request is an important factor. Poor queueing discipline can result in lock starvation and poor performance. The simplest approach is based

on a first come first served principle. This approach works well on a multiprocessor as it guarantees lock fairness and thus avoids lock-starvation. In a distributed machine or grid setting this method can result in poor performance if the lock contention is high, and/or involves requesters from multiple sites 'trashing' the lock among them.
Distributed lock implementations have been proposed where a weighted balancing approach is used [145, 146]. Here queueing discipline is governed on the location of the requester per process, per node, per site or other, whereby 'preferred' access is granted to threads nearer the current owner. By following this approach better performance is achieved as local user thread requests can be satisfied first, lessening of the ping-pong effect where a lock is passed between processes. This algorithm mitigates the potential problem of lock starvation.
The SMG DSM implements this approach by assigning each lock a site quantum, where queueing discipline ensures a fairness approach where one node or site cannot have continual access to the lock. If grid hierarchy information is available then requests can be directed to the local proxy for the site. If there are other requests outstanding at the same time, then the request can be queued locally (not implemented in SMG yet). The factors that influence the lock queueing mechanism include the process_identifier, timestamp, and the request_type (exclusive or non-exclusive).

8.2.4 Lock release

Upon the release of a lock the action that occurs depends on the mode in which the lock is held. This process is depicted in Figure 8.2. The API call to perform a release of a lock that is held under any access mode is:

int **SMG_lock_unlock**(int lock_identifier);

On the release of a lock held in non-exclusive mode, the owner field is first examined. If the releasing node is not the current owner of the lock then it will notify the probable owner. It can only do this immediately if it has not delegated equivalent ownership to another process, or no other local thread has.
If the lock was held in exclusive-mode then the process checks if there are requests pending. If so the first in the queue can be satisfied. When a lock held in exclusive mode is released all bound objects must be examined to check for modifications. If bound objects are modified the next owner of the lock must be given the update notices. All locks operating in EC mode have a 'dirty' bit that indicates if any bound object has been modified. This bit is examined upon delegation of lock ownership although a difference in lock logical time-stamp can also indicate this. More information follows in Section 8.3.3.
Where another thread within the process of the releasing thread requires exclusive access, then no release action is performed with respect to the consistency protocol (memory is already guaranteed to be processor consistent in this case). The benefit of this is that no coherence action results, so by association no write collection handler is invoked, and so the handover is immediate.

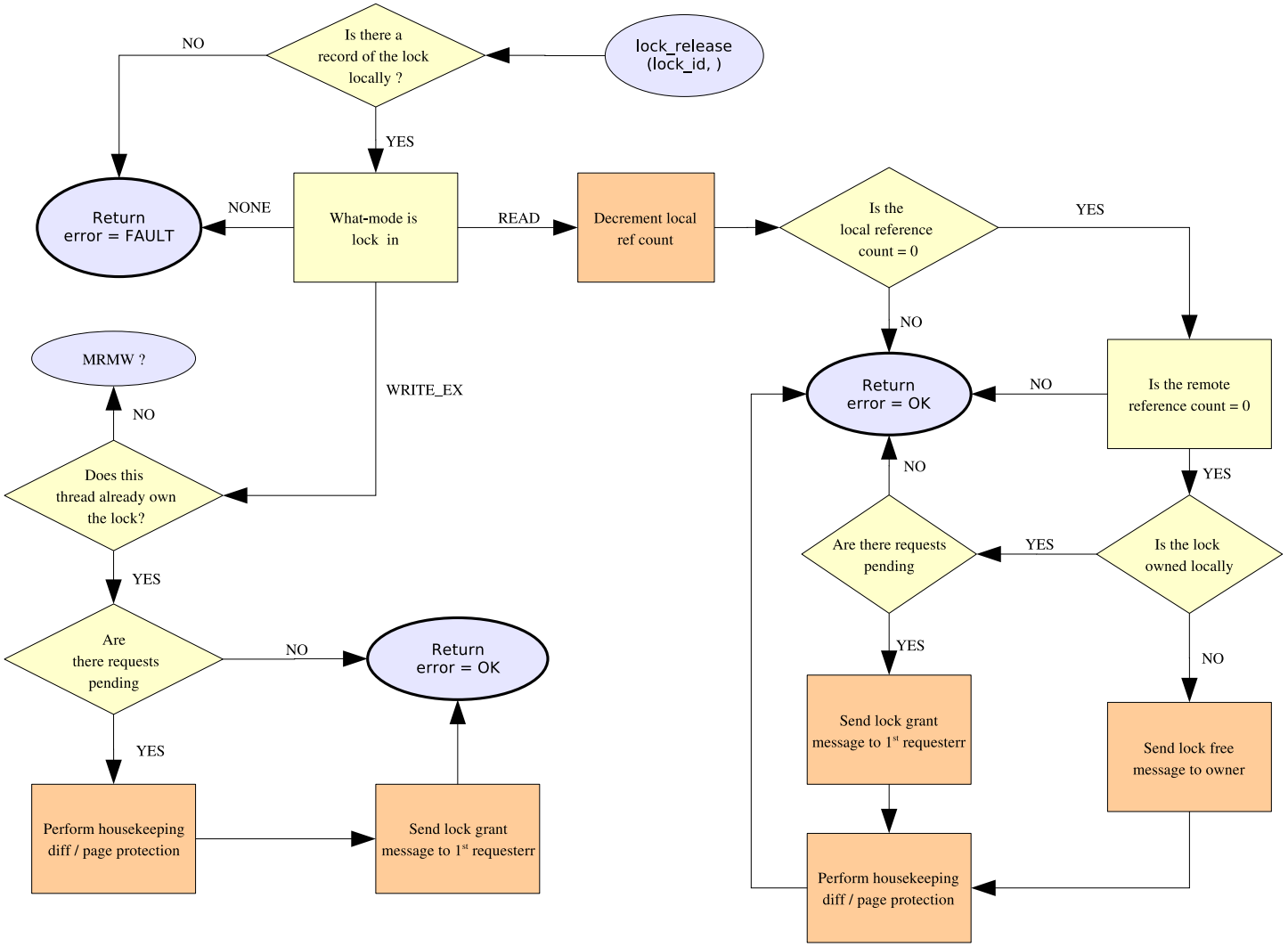


Figure 8.2: *Lock Release flow diagram*

8.3 SMG Barrier Primitive

In a similar manner to the lock devices described above, all barrier operations of the SMG API reference a global numerical identifier. This in turn is used by the DSM to reference the local (to each process) internal structure representing the barrier primitive. The issues of distributed barrier implementation were discussed in Section 2.4.2. In essence there are two stages: barrier arrival and barrier departure. In the former stage all threads of execution must declare arrival at the barrier, which must be coordinated in some manner dictated by the barrier algorithm. When this has been achieved the latter stage takes place and all processes may depart by being informed to proceed in a similar manner. The algorithms and functions implementing the actions described in Chapter 7 (consistency, coherence functions, etc.) that occur during the internal between arrival and departure phases of the barrier are discussed in Section 8.3.2.
In a DSM, extra work is involved as coherence events dictated by an active consistency model, described in Section 7.2, need to be performed when all local conditions have been met. Figure 8.3 demonstrates the actions for a shared memory region under an EC model and Update protocol. The coherence information from a particular process is piggybacked on the `BARRIER_ARRIVE` notification. All coherence information is distributed (if applicable) on the `BARRIER_PROCEED` notification. From a weak consistency point of view, the `BARRIER_ARRIVE` stage is synonymous with the release operation, while the acquire operation is synonymous with the `BARRIER_PROCEED` action (this can be seen in Figure 8.3).

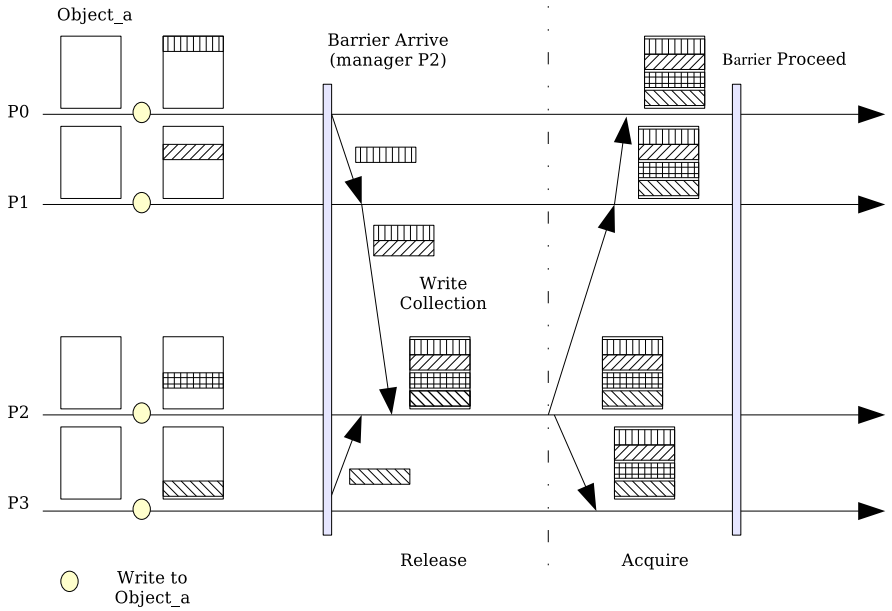


Figure 8.3: *EC-MW Write collection at a barrier*

8.3.1 SMG Barrier API

The default mode of operation of a barrier can be specified by declaring the barrier and its default flags before the barrier is first used. This can be achieved using the *SMG_barrier_declare* function. There are a number of modes that can apply when each process contains multiple user threads. Consider the use of a barrier in a possibly multi-threaded user application. As threads can only be visible at the process level it is important to consider what action is to be taken when a barrier is reached. Should the barrier fire when the first thread reaches it or should it wait until all threads reach it? Currently SMG barriers require all threads to have reached the barrier so that memory can be made properly coherent. If this was not the case a thread could be waiting for barrier proceed notification while another local thread modifies a shared memory section bound to the primitive.
The invocation of the global barrier itself is pretty intuitive. All processes satisfy their local requirements by the user threads, dictated by the mode, invoking the *SMG_barrier* function. These relevant local threads of execution of the process will block until notified by the DSM system of the arrival of the `BARRIER_PROCEED` event. The internal DSM actions performed between the arrival and proceed are described in Section 8.3.3.
Additionally there is a barrier function available to synchronise a subset of the processes partaking in the application. However, currently shared memory objects cannot be bound to such a barrier as coherence updates to a process not partaking in the operation could not be well defined, potentially leading to data races conditions.

int **SMG_barrier_declare**(*int* barrier_id, *int* type, *int* param);
int **SMG_barrier**(*int* barrier_id, *int* flags);
int **SMG_sub_barrier**(*int* barrier_id, *int* who, *int* flags);

8.3.2 Barrier algorithms

There are numerous barrier algorithms available, the appropriate choice depends on the overall requirements. We require a flexible hierarchical distributed barrier. The central server implementation of the MPI barrier primitive in MPICH1 & MPICH2 is not suitable for the DSM 8.4(a).
Previous DSMs have adopted hierarchical implementations of barriers [146] which aim to minimise the amount of high-latency inter-site communication. Treadmarks implements the central server algorithm, while other work [147] implements a more efficient algorithm using a Binomial Spanning Tree (BST) 8.4(b).
The approach taken for barrier implementation in SMG is to use a combination of the above. It was shown that significant performance increases can be achieved where multi-site grid applications with collective operations, like barriers, are made topology-aware [52]. An optimal tree for the physical execution environment, *Hierarchy Optimised Tree* (HOT), can be constructed if topology support is available. A binary tree system is used within a site, while a normal tree is used between per-site managers and the barrier manager itself. Currently all barrier primitives rely on the same HOT structure that is constructed at run-time. The maximum depth of such a tree is $log_2(n+1)$.

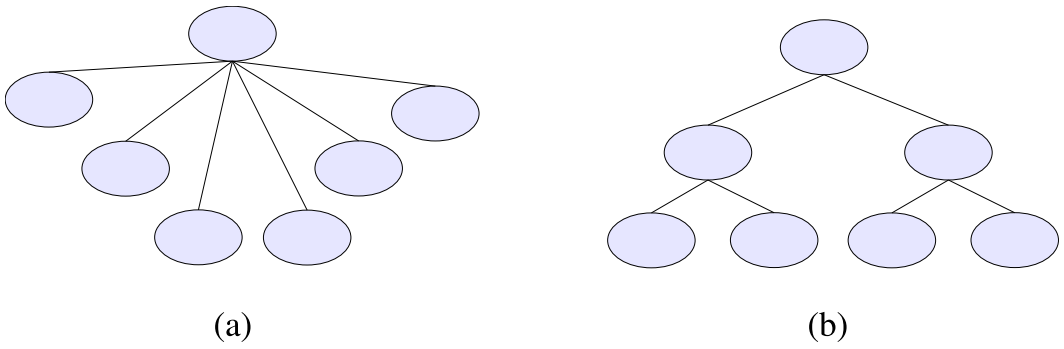


Figure 8.4: *SMG Barrier Algorithm*

8.3.3 Action taken at a Barrier

When a process encounters a barrier the tasks that are performed depend on whether there are shared memory objects associated with that particular barrier. First the local update queue is examined for any pending updates that need to be performed, such as notifications of change of the shared object bound. The most obvious, with respect to the DSM, are the coherence actions that must occur at this point. Any objects that were modified in the preceding interval are updated. The consistency protocol as described in Section 7.5.

The interval between the barrier arrival and barrier proceed is a suitable time to log events to the monitoring system, including resource usage (computation & communication), system statistics such as number of page-faults, the statistics on shared memory that are bound to the barrier, etc. The implementation of this functionality and the actual format of the logged data is discussed in Chapter 9.

The ownership of an object and management of the barrier should transfer dynamically to the process, or a process within the site that performs the most writes. The information & monitoring system could be used for this purpose. The manager will see the updates coming in at the barrier point, and so can make the decision and apply it during the next barrier period.

Binding shared memory with barriers provides the ideal opportunity to allow multiple-writer protocols. During a barrier interval multiple writers are allowed to perform non-conflicting writes to the same shared memory region. When the barrier is reached, the modifications done by all processes are merged by the barrier administrator, and then sent to all processes in the system. Employing multiple-writer protocols allows some further optimisation, such as write collection aggregation. As barrier proceed notices occur the write collections can be merged at this point. This results in reduced inter-process communication as less write collection data needs to be transferred, and the workload of the barrier administrator is reduced, but the computational overhead will be increased. This process is depicted in Figure 8.5.

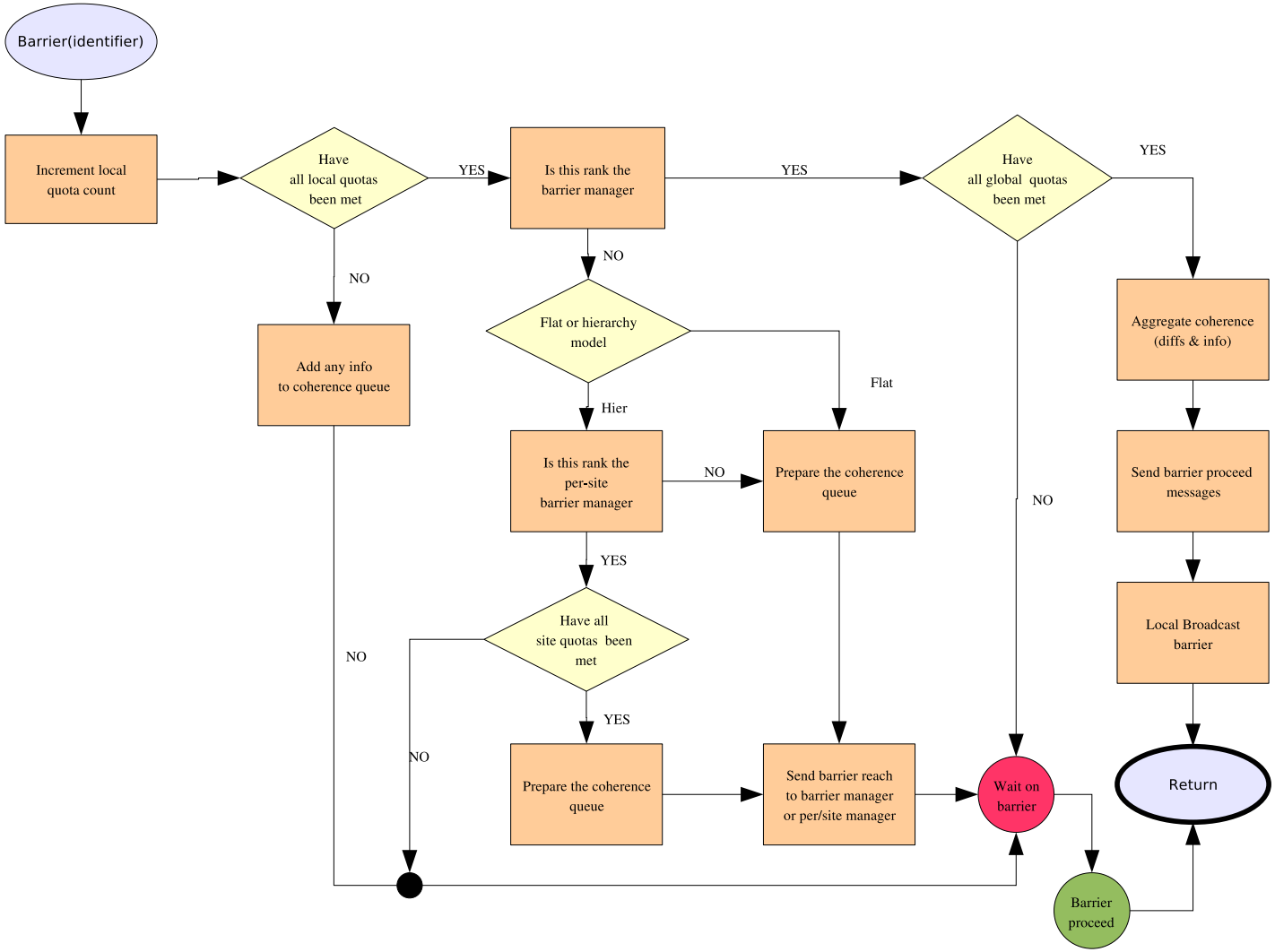


Figure 8.5: *Barrier Event Diagram*

8.4 Implementation issues

Most of the issues arising from the implementing synchronisation primitives relate to the conflict with the shared memory management requirements.

- *Lock Handover*: The lock queueing mechanism aims to ensure that all lock requests are eventually serviced. It cannot be proved conclusively that the SMG implementation for lock queueing mechanism is totally fair as the primary desire was to minimise communication, so under certain circumstances the locking mechanism may perform poorly.
- *Lock Ownership resolution*: When an exclusive lock is required the current owner must be located. This process may require the request to take a number of hops. This, plus the need for eliminating the potential for the lock ownership never to be transferred, introduces extra communication into the system.
- *Barrier Tree Dependent Size*: It is more efficient to implement an K-ary scheme rather than a binary one, i.e a process will have a maximum of K children as compared to the current implementation of two ($k = 2$). This would reduce the depth of the tree, but the potentially significant workload and memory demands required by coherence events mitigates against a larger size k. A low value ($k = 2$) was chosen due to these unknown requirements. A more efficient implementation would set this value dynamically based on the run-time conditions.

8.5 Summary of decisions

The following summarises decisions made in the area of SMG synchronisation:

- Lock primitives employ a queueing discipline that is mindful of the source of requests, and will discriminate in order to minimise inter-process data transfers. If topology information is available then this can be further optimised to take account of the existence of higher latency inter-site transfers.
- Multi-threaded user applications should be supported where necessary, e.g. lock primitives are thread-aware and optimised for this situation.
- All user threads in a local SMG process should arrive at the barrier to allow that process to signal its arrival. This and the implementation of barrier groups (opposed to global barriers) are future work.
- Multiple-writer protocol only allowed when the shared memory object is associated with a barrier synchronisation primitive.
- The period between a barrier arrival and barrier proceed is a suitable time to log application monitoring information if it is required. This is done so as to utilise communication and computation resources that are available at this time.

CHAPTER 9

Optimising for the Grid Application

Chapter 2 briefly mentioned the effects that latency within a parallel computing system composed of distributed processing nodes can have on the overall performance of a parallel application. As the trend indicates that such use is set to increase as computational grids gain acceptance, these problems will become even more apparent. Thus any efforts that mitigate against this are very valuable. One method is to make an application aware of the environment in which it executes. To achieve this, topology information needs to be made available to the DSM system.

As previously described, it is possible that some locations or shared memory regions may be responsible for excessive DSM communication. This may be due to poor algorithm design or implementation, or it may just be an inherent feature of the application. Where excessive communication occurs it may pay for the application developer to use message passing techniques instead of shared memory to implement the functionality. To do this a method where the parallel application can log information, and potentially accessible by the developer at run-time is required.

In this chapter the integration of the SMG DSM with an information and monitoring system is discussed. The methods by which environmental information is used by the DSM, and how application monitoring information is published are also described. In addition some of the data in the system can be used by the application, and can be used by an application developer to monitor parallel applications at run-time. Features such as debugging and deadlock detection would be of benefit to an application developer.

A more interesting use of monitoring data is to harness it to assist a developer to incrementally and selectively add performance-enhancing message passing. In theory this will allow an application written in shared memory style to be incrementally modified so that sections of the application that execute inefficiently are converted to more efficient message passing. If this can done in an directed fashion then resources can be targeted to where the greatest benefit will accrue, and can stop where diminishing returns dictate that further hybridisation is not economical.

9.1 Integrating Information & Monitoring Services with SMG

For the integration of the DSM with the information & monitoring systems, it was decided to define two separate APIs, one each for information (Listing 9.1) and monitoring (Listing 9.2), as a way (i) to insulate the DSM from the actual implementation details, and (ii) to prevent these independent tasks from being conjoined to the same system. By doing this new systems could be easily supported by re-implementing the API.
One system, R-GMA, was previously discussed in Section 5.2.1. Some of the other prevalent grid information & monitoring systems are described in Appendix 5.2. The decision to use R-GMA was based on the ease of querying and inserting data. Although the performance of the R-GMA is currently slightly less than that of MDS for query operations [148], it was chosen due to its flexible nature and querying ability. To enable further choice, an off-line *file* information system was created that reads a user-configured file containing the required information for the information provider source. In a similar manner, off-line monitoring can achieved through logging of data to normal files, and is used in situations where the overhead (memory resources) of using a monitoring system becomes too significant.
Separating both tasks into two separate APIs allowed different system implementations to be used concurrently. For example the R-GMA information source can use R-GMA and monitoring can use the file logging mechanism. The R-GMA monitoring implementation uses SMG defined schemes to publish information, while the SMG information library relies on the standard GLUE Schema [120] for information sources and a SMG schema for any SMG defined information. At compile time an application developer can link to the required implementations, allowing for exploration of different aspects of an implementation.

9.1.1 Using the R-GMA Information & Monitoring system

The integration of the DSM with R-GMA involves implementing the SMG_info API. The resulting system is depicted in Figure 9.1. All data is accessible using the consumer API, and can be viewed on-line via the R-GMA *BrowserServlet*. Although consumers are shown in all SMG processes, which are to be used to implement a feedback loop into the DSM, this functionality has yet to be exploited by the DSM. Other work has demonstrated the benefits of providing such a feedback loop [149].
A R-GMA table is defined to register a SMG application instance. Another is used to record all the process information for participants in the application. The R-GMA consumer is used to query which sites the processes belong to (the information is available via GLUE Schema information producers). When this is done, topology information can be distributed to the DSM system threads. The *file* device will return whatever topology information the user has defined; the simplest case is where all functions return a view of a one-dimensional cluster.

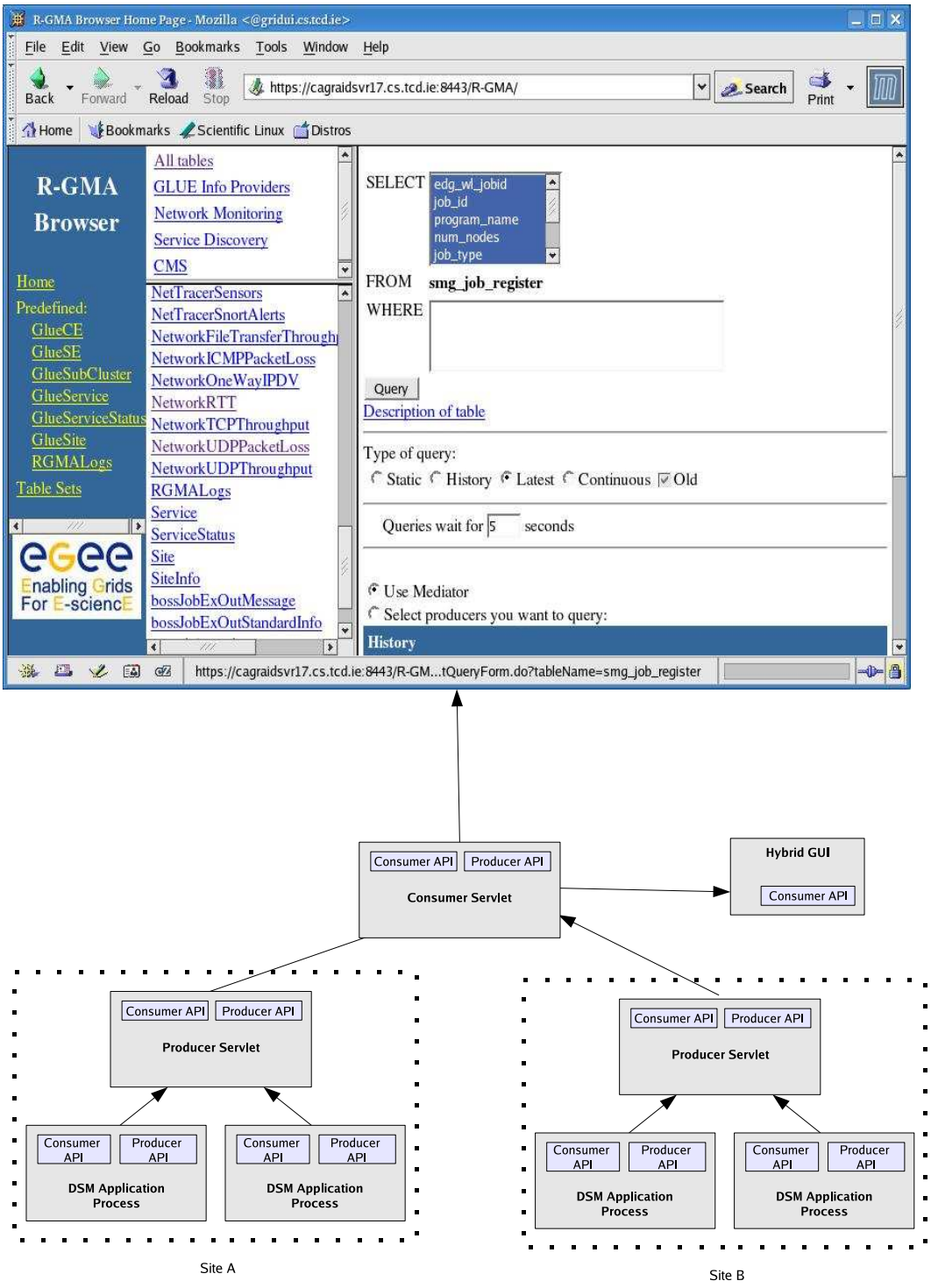


Figure 9.1: *Integrating SMG and R-GMA, and use of BrowserServlet*

9.2 Environment Information

When an application is run and use of an information system is required, all the SMG processes must first register and all application processes must block until this is done. When all have done so, it is possible for the administrative process to generate a topology tree by obtaining site locations for all processes. For this implementation a two level hierarchy is assumed (site – cluster). Other efforts such as [52] have support for a hierarchy of greater depth.
The API that must be implemented to enable the SMG DSM engine to access the information service is given in Listing 9.1. The initialisation/cleanup routines are *INFO_init* and *INFO_finalise*. *INFO_getNumSites* will return the number of distinct sites participating in the current job execution. *INFO_getSiteNum* returns the site for the given process specified by the process rank *node_rank*. *INFO_getSiteSize* will return the number of processes at a particular site that are involved in the job. *INFO_getSiteNodeList* will return a list of nodes at the site specified by the site_rank argument. The size of the array is that as returned by *INFO_getSiteSize*, and *INFO_getDefaultSiteMasterList* will return the per-site process that assumes some management responsibility if the administrative process is not local.

```
int INFO_init(int my_rank, int my_size, char *processor,
              char *jid, char *program_name, char *type,
              char *user_name);

int INFO_finalise(int exit_code)

int INFO_getNumSites()

int INFO_getSiteNum(int node_rank)

int INFO_getSiteSize(int site_rank)

int *INFO_getSiteNodeList(int site_rank)

int *INFO_getDefaultSiteMasterList()
```

Listing 9.1: *SMG Information interface*

9.2.1 Alternative Information Usage

The topology tree is used to minimise inter-site communication as all processes will be aware of the other processes in their local site. This occurs for lock, barrier, and shared memory usage. Lock synchronisation primitives requests can be queued and proxied on a per-site basis, so where multiple requests occur within a site one request originates from the site. Barriers have a per-site manager and only these managers communi-

cate among each other. MPI collective operations implemented in such a manner have demonstrated significant increases in performance when topology information is available [51, 150]. Shared memory coherence operations generate the majority of communication in a DSM. Topology awareness allows minimising the volume of inter-site coherence messages through the maintenance of per-site caches.
There are potentially many uses of the information system. In a similar manner to the benefits of two-tier fabric strategies used to keep the processors' cache memories synchronised in modern SMP systems [151], data placement efforts, such as *libnuma*, *memory affinity* and autonomic workload distribution, use dynamic data obtained from the information system. In SMG better load-balancing can be achieved by using the *SMG_work_distribution* function that allows to partition a given range *from – to* according to a given strategy which can be supported by the information system, e.g. load-balancing according to CPU power (only sites where the compute resource is homogeneous are currently supported).

```
int SMG_work_distribution(int TYPE, void *from, void *to,
        int how, void *my_from, void *my_to, void *param);
```

Although topology information is not currently exposed to the application developer it may be useful to allow an application access to it. The user has the ability to browse all monitoring and job meta-information produced by a job using the R-GMA browser (with appropriate credentials installed in the browser). This is demonstrated in Figure 9.1.

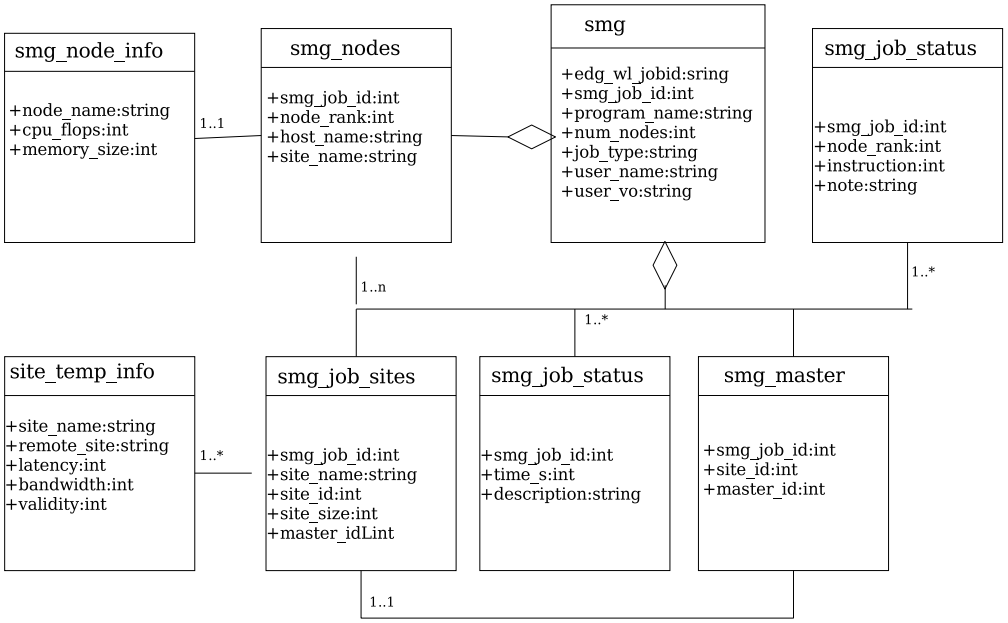


Figure 9.2: *SMG Information UML Diagram*

9.3 Monitoring Data

The monitoring system allows for the distributed logging of DSM and application runtime information. This occurs at appropriate times such as synchronisation points. Macros are used to specify the data logged within the application code as it occurs. Using the C *__FILE__* and *__LINE__* preprocessed macros the code location can be identified. The API for the monitoring consists only of initialisation, cleanup, and the following simple calls to log information.

```
int log_Init_l(int my_rank, int my_size, int job_identifier,
               char *job_string, FILE *debug, int flags);

void log_Finalise();

void log_Event4(char *Type, int param\_1, int param\_2,
               char *param\_3);

void log_Event9(char *Type, int param\_1, int param\_2,
               int param\_3, int param\_4, int param\_5,
               int param\_6, int param\_7, int param\_8,
               char *param\_9);

void user_application_log(int type, char *event, int param_1,
               char *param_2);
```

Listing 9.2: *SMG Monitoring interface*

The generic format of a logging message is given below. An example of usage is also shown. The definition of the logging message is independent and is defined on a per message type basis. In the R-GMA implementation a SMG monitoring schema/table was defined. All processes in the application produce information according to this definition via the producer API, using continuous producers on the back end.

TYPE, Argument_0,..., Argument_N, Code_Line, Source_File
BARRIER, 1, 2, -1, -1, 0, 0, 0, 0, **23**, **basic.c**

The monitoring information produced in the R-GMA implementation is accessible through the consumer API. The overhead in accessing the monitoring system results in a performance penalty when logging data. The R-GMA implementation can use a separate thread to perform all logging operations; a log method will pass the parameters to the logging thread thereby ensuring the user/DSM-system threads are not I/O bound. The *file* device will log information to per process files.

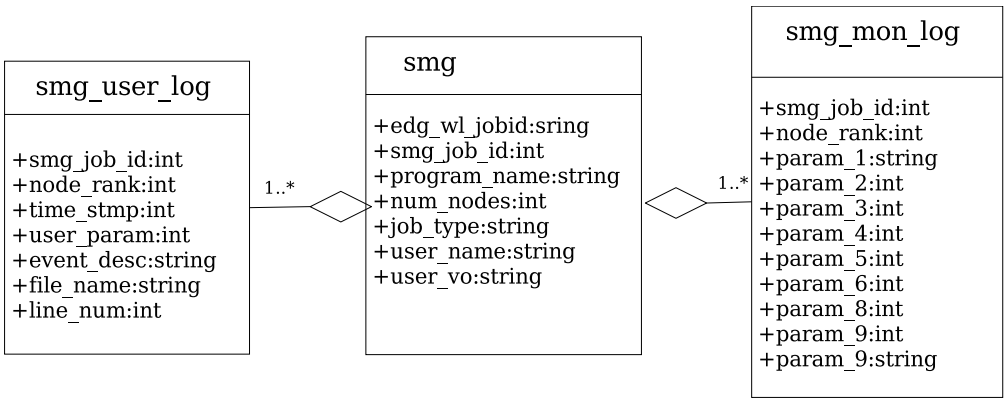


Figure 9.3: *SMG Monitoring UML Diagram*

9.3.1 Alternative Monitoring Usage

The file monitoring implementation will log monitoring information into a per-process file. The information is logged at suitable intervals, and no provision exists for access to these files. The user will need to interpret the data. When the R-GMA implementation is used, the user is only able to query events/data that is of interest, since vast quantities of monitoring data can be produced. SQL *SELECT* queries are generated against the available information as a data filter.
A front-end viewer is available to present the information to a developer in a categorised fashion. Screen-shots of this viewer appear in Section 10.3. The viewer includes the ability to highlight the relevant sections of user-application code, and so one can view system information relating to an instance of a global barrier and view the corresponding code.
Similar work has been reported in the literature that uses monitoring information to obtain optimisations such as: speculative parallelisation of loops [125]; dynamic scheduling of nested parallel loops [126]; and traditional OpenMP/MPI hybrid code [152]. Development tools such as Marmot [153] have also been used to verify MPI applications and to identify situations such as deadlock. Other work enabled the monitoring of MPI applications using R-GMA [122].
The ability for user code to access monitoring events in an application is not usually available (logging to a file may not be possible due to access restrictions on the directory). This can be a frustrating experience in a user application that may execute for a long duration. The function *SMG_user_event* can be used within application code. To access the information during application execution a corresponding querying mechanism is required, e.g. with the R-GMA approach the monitoring data can be accessed via a R-GMA Consumer, however, a general API call is supplied (*SMG_get_events*).

int **SMG_user_event**(char *user_tag, int lineno, char *filename);
int **SMG_get_events**(char *job_identifier, int flags, int param);

9.4 Hybridising Parallel applications

In Chapter 4 it was seen that a DSM will, in general, transmit more data than is required in the course of a job execution than a similar job implemented in a message passing style due to DSM engine overhead. This additional communication has the potential for significantly reducing performance if the DSM is unable to effectively share data due to the excessive communication use. The main source of this problem was discussed in Section 4.6, i.e. coherence protocols are developed for the general case and can react very differently when dealing with unanticipated sharing patterns.
However, the code responsible for this poor performance is very localised and derives its impact from the DSM's lack of ability to adapt to the sharing pattern of the application. If the application developer can dictate the communication usage by implementing the data sharing functionality with message passing, then superfluous data will no longer be transformed. As this modification only needs to be done in a localised fashion the two programming models can coexist in a hybrid fashion, hence the term hybridisation.
Many applications need to be scaled to new levels, some of these involve simple communication patterns at global barriers, but as an application scales the coherence communication in particular may grow commensurately, and may saturate the communication link. This may occur in the barrier usage scenario depicted in Figure 8.3, the application with a problem set at a given scale may show good performance, but the same application at a larger scale may not as the additional coherence data may exceed the available bandwidth, and superfluous data transfer may occur.
Problems with EC memory and lock usage should not occur, otherwise the resolution of sharing and lock binding needs to be investigated. For the vast majority of parallel applications the candidate code fragments suitable for hybridisation are relatively few in number. The hybrid programming concept is not new. Shared memory and MPI hybrids have shown that good potential exists [40, 5] (in most situations MPI is used for simple data transfer), while Version 2 of the HPF language provides the Extrinsic directive [35] specifically to enable hybrid programming.
However what is new in this thesis is that the application of the message passing style is now the exception, since the DSM is now responsible for all but these few hybridised code fragments. It is desirable to reduce the DSM overhead associated with the fundamental DSM concepts, namely write trapping, write collection, consistency, and coherence. In general the consistency/coherence combination is a best effort; implementing more combinations will support a greater range of data sharing patterns.
There are efforts elsewhere that take a similar approach: profiling parallel applications, analysing the resultant data, providing feedback to the user. These systems [154, 155], take a more holistic view compared to the restricted view that SMG takes. They provide probes for different parallel programming models (OpenMP, Message-passing). The most substantial of these systems, TAU, provides tools for managing experiments (PerfDMP), a component for enabling artificial intelligence (PerfExplorer) to identify code regions, and a sophisticated user interface (ParaProf) to aid the developer. Tau has its own logging format which currently prevents it from analysing SMG monitoring data.

9.4.1 Performing Hybridisation

For this thesis hybridisation is performed by the developer; it is possible that this process will be performed by a compiler, however, given the difficultly in generating message passing code from automatic parallel compilers [156], this may prove difficult. The basic process is depicted in Figure 9.4. First the candidate (actual shared memory region and code location) responsible for the poor performance is somehow identified to the developer, see Section 9.5.

The methodology in implementing the hybridisation will require the developer to write a valid message passing sequencing. This requires that the destination of send, and source of receive, operations be known. In this respect, the hybridising engine can only make a best guess as these variables are based on the usage pattern of data.

The developer will begin by disabling the implicit sharing of the shared memory. Next, the shared data locations may be modified using the required message passing calls, with the source and/or destination buffers of messaging primitives possibly being the shared memory regions. Once the modifications/message passing are complete the modified regions must be explicitly validated/invalidated before sharing can recommence. This allows for the different local changes to be resolved globally before the next relevant coherence event is demanded by the consistency model, i.e. the shared regions must be made globally consistent after message passing is used.

The actual candidates for hybridisation are determined by processing monitoring data information streams, which can either be done on-line or off-line, depending on the monitoring system used. The user criteria will also determine what recommendations are presented to the user, but in the majority of cases the criteria will be to maximise performance through the minimisation of coherence communication and to a small extent DSM overhead.

Appendix E shows a benchmark application, in both the original SMG (page 292) and hybrid (page 296) versions.

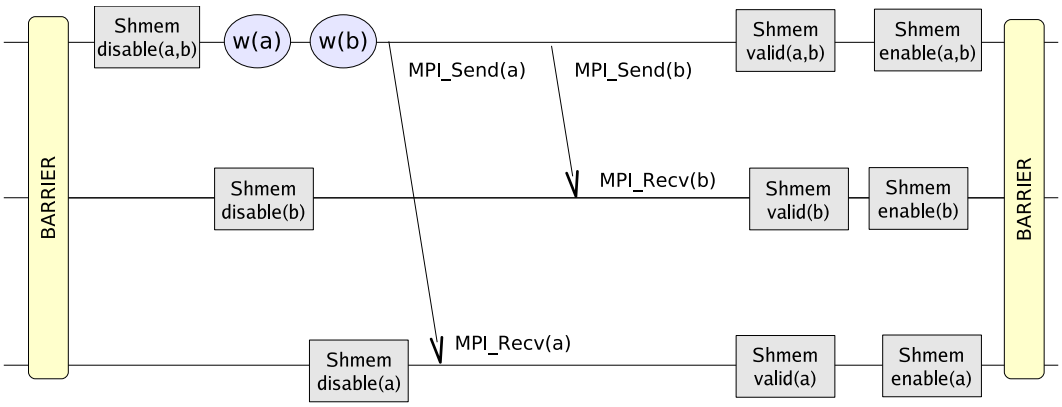


Figure 9.4: *SMG and Hybrid use*

9.4.2 Required SMG API enhancements

Allowing the user to introduce hybrid constructs at various locations requires support in the DSM. The DSM must be made aware that the user is modifying shared data, so the DSM system's control must be temporarily suspended. When finished the DSM control must be re-enabled. This is achieved using the API commands listed below. *SMG_shmem_disable* will turn off DSM control on a particular shared memory region, and *SMG_shmem_enable* will in turn on DSM control for a given shared memory region.

```
int SMG_shmem_disable()
int SMG_shmem_enable()
```

When control is disabled, subsequent DSM actions should be minimal. The disable action will disable the DSM page faulting, turn off the visibility at consistency points, and make a complete twin of the object for write collection when the DSM is re-enabled. Before control can be returned to the DSM the shared memory must be re-qualified globally by validating or invalidating. This ensures that no two process spaces have modifications to the same locations in the shared memory region. If this was not the case then a write clash would occur upon the following coherence action. The SMG commands to achieve this are *SMG_shmem_validate* to validate a section of the shared memory region, and *SMG_shmem_invalidate* is a complementary function that invalidates (undo) any modifications to parts of the shared memory region.

```
int SMG_shmem_validate()
int SMG_shmem_invalidate()
```

To enable the developer to construct hybrid applications a message passing infrastructure needs to be available. Rather than requiring the developer to establish this, it is sensible to allow the use of the (message passing) communication infrastructure already created by the DSM. A SMG API call is provided, *SMG_get_comm_handle*, to enable the developer to gain this access to the message passing layer. This will return a handle enabling the developer to use the underlying message passing system. Obviously this handle definition will be dependent on the communication layer itself. In the case of the SMG MPI communication module this would provide a MPI communicator. The user is not permitted to use the default MPI_COMM_WORLD handle, as the process identifiers used by the DSM and the user's message-passing code should correspond. The request for a communicator must be through the DSM. The user's hybrid message passing code cannot use the same MPI communicator as the DSM communication implementation (the DSM communication thread currently filters processes all incoming messages on this communicator) so a duplicate of the DSM's communicator is made.

```
int SMG_get_comm_handle(void *comm_t);
```

9.5 Hybridisation Identification

Ultimately the motivation for hybridising an application will be to increase performance of a parallel application by decreasing the overhead components in Amdahl's equation (Equation 2.4), and thus (hopefully) decrease the execution time. Hence a baseline execution performance profile is required to compare the performance against. This may be a serial, MPI, or an execution of a DSM version of the application, but at a lesser scale. Alternatively the developer may choose just to eliminate the potential areas of code responsible for poor performance. All parallel applications will exhibit some potential candidates for hybridisation. These areas/shared-memory regions that are responsible must be mapped to the locations in the application code.
The pertinent question for the developer is whether it is worthwhile to justify hybridisation The hybridisation engine seeks to identify these regions. It works by processing the monitoring events generated; the format of these events were highlighted in Section 9.3. The methodology is that all the monitoring data is viewed as a virtual database and the engine runs queries on the data based on the requirements. The hybridisation engine is built to process the events in an agnostic manner, no matter where the source, so an on-line monitoring system (such as R-GMA) or offline source (such as the flat-file approach) can be used, but the user must specify whether the source is off-line or on-line. User-denoted events can be specified using the call *SMG_user_tag*, to aid the developer in establishing the performance of the system. Obviously this requires monitoring system to be enabled in the application.

int **SMG_user_tag**(char *user_tag, int lineno, char *filename);

9.5.1 Rating Hybridisation Candidates

The policy of the hybridisation engine is currently to seek to identify those regions that contribute to the majority of inter-node communication. Use of a DSM results in message generation. This is the source of problems that hybridisation will try to remedy. Depending on the DSM characteristics this can mean minimising the volume of messages or the message payload size. The candidates for identification are based primarily on the interval release-acquire pairs (i.e. synchronisation variables). As synchronisation variables can be used repeatedly throughout the application the use of variables must be mapped to the locations of code where they were used. These areas can be a particular line in the application source code, a function, or a region of code that is delineated by the acquire and release phase of synchronisation operations (that were described in Sections 2.4). A barrier interval is uniquely identifiable via two barrier events. Each barrier event is uniquely recognisable using the file and the source code line number where the barrier is called and the barrier timestamp.
The candidate metric can be a function of message transfer per (LOC), per object, per synchronisation primitive, or the interval duration. At present there is no automatic or user-defined ranking policy. Instead, the candidate metric is presented visually as a pie-chart, see Figure 9.5.

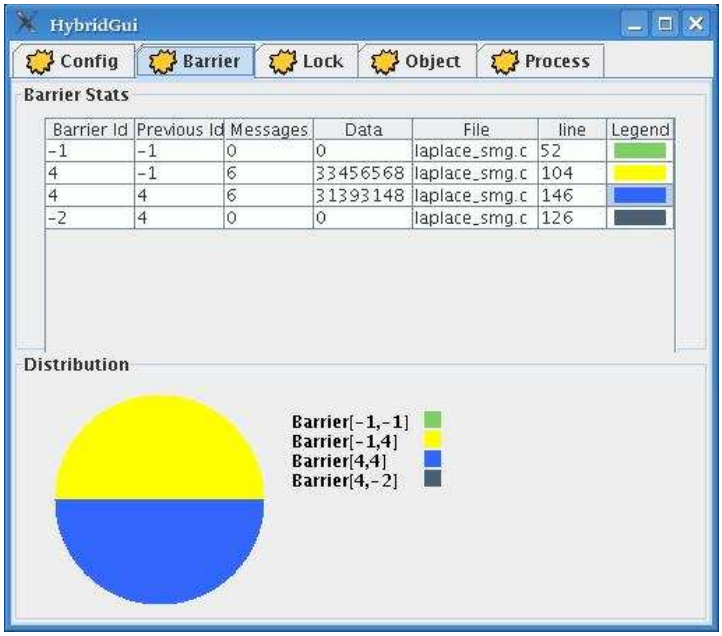


Figure 9.5: *Hybridisation GUI*

9.5.2 Hybridisation GUI

The hybridisation GUI works by directing the user to the source code locations where the majority of communication occurs, and also to the shared variables that are responsible for the communication. The code browser illustrates the 'hot-spots' in the application and the objects responsible. This allows for incremental and localised optimisations. The participating processes are also identified.

The developer can examine the interval between synchronisation operations, such as between barriers, and view the volume of data transfers generated between processes. From other information gathered during the interval they can identify the memory objects causing the communication. The amount of communication between barriers in a matrix multiplication example is clearly illustrated in the monitoring interface in Figure 9.5, and the the actual application code that is responsible can be identified as in Figure 9.6. From other information gathered during the interval we can identify the memory objects causing the communication. The hybridised code resulting from the identification process is given in Appendix F, page 296.

Other features include the highlighting of fragmented shared memory use which may be the result of poor algorithm design or implementation. This can be identified when a shared memory region is repeatedly modified by multiple processes in a fragmented fashion. Such a scenario may occur when a developer, unaware of the potential differences between C and FORTRAN languages (row and column ordering), transposes a matrix multiplication algorithm.

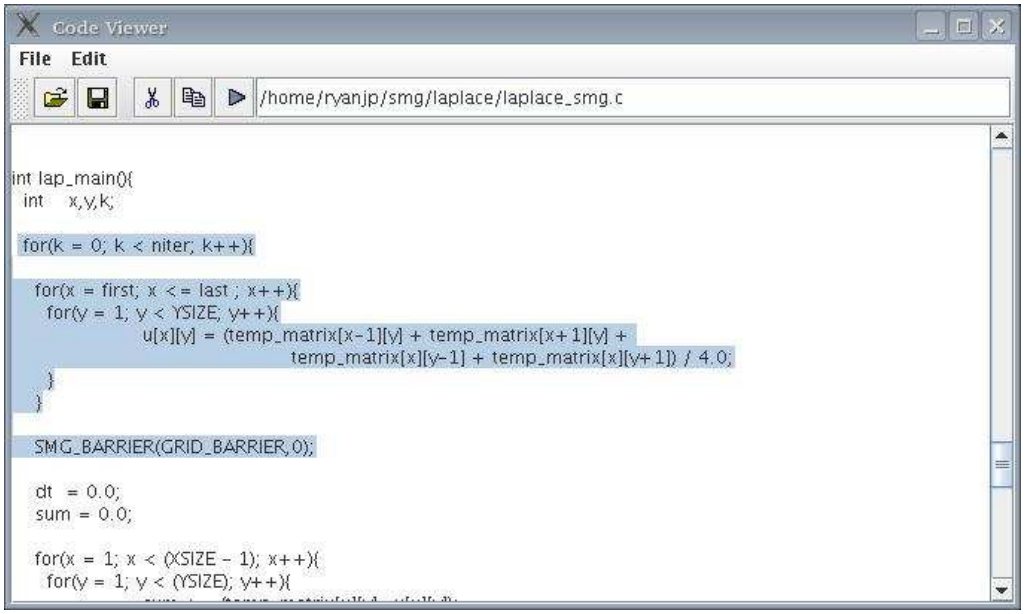


Figure 9.6: *Hybrid Identification*

9.6 Incremental Hybridisation

Incremental hybridisation is an extension of the hybridisation process that allows for incremental optimisations in an application through the use of incrementally adding hybrid programming measures. The hybridisation process described in Section 9.1 can be repeated until a given tolerance is reached, dictated by the law of diminishing returns, where it is no longer sensible to optimise further as no significant (worthwhile) benefits are obtainable.

9.6.1 Version control system

Incremental hybridisation implicitly demands a facility to compare the different versions of application code, and the resulting data. The hybridisation engine makes use of a Version Control System (VCS) to maintain different versions of an application that has undergone hybridisation, thus allowing for backtracking as needed. In addition, it stores the execution data of the applications between executions and a condensed form the the monitoring data (it may be unsuitable/unsustainable in many cases to keep large amounts of monitoring data). Currently SMG does not allow developers to maintain separate VCS repositories for application and monitoring data.
Future incremental hybridisation may benefit from the ending working in tandem with the source code control framework as illustrated in Figure 9.7, so that it could allow *semi-automatic hybridisation* perhaps guided by predicate logic. All previous application runs could be analysed calculating the benefits of hybridisation.

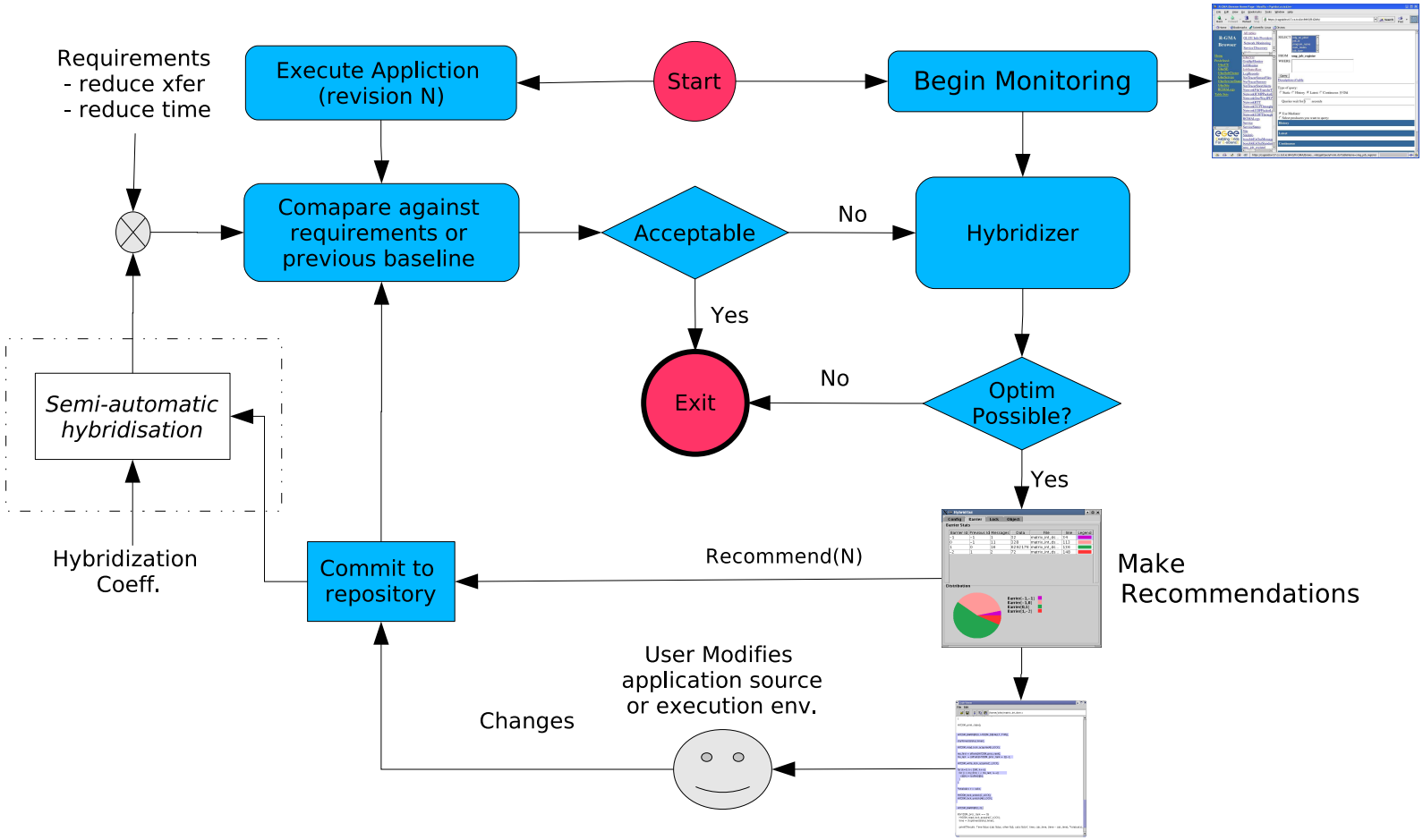


Figure 9.7: *General steps involved the Incremental hybridisation*

9.7 Implementation Issues

R-GMA was chosen for the information & monitoring system as it offers greater flexibility in the accessing of information, however, many issues arose from features of this system.

- The run-time monitoring system (enabled by R-GMA) can only handle a limited amount of events ($\tilde{8}$0,000) events at any given time. To enable more events within a given period then an offline monitoring system is required, with logging to files during the execution of the application.
- Additionally, R-GMA imposes a limit of 256 characters on the length of a string variable. While this is not a problem in the majority of cases it does restrict the length of the file paths and the developer's ability in the 'tagging' of applications.
- All processes must register with the information system. The master process will poll the system waiting for this to occur. Start-up messages are sent to all processes once notification has been received and the topology map has been generated. This results in very poor start-up times.
- The use of the R-GMA information system required the use on non-GLUE-compliant tables.
- Currently the R-GMA C APIs are in a state of flux so the most recent releases have not been used.
- The impact of logging system information necessitated the use of a thread to perform this task (it takes a relatively long time to insert data into R-GMA). As a separate thread is required for the DSM system an additional thread will steal resources, so it must be run at a lower priority.
- The front-end 'hybridisation' tool is limited in its functionality. It can only perform set queries on the monitoring data it obtains.
- The hybridisation tool is also restricted in its ability to identify potential areas, or the interval, of code for hybridisation to within one file. It should be noted that this is a typical case of OpenMP applications where parallel regions must be self-contained in a single lexical context.
- The hybridisation tool will have problems resolving the correct code locations to the developer where a translator has converted OpenMP code to SMG DSM code.

CHAPTER 10

Evaluation

The primary goal for this thesis was to explore the use of the shared memory programming model with grids. To achieve this successfully, communications must be efficiently and sparingly used. The mechanisms developed in the course of this thesis are primarily concerned with achieving this while contributing little additional overhead. Below some analysis and evaluation is performed. Three main contributions are evaluated: the subscription protocol, hybridisation, and providing topology information to enable efficient synchronisation operations while operating in a grid environment. The evaluation objectives can be categorised under four main headings:

1. Evaluation of the SMG DSM as an execution model compared with MPI.
2. Evaluation of the hybridisation approach, where more efficient message-passing mechanisms can incrementally replace DSM actions when circumstances dictate that performance improvements can be obtained. The process of incremental hybridisation is also explored.
3. Evaluation of the SMG DSM with and without a grid information system.
4. The evaluation of SMG on a grid with emulation of the overhead of inter-site latencies versus SMG on a cluster of identical compute resources.

A number of applications have been used. While some have been unsuitable for one feature they have proven suitable to evaluate another. The metrics employed are overall performance and scalability (through efficient use of communication resources).

Note: All graphs that feature the number of processors in an application are presented in a logarithmic scale, Log_2(No. Processes), see Figure 10.1.

10.1 Experimental Methodology

The DSM components are represented in Figure 10.2. The DSM communication uses the same message-passing libraries as the MPI version of the test applications. An assortment of applications has been taken and versions constructed for all the features that needed to be tested. In this section the applications used are described and why they were chosen. The execution environment is described. As cross-site multi-threaded MPI is not available at present, the grid is simulated. The interesting aspects like page-faulting, page protection, write trapping/ write collection strategies, etc., are all validly tested by the simulations.

10.1.1 Test applications

Test applications were chosen that range from those that have no real basis in parallel computing to those that are embarrassingly parallel. Marinov et al [157] list characteristics of DSM applications. The test applications are routines of real value themselves but are examples that exhibit the temporal/spatial memory sharing patterns of 'real' applications [158].
While this chapter adopts a light touch regarding presentation of results, these are the distillation from approximately 8700 (cumulative) wall-clock hours of high performance computing experiments. The applications used to evaluate SMG are: Embarrassingly Parallel (EP), Conjugate Gradient (CG), Fast Fourier Transform (FFT), and Integer Sort (IS) from the NAS benchmarks [159, 160]; Barnes-Hut from the Splash Benchmarks [161]; and the common Matrix, SOR, Laplace, TSP, and Gauss benchmarks that implement well known routines. EP and Matrix were used to benchmark the overhead of the SMG DSM engine, while nearest neighbour applications like SOR and Laplace were used to evaluate the sharing performance and communication efficiency of the DSM. IS, Gauss, and Barnes-Hut demonstrate the use of different synchronisation primitives. FFT demonstrates the ability to construct an application with irregular access patterns that SMG finds difficult to handle. Other applications have been ported to SMG and MPI but were eventually deemed unsuitable for further evaluation for a number of reasons, mostly that their results were too similar to the chosen applications to be relevant, e.g. Jacobi is in the class of nearest neighbour applications.
A number of versions of each application were required for evaluating different comparisons. These versions included MPI, SMG using the update protocol, SMG using the Subscription protocol, SMG using multiple user threads, SMG-MPI hybrid, and SMG for a Grid with/without access to an information system.
The various characteristic types of shared memory accesses that occur, described by [34], are discussed in Section 4.1. The SPLASH benchmarks have some technical considerations [161] when ported to non-shared memory platforms. Lu's Thesis [162] identifies the problems with comparing a message passing version (PVM) of some of the implementation of the SPLASH benchmarks against a shared-memory style (Treadmarks DSM), i.e. some of them are just not suitable to DSM because of the excessive communication to computation ratio. Other work has demonstrated the difficulties in running NPB over

DSM (Scash, OdinMP)[163]. Unfortunately there are no credible benchmarks that have been explicitly developed for benchmarking the use of computational grids. There are grid applications that portend to do this [164], but really don't (e.g. work-flow using NAS benchmarks). Where the MPI programs were developed for this thesis, they were constrained to use the same algorithm employed by the DSM versions, so they might not be the most efficient implementation for the available resources.
Only the results for EP, Matrix, Laplace and SOR are presented here, as the others simply confirm the same salient results for these four applications. For these four applications, the term P used below refers to the number of processes, and N refers to the dimension of the application. Where MPI collective operations are required the effective message count is taken to be equal to $2 \times (P-1)$ in the calculation of the total amount of messages generated.

Embarrassingly Parallel

The Embarrassingly Parallel (EP) benchmark is part of the NAS suite and generates pairs of Gaussian random deviates. This benchmark is representative of many Monte-Carlo style simulation applications in that it is heavily computation-intensive with little communication/synchronisation. This application was chosen as it is useful in establishing a base reference for the computational capacity of the system.
It can be readily seen that the parallel processes can be independently generated. The program generates $n\pi/4$ Gaussian pairs per process. The only communication is the gathering of an array of results from the processes that is done at the end by the root process. This only point of communication is between the root process and all other processes and only involves (P-1) messages in total.
The SMG implementation allocates a small shared array where the resultant data is deposited by each process. The MPI implementation uses a gather operation to sum the partial results of all processes. The total data communicated with both is approximately (P -1) * N. For all test scenarios it was assumed the value N = 36 (the maximum for the NAS application).

Matrix

It is only natural that the matrix multiplication example that was used in Section 2.1 to demonstrate the benefits of parallel computing be used as one of the evaluation applications. The application multiplies two dense matrices A and B, with dimensions $N \times M$ and $M \times P$, to produce a resultant matrix C = AB with dimensions $N \times P$. As previously discussed this is an easily parallelised application, where each thread of execution computes a subset of the resultant matrix.
Like the EP benchmark, matrix multiplication is of the embarrassingly parallel class. It has been seen already that a naive implementation of Equation 10.1, matrix multiplication of two square matrices with dimension N, has computation that is $O(N^3)$, while communication is $O(N^2)$, so for relatively large values of N this is an ideal application for parallelisation.

In the DSM version all of the matrices are allocated as a single shared region. There are two communication regions where both MPI and DSM versions have a root process that initialises the incident matrices A and B, and distributes them to other processes for computation. When computation is finished all computed portions of C are gathered by the root process. Square matrices are assumed with dimension 6144. For the integer matrix multiply version the storage requirement for the three matrices is 432MiB, although a floating-point version is also available (given the same dimensions the storage requirement becomes 860MiB).

$$C_{ij} = \sum_{n=1}^{m} a_{in} b_{nj} \tag{10.1}$$

Laplace

Laplace is a simple iterative stencil application that implements an algorithm for a stripped-down version of the Jacobi transformation method of matrix diagonalization for approximating the solution to a linear system of equations. During each iteration each element is updated based on the values of its nearest neighbours (usually the average, i.e. Equation 10.2), with the boundary values remaining fixed. The Laplace application, although fundamentally solving the same problem as SOR, uses a different algorithm resulting in half the number global barrier calls and a commensurate number of messages. As Laplace converges more slowly than SOR, with each grid point changing slowly with each iteration, extremely large volumes of data are transmitted. Iterative schemes of this type require time to achieve sufficient accuracy and are reserved for large systems of equations where there are a majority of zero elements in the matrix. The implementation of the algorithm assumes the contrary, with many of the elements being initialised to non-zero values.
The computation complexity is $O(N^2)$ with communication $O(N)$ for an efficient implementation (such as the MPI implementation). However the DSM versions can generate traffic of $O(N^2)$. [58] derives a metric from the equations expressed in Section 3.3 for obtaining the potential speedup for this class of stencil operations in a Grid environment. For computation involving processors distributed across grid sites the dimensions would need to be very substantial; this magnitude is not supported in SMG at present due to the limitations with the virtual address space size.
The DSM application is implemented using barrier synchronisation primitives with a bound shared memory region for the grid. Like the SOR application below, the dimensions for the problem size are 6144 X 6144, with each element being a double-precision floating point value, giving a shared region requirement of 288MiB when the application is implemented using a naive approach with a single shared memory region for the whole application grid. This involves all modifications to the whole shared array being transferred among among all processes at the end of each iteration. For the MPI version only the region to be processed and the neighbouring strips need to be transferred between the processes.

$$New_A[i][j] = (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \ / \ 4 \qquad (10.2)$$

SOR

Successive-Over-Relaxation (SOR) is a simple iterative relaxation algorithm. It is one of the numerical methods for solving partial differential equations. The equations are represented discretely using a two dimensional array. During each iteration each element of the array is updated as a function of its nearest neighbours and a given relaxation parameter, omega, typically as defined by Equation 10.3. The implementation for this thesis uses the standard red-black approach to prevent a node overwriting a value before it is accessed by a neighbour. Each iteration is divided in two, with alternate elements updated in each half, i.e. there are two synchronisations per iteration. The number of iterations can be fixed or variable, dependent upon a stopping condition that is usually specified by the user.
Like the Laplace application (which is from the same preconditioner class of applications) the complexity of the algorithm for a system of size N is a fairly modest $O(N^2)$, so it makes a poor candidate for parallelisation as there is considerable overhead (no matter what the implementation) in distributing the initial matrix and communicating partial results. The communication requirement is $O(N^2)$, as with every iteration all neighbouring values must be exchanged between processes.
In the DSM versions barriers are used to synchronise the processes, while the MPI versions use blocking send/receive pairs. The input data for the application is a square matrix with dimensions 6144 X 6144, giving a shared memory region of 288MiB (larger sizes are possible, but the value of N = 6144 was chosen to be consistent with other applications). For the SMG version all processes allocate the shared region locally. For the MPI version only the region to be processed and the neighbouring strips need to be transferred between the processes. SMG allocates a single shared memory region of size (X * Y * sizeof(value)).

$$\begin{aligned} New_A[i][j] \ = \ & ((A[i-1][j] \ + \ A[i+1][j] \ + \ A[i][j-1] \ + \ A[i][j+1]) \\ & * \ omega \ * \ 0.25) \ + \ (1 - omega) \ * \ A[i][j] \end{aligned} \qquad (10.3)$$

10.1.2 Testbed Description

The basic characteristics of the machines used are given in Table 10.1. Basic machine metrics were obtained using lmbench [165]. The systems were the *Moloch* and *IITAC* clusters from the Trinity Centre for High-Performance Computing (TCHPC) [1], and the Walton cluster from the Irish Centre for High-End Computing (ICHEC) [2]. As of writing, the latter two occupy positions No.345 and No.367 respectively of the *Top 500 List* [166]. Walton is not relevant to the the four applications discussed here, so will not be mentioned further. All machines run a version of Linux that is compatible with

the needs of SMG, i.e. kernel version 2.4.5 or later. The salient attributes regarding OS operations (i.e. system calls used extensively in SMG like memory mapping/protection) and *pthread* functions for each system are given in Appendix A, while the physical specifications are given in Table 10.1.
The SMG applications use the exact same message passing library as that used for the MPI applications, i.e. MPICH2-1.0.4, using the *ch3:sock* communication device. It must be noted that while the more efficient Infiniband Interconnect is available on the IITAC cluster, it was not used as the MPI distribution MVAPICH2 [167] (which is based on the MPICH2 distribution) has until recently not supported multi-threaded MPI applications; the eventual support came too late for the use of this interconnect to be considered. As the Gigabit Ethernet is only intended for management of the cluster, the bandwidth and latency figures (see Appendix A, page 191) are much lower that what one would expect from this interconnect.

Attribute	Moloch	IITAC
Num. Nodes	65	346
Node Type	IBM x335	IBM e326
Num. CPU	2	2
CPU Model	Intel Xeon 3.06 Ghz	AMD Opteron 250 2.4 Ghz
L1 Cache	16Kb I, 16Kb D	64Kb I, 64K D
L2 Cache	512Kb	1024Kb
Memory	2GB 400Mhz DDR	4GB DDR PC3200
Interconnect	2 X Gbit Ethernet	2 X Gbit Ethernet + 10Gbit InfiniBand (IB)

Table 10.1: *Infrastructure attributes*

The latency and bandwidth metrics for communications between nodes in the systems are given in Appendix A. These messaging costs are inclusive of the overhead associated with the MPI implementation.

Grid Simulation

There is no grid-enabled MPI implementation that currently supports multi-threaded applications (a requirement of SMG). All the testing therefore employed a non-grid flavour of MPI. The SMG DSM was tested in a virtual grid environment, with the number of sites configurable at run-time through file-based information (the systems used do not have the required software for use with the R-GMA information system).

- **Single-Site:** for single site MPI configuration, where there are no benefits from the information component, jobs could nonetheless be submitted that access the information & monitoring system. Applications were able to make use of the file based monitoring component for logging information.

- **Multi-Site:** as the available grid infrastructure did not support cross-site multi-threaded MPI, it was simulated using the same systems outlined above. This was achieved through the simulation of the characteristics (i.e. latency and bandwidth) of the inter-site communication links for a hypothetical grid consisting of four sites. It would have been better to have obtained valid data from the grid information system, but alas there was no grid information system in the simulation, so the latencies between sites were simulated using data obtained using the tools at `www.hea.net`. The figures for bandwidth (Table 10.2) were taken from the *MRTG* tool, while latencies (given in Table 10.3) were obtained using the *Looking glass* tool. The actual figure for estimated bandwidth was derived from the differences between the stated figures for the maximum bandwidth obtainable and the maximum bandwidth utilised in a month. The bandwidth between two sites was determined to be the minimum of the estimated bandwidths of the two sites.
  These values were fed into the information system implementation, and this in turn supplied these values to a modified version of the MPI implementation of the SMG communications API (see Listing 6.1) that provided the simulation of delays between grid sites.
  For multi-site grid simulations the file-backed approach was used to provide user-specified information such as site topology (this allowed logical partitioning of one of the systems described above into different sites), site information (memory, CPU power) and the site interconnect network information.

The information was accessed in any of these scenarios using the same defined SMG API (see Listing 9.1).
The system performance measurements are not representative, since the DSM communication was unable to avail of blocking receive MPI calls. However, the significant memory overhead associated with the DSM (update coherence) persists, as well as the substantial processing and memory requirements for consistency actions (twinning/diffing). This can be overcome at the expense of increased coherence message volumes.

	TCD	**UCC**	**UL**	**UCG**
TCD	-	6.03	11.65	11.65
UCC		-	6.03	6.03
UL			-	13.66
UCG				-

Table 10.2: *Simulated Inter-site Communication Bandwidth (MB/s)*

	TCD	**UCC**	**UL**	**UCG**
TCD	-	8.200	4.000	7.400
UCC		-	10.701	13.025
UL			-	9.600
UCG				-

Table 10.3: *Simulated Inter-site Communication Latency (ms)*

10.1.3 SMG Metrics

The DSM components for evaluation are depicted in Figure 10.2. The attributes of the DSM that are of most interest are the consistency and coherence combinations and to a lesser degree the DSM response times and the time to perform write collection on an object. For this thesis only one consistency model is available, entry (EC), while there are two choices for the coherency model, update (UP) and subscription (SUBsc).

DSM invocation

Table 10.4 gives the response times for a range of rudimentary DSM operations (obtained using lmbench) such as the local DSM and remote DSM requests (the value stated for the remote request time is that between two machines for the given cluster) that measure the time for the DSM engine to respond to a user generated request. The times for the remote operations are a reflection of the interconnect latency for both clusters.

DSM write collection

The time for write collection when the diff creation method is used is that indicated in Table 10.4 as *Diff Creation*, assuming a 4KiB page with a four integers modified; the alternative 'raw' write-collection method only takes the time to copy a complete memory page (i.e. 0.46μs). The time to apply the resulting diff collection is stated as the *Diff Application* time. The stated time for a pagefault is the time between the fault occurring and the DSM system first responding, i.e. minimum overhead, and not the time for the SMG pagefault handler to execute as this time will vary depending on the shared memory object's consistency, coherency, and write detection method.

SMG barrier synchronisation primitive

The performance of the SMG_Barrier compared to some MPI collective routines is illustrated in Figure 10.1. These results were obtained using a simple benchmark which measures the time to perform a given number of collective operations. The operations for MPI are the *MPI_Barrier* and *MPI_Alltoall* operations. On first inspection it would appear the performance of the SMG is significantly worse; this is misleading as the SMG barrier implements a specific algorithm to support memory coherence (as mentioned in Chapter 8 latencies are propagated along the barrier tree), while the MPI implementation can use an algorithm such as Butterfly or Combining Tree that doesn't have to support situations where no data is distributed among processes. The MPI_Alltoall function may for some implementations enable better performance at the expense of scalability.

Operation	Moloch	IITAC
Pagefault	7us	6us
Page Protection (none)	0.9764us	0.613900
Page Protection (r/w)	6.053900	1.948600
Twin Copy (page)	0.46us	0.46us
Diff Creation (page)	12us	19us
Diff Application	1us	2us
Local DSM Request	26us	25us
Remote DSM Request	413us	228us
Simple syscall	0.3838us	0.0748us
Simple read	0.6469us	0.2126us
Simple write	0.5782us	0.2120us
Signal handler installation	0.8627us	0.1719us
Signal handler overhead	2.4455us	1.3465us
Protection fault	0.7772us	0.1915us
Process fork+exit	216.5385us	135.3us
Mmap read bandwidth (4KB)	11,264.92MB/s	19,947.68MB/s
Memory bzero (4KB)	11,419.04MB/s	30,515.50MB/s
Select (100 fd)	19.3194us	10.3446us

Table 10.4: *Basic operation costs*

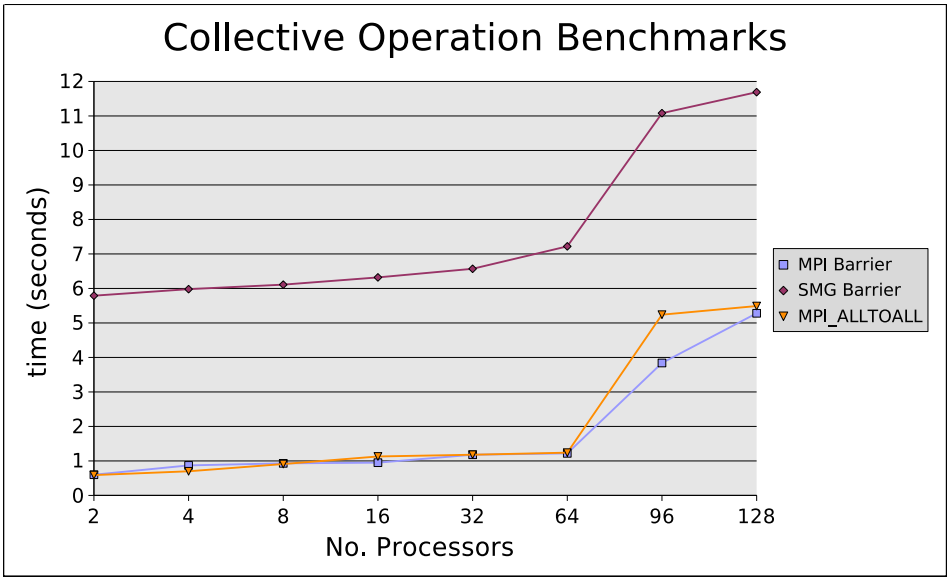


Figure 10.1: *Performance of different collective operations*

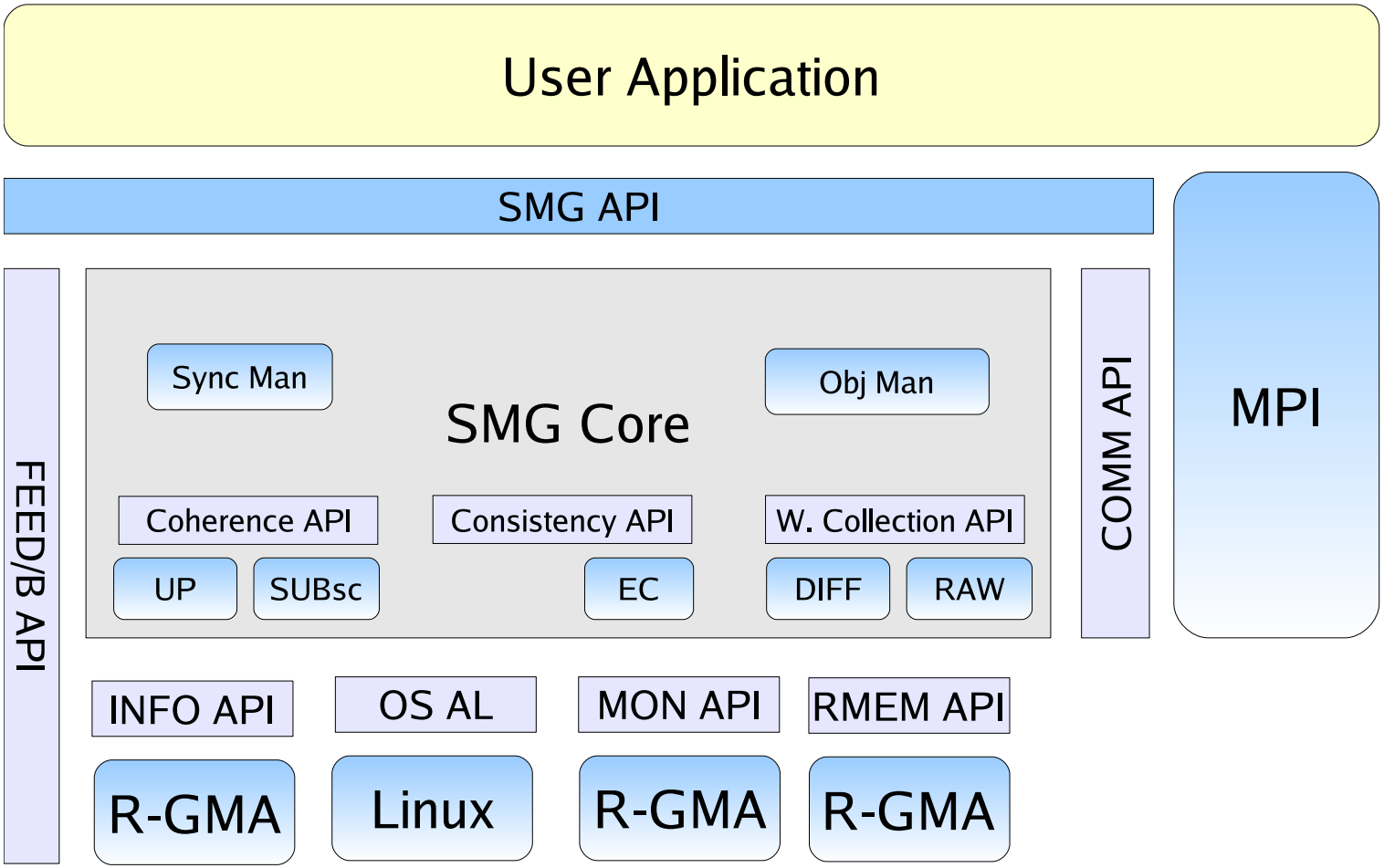


Figure 10.2: *SMG System Block Diagram*

10.2 Performance of SMG versus MPI

For all the applications, an MPI version of the application was executed, and also the SMG variants that included the update protocol, the subscription protocol, and multi-threading. In all cases, for the same application, the tests were executed on the same cluster. The nearest neighbour class of applications are not considered in this section as they are beset by excessive coherence communication overheads, and simply are not competitive for applications with low computation to communication ratios.

EP

EP has a very large computation to communication ratio, and this enables the performance of SMG to be very competitive with MPI, see Figures 10.3 to 10.5, and Table 10.5 (SMG-sub refers to the use of the subscription protocol). This is hardly surprising as little work is required of the DSM. This does demonstrate that the DSM engine imposes little if any overhead, and demonstrates the effectiveness of having a communication module with a separate thread. Note that in this application, the performance is independent of the subscription protocol (as indicated by similar plots in the graphs below) as the DSM engine is employed very little throughout the execution.
The EP application itself results in little communication, with the bulk of any communication usage for the SMG variants resulting from the control messages for the initialisation and finalisation of the system.

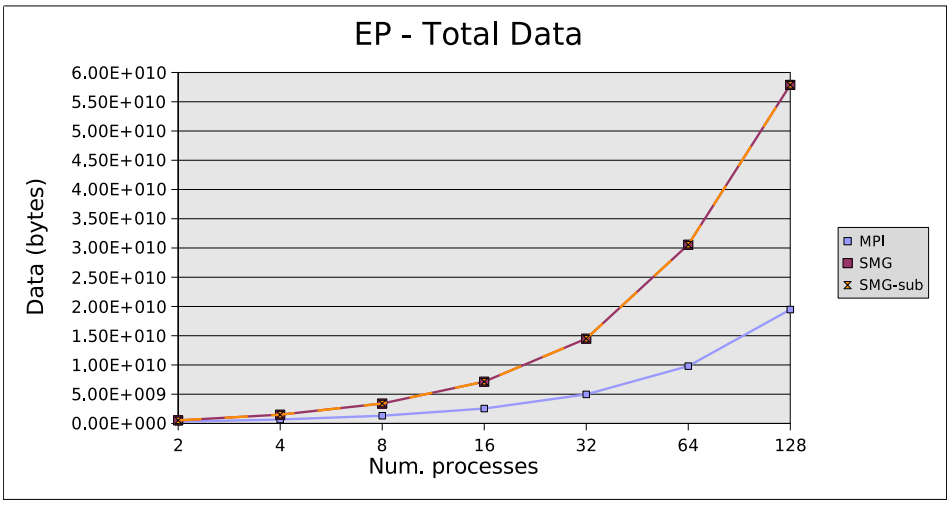


Figure 10.3: *EP Total Data sent (SMG is obscured by SMG-sub)*

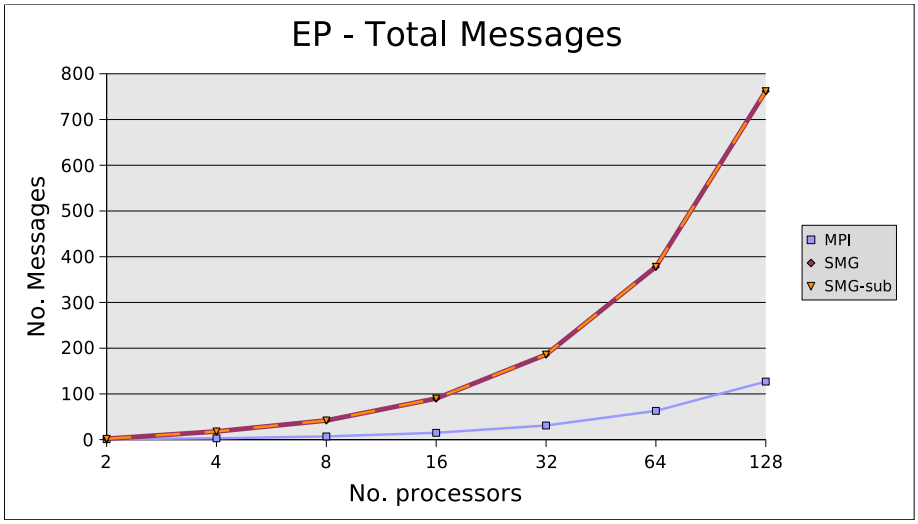


Figure 10.4: *EP Total Messages sent (SMG-sub obscures SMG)*

No. Procs	2	4	8	16	32	64	128
MPI	2	4.02	8.04	14.5	28	53.9	86.19
SMG	2	3.99	7.99	15.87	31.15	58.56	87.51
SMG-sub	2	3.98	7.49	15.8	29.05	54.69	80.15
SMG (threaded)		3.77	7.54	15.91	28.25	55.77	87.02

Table 10.5: *Speed-up for EP execution on IITAC*

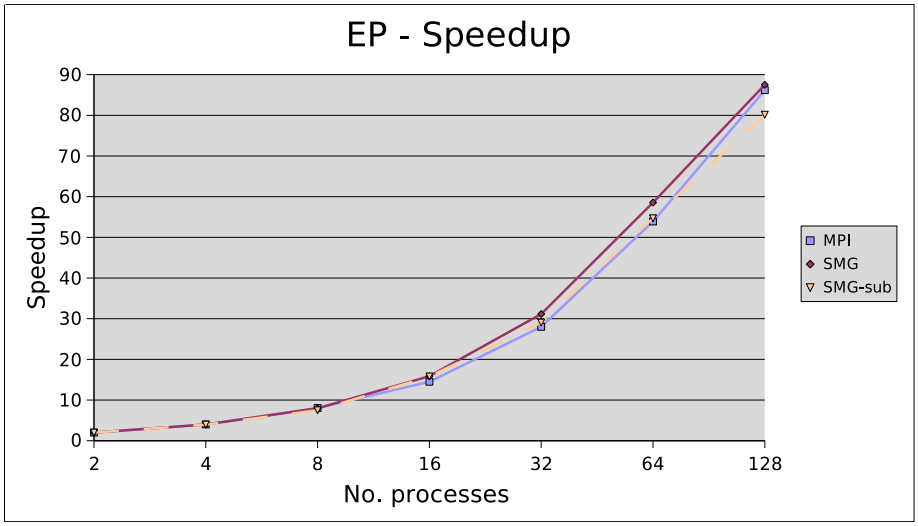


Figure 10.5: *EP Speedup*

Matrix

The communication behaviour of the Matrix application (Figures 10.6 to 10.8) highlights that SMG imposes no major overhead for applications with high computation to communication ratios, and so good speedup is obtainable as shown in Table 10.6. The volume of data transferred is greater than for MPI as once the resultant matrix is gathered through the release phase of the barrier process at the coordinator it will be distributed to all other processes during the acquire phase. Coherence traffic could be minimised at the acquire phase of the initial barrier if the barrier coordinator performed the initialisation of the incident matrices, i.e. no coherence data would be seen during the arrival phase. The addition of a simple function to the SMG API can easily provide this:

SMG_barrier_iam_coordinator(int id).

The subscription protocol variant of the SMG application can offer no advantage as the sharing pattern cannot be established until after the first barrier. The protocol introduces small amounts of additional coherence overhead to initialise the subscription lists.

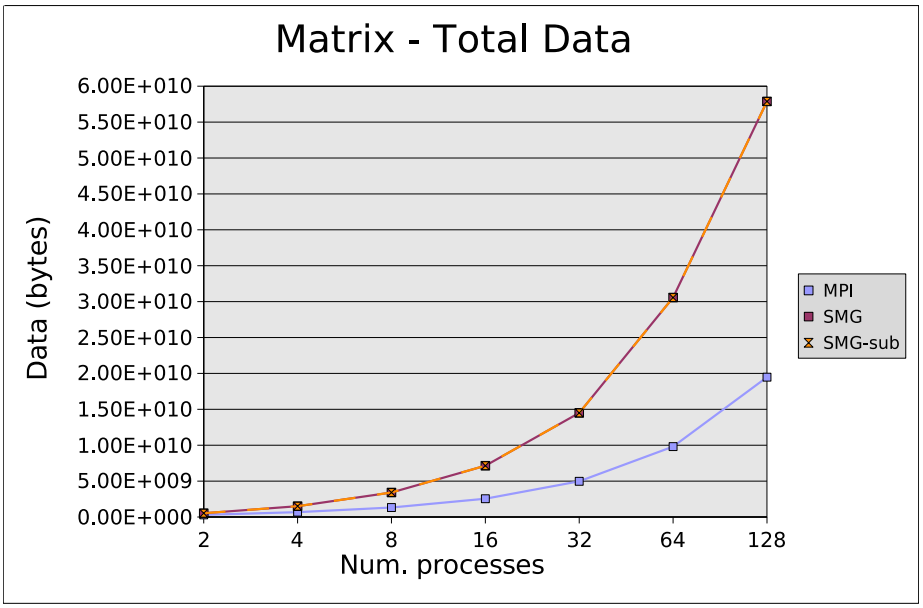


Figure 10.6: *Matrix Total Data sent*

The number of messages generated with the MPI versions is also lower than for SMG for the Matrix application, see Figure 10.7, but these values are inclusive of initialisation messages for the DSM engine. In general SMG will be competitive in terms of numbers of messages generated, but not for the cumulative payload of those messages.

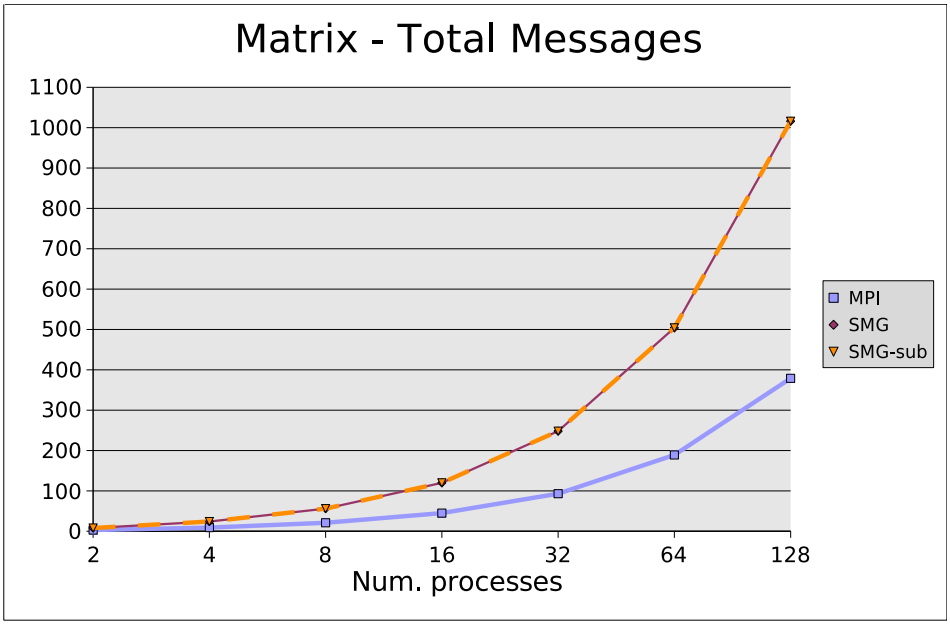


Figure 10.7: *Matrix Total Messages sent*

There is no discernible difference in the execution times between the SMG variants apart from that the update protocol variant does suffer a degradation compared to the other variants for processor counts of 128; this may be due to intermittent effects. The seemingly unusual speed-up figures given in Table 10.6 (2.01, 4.17, and 8.06), for the MPI variant of the application with processor numbers of 2, 4, and 8, would seem to stem from the ability for a processor to maintain a larger amount of the application's working set within the processor's cache, which results in a higher cache hit-rate, thus super-linear performance with respect to a single processor run.

No. Procs	2	4	8	16	32	64	128
MPI	2.01	4.17	8.06	15.76	30.96	66.96	114.54
SMG	2	3.99	7.97	15.24	29.76	56.09	104.17
SMG-sub	1.98	3.98	7.89	15.66	30.11	56.03	107.76
SMG (threaded)		4	8.44	16.5	32.2	58.54	107.93

Table 10.6: *Speed-up for Matrix execution on IITAC*

At this point, it is important to highlight the increase in the number of DSM page-faults, as illustrated in Figure 10.9, when the subscription protocol is used rather than the update protocol. This 'feature' was expected, see Section 7.4.2.

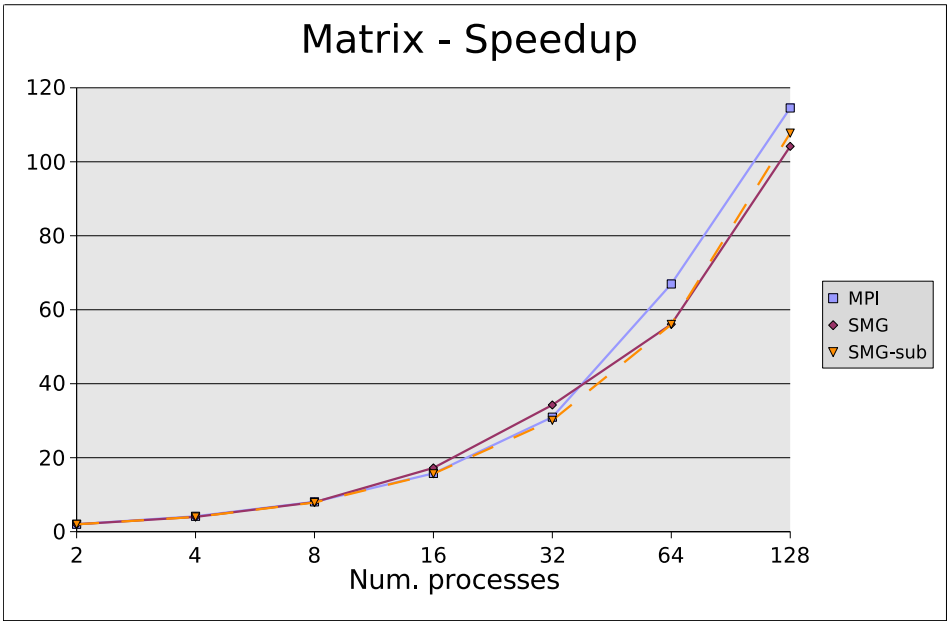


Figure 10.8: *Matrix Speedup*

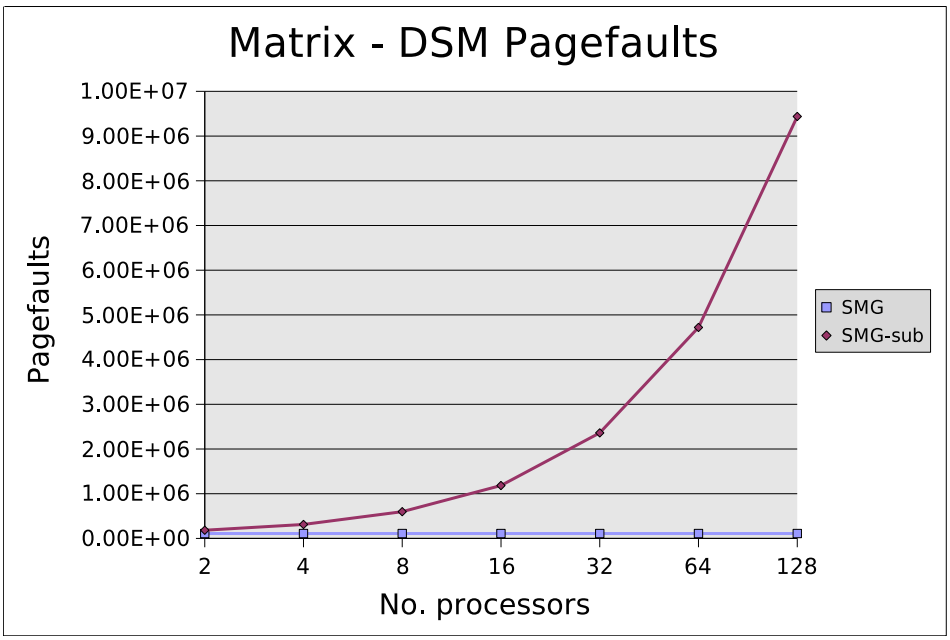


Figure 10.9: *DSM pagefault count for Matrix*

Laplace

The major deficit of the update protocol is evident in the message payload figures for the Laplace application, see Figure 10.10. The 'avalanche' effect associated with the update protocol is self-evident when the process size increases beyond 16, resulting in the parallel application being communication bound for a significant amount of time as the interconnect becomes saturated with traffic, see the difference in Table 10.7 for the total data transferred between SMG and MPI for applications with 128 processes (This is the scenario described in Section 7.4.1). The SMG message payloads are orders of magnitude greater than for message passing versions. The numbers of messages are relatively similar for all cases, see Figure 10.11. The benefits of the subscription protocol are further evidenced by the reduction in the data volume transferred, but there is still significant overhead associated with the protocol implementation.
The subscription protocol does offer some respite in terms of speedup at low processor numbers (Figure 10.12), but as the number of processors are scaled up the overhead associated with the protocol implementation becomes detrimental to the performance of the application. This figure also demonstrates the terrible performance of SMG using the update coherence protocol i..e no speedup occurs for this application, due to being communication bound.
It must be noted that the MPI version of the application also performs poorly when the number of processors are scaled into double digits, and performs very badly thereafter. The computation-communication ratio is not high enough to deliver good scalability. Larger application dimensions are possible, say 8192 $\times$ 8192, but for the purpose of this thesis, i.e. the performance of the SMG DSM, are inconsequential.
As indicated by Figure 10.13, the subscription variant of the application does incur extra DSM overhead. The most visible manifestation of this is the number of extra page-faults that are generated.

No. Procs	4	8	16	32	64	128
MPI	9.36E+8	1.32E+9	2.55E+9	4.97E+9	9.81E+09	3.96E+10
SMG	9.65E+9	1.71E+10	3.99E+10	7E+10	2.98E+12	5.88E+12
SMG (Sub)	9.46E+8	2.28E+9	5.17E+9	1.19E+10	2.87E+10	7.65E+10

Table 10.7: *Laplace - Message payload (in bytes)*

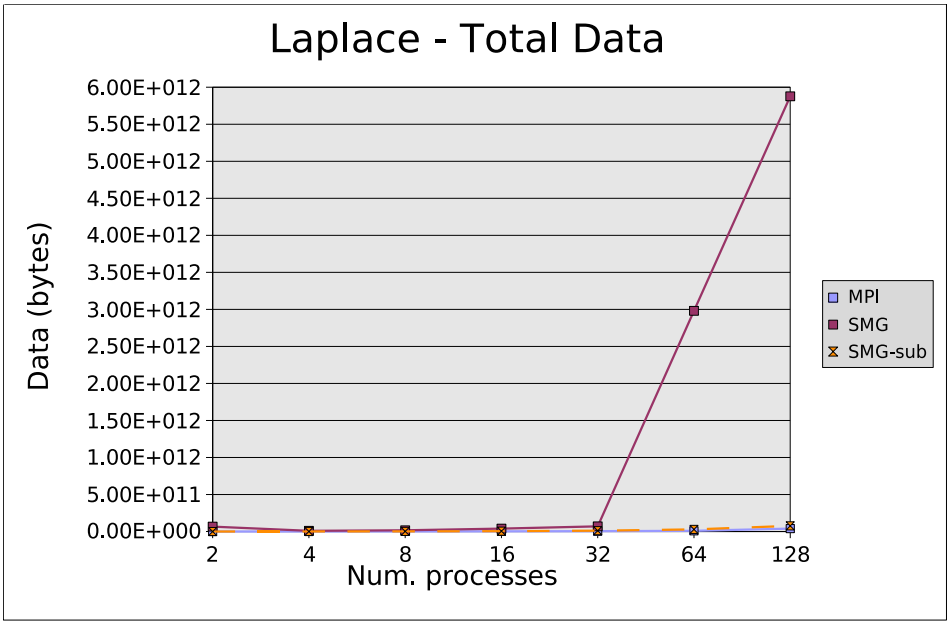


Figure 10.10: *Laplace Total Data sent*

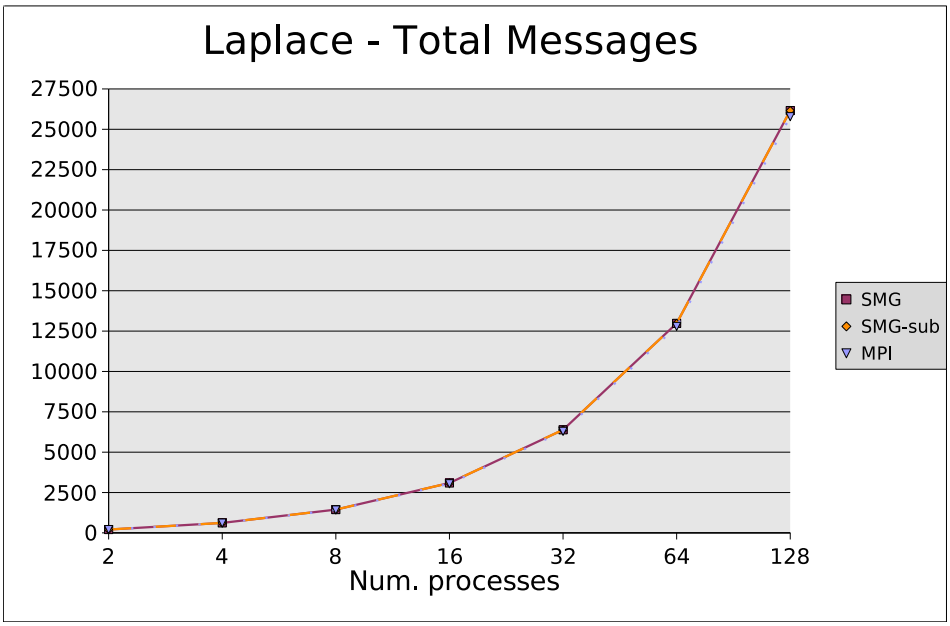


Figure 10.11: *Laplace Total Messages sent*

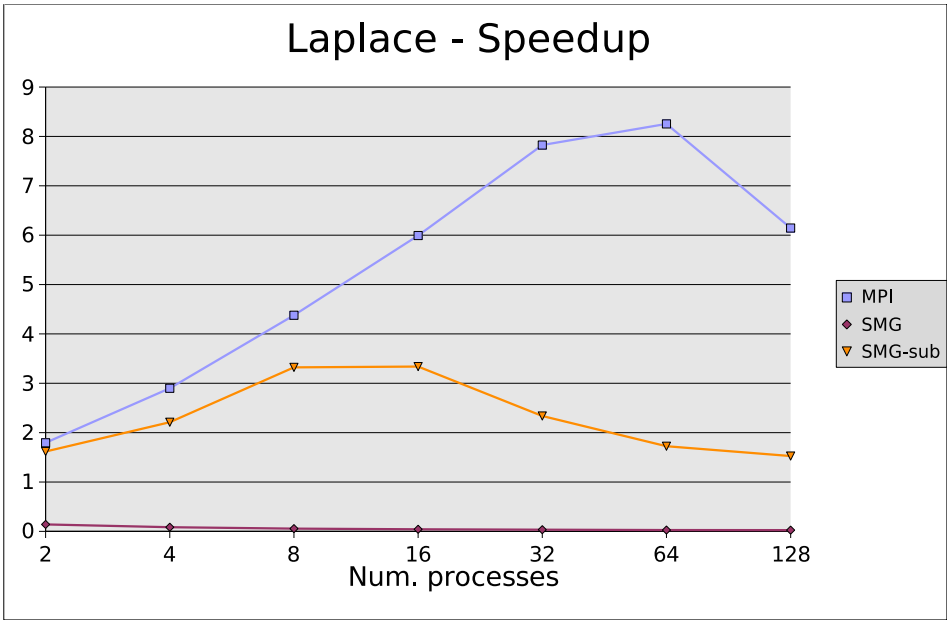


Figure 10.12: *Laplace Speedup*

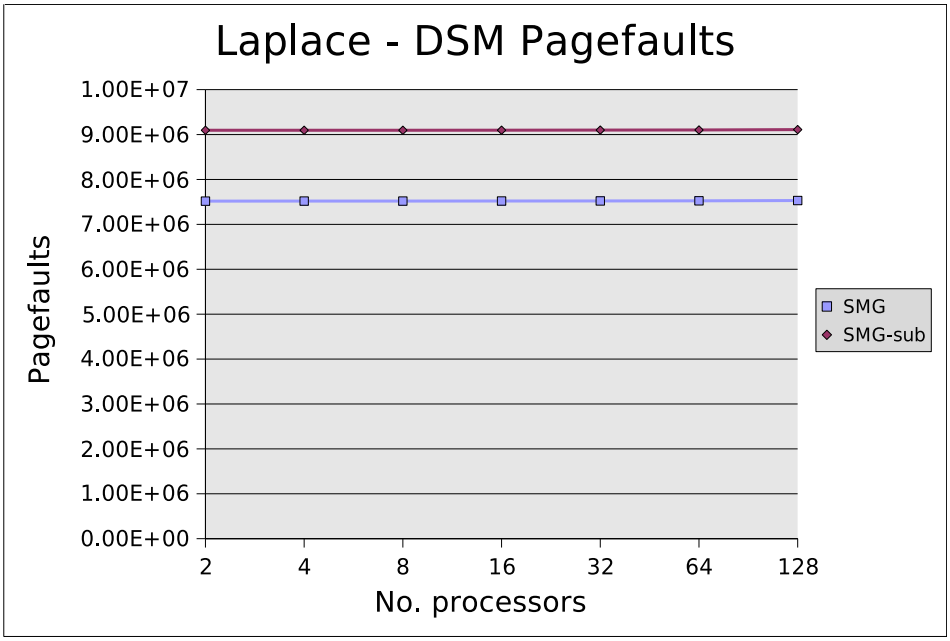


Figure 10.13: *Laplace - Pagefault count for different protocols*

SOR

Again, the major deficit of the update protocol is evident in the message payload figures, see in this case in Table 10.8, or in graphical form in Figure 10.14. The SMG message payloads are significantly greater than for message passing versions. The numbers of messages are relatively similar for all cases (Figure 10.15). The benefits of the subscription protocol can be seen in the dramatic reduction in the data volume transferred compared to the SMG update protocol version, but it is still significantly higher than for MPI. Like the Laplace application there is little speed-up from either SMG version, but as shown in Figure 10.16 for larger numbers of processors all variants performance deteriorates as the computation-to-communication ratio is too small to maintain scalability.
Figure 10.17 illustrates the DSM overhead associated with the use of the subscription protocol with respect to the update protocol. This difference may be reduced if the rate at which the subscription performs flushing of the local process' own subscription list (see Section 7.4.2) is increased.

No. Procs	4	8	16	32	64	128
MPI	3.607E+8	4.391E+8	5.960E+8	9.098E+8	1.537E+9	2.793E+9
SMG	1.894E+9	4.056E+9	8.035E+9	1.562E+10	3.038E+10	5.949E+10
SMG (Sub)	5.398E+8	6.396E+8	1.179E+9	3.336E+9	7.708E+9	1.628E+10

Table 10.8: *SOR - Message payload (in bytes)*

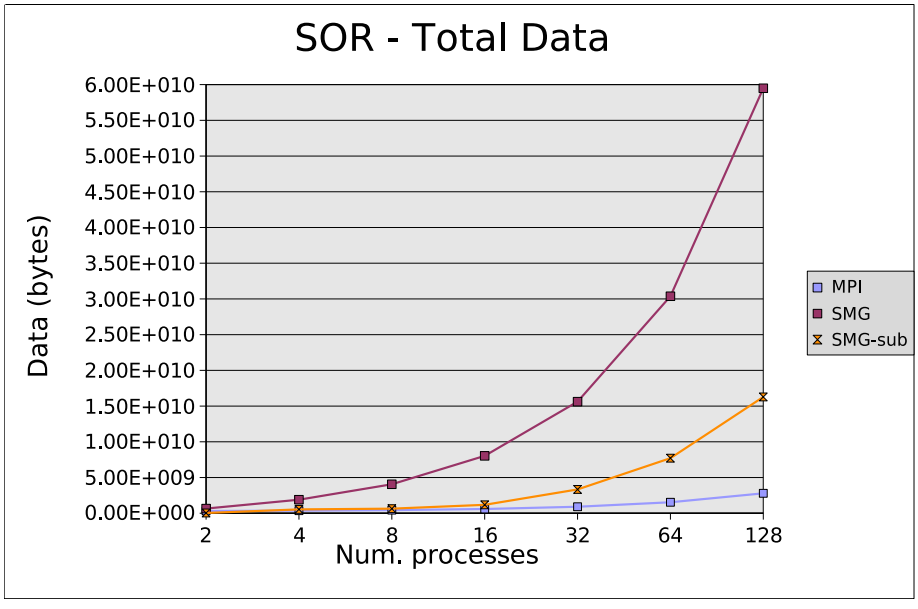


Figure 10.14: *SOR - Total Message payload of MPI and SMG consistency protocols*

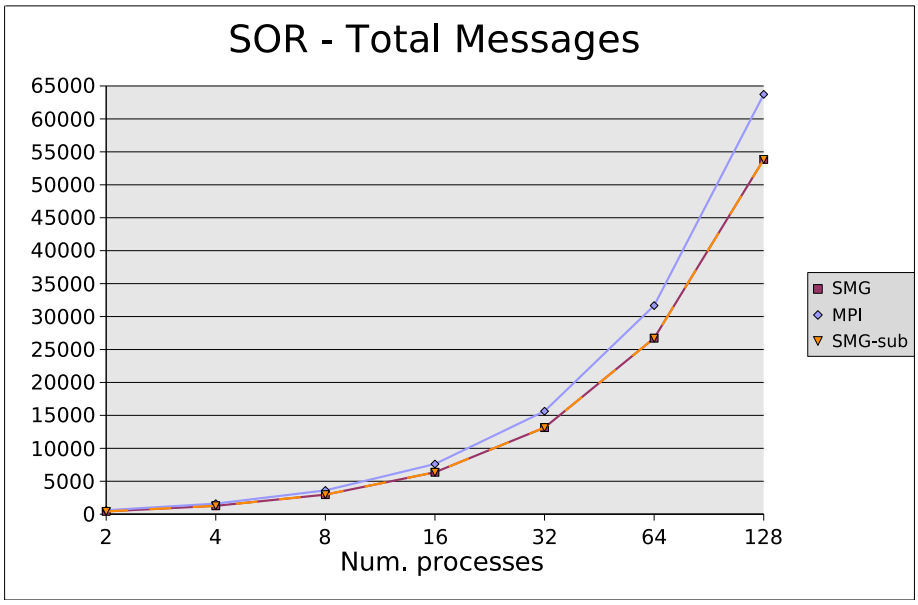


Figure 10.15: *SOR - Total Messages sent*

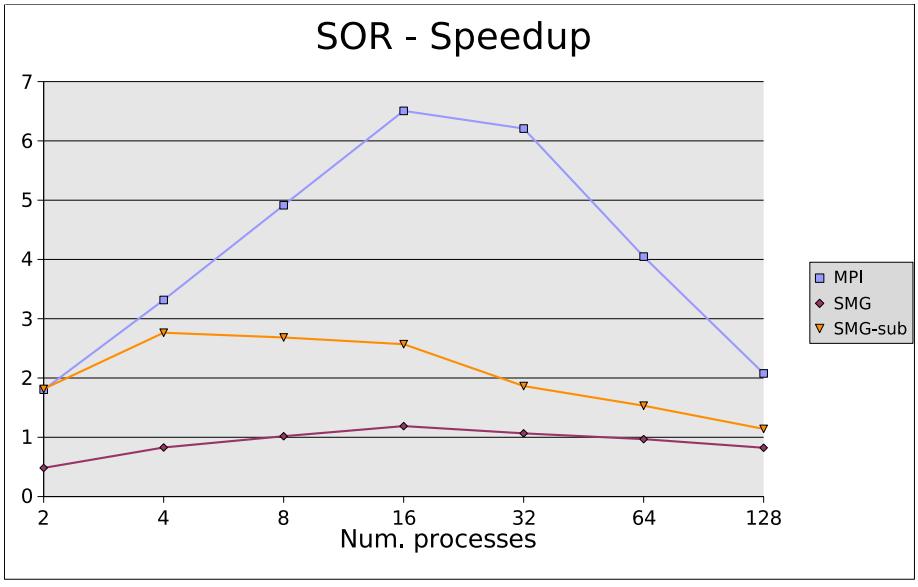


Figure 10.16: *SOR - Speedup*

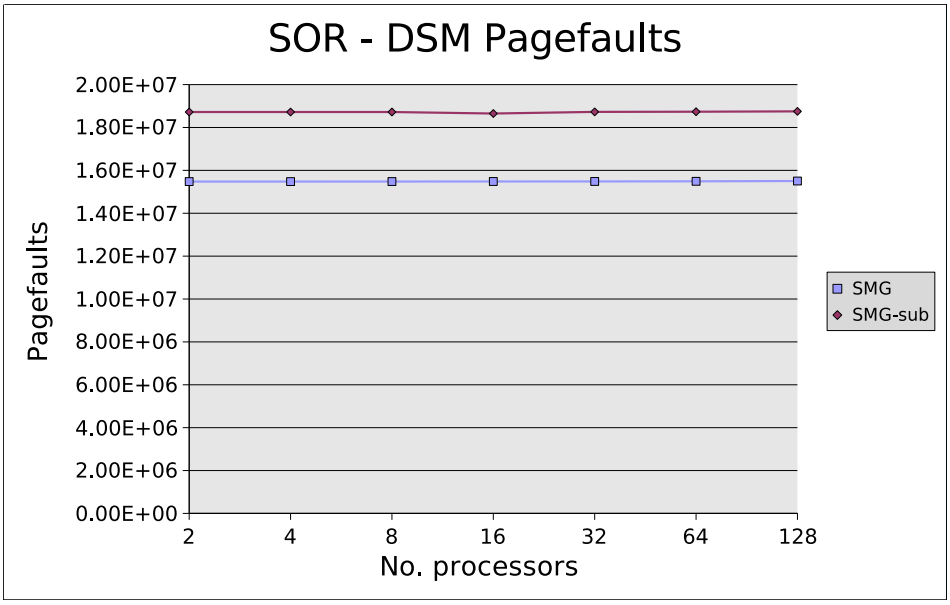


Figure 10.17: *SOR - Pagefault for different protocols*

10.3 Benefits of Hybridisation

Identification for hybridisation was confined to the those locations identified with executions for small processor runs. All test applications are relatively small in size, and so only one/two worthwhile candidates were identified for each.

EP

There is no identifiable scope for hybridisation with the EP application, as there is little communication. Thus the application can be equally well implemented using either shared memory or message passing. Hybridisation with message passing would have no benefit, and might even degrade performance if only in a trace amount due to the overhead with invoking the hybridisation support routines.

Matrix

Again, there is little identifiable scope for hybridisation with the Matrix application and hence the application can be implemented using either the shared memory or message passing style. The incident matrices of the application are initialised by one process and these are propagated to the rest, as is typical for the read type of shared variable described in Section 4.1.
Hybridisation can ensure that the gathering of results is restricted to only the root

process, which is the case implemented using the MPI version, but not the DSM version where by default the whole result-set is distributed to all processes. The effect of this can be seen in the reduction in the volume of data transferred (Figure 10.18), accompanied by a slight increase in speed-up (Figure 10.20).

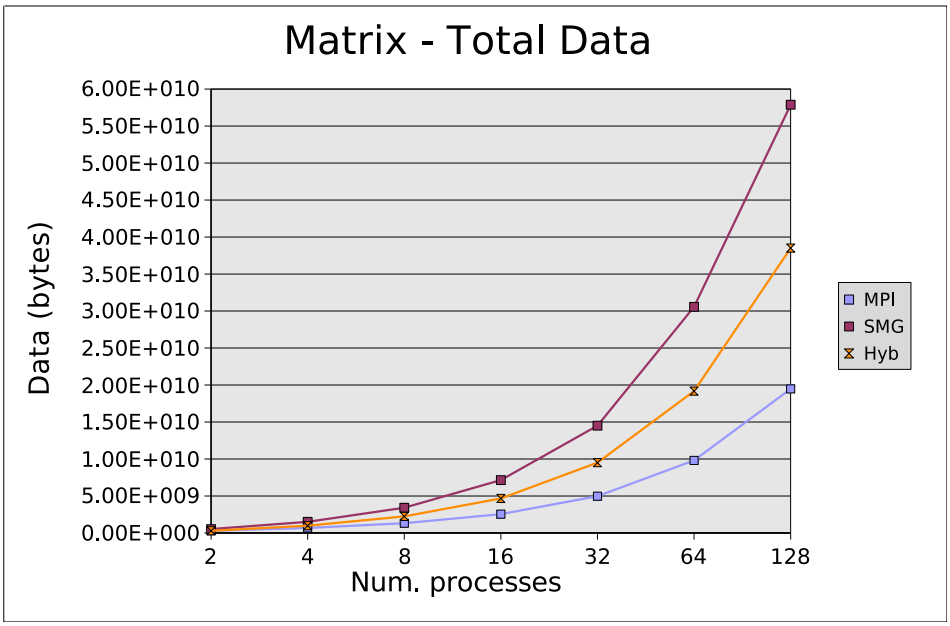


Figure 10.18: *Matrix - Total messages with MPI/SMG hybrid version*

No. Procs	2	4	8	16	32	64	128
MPI	2.01	4.17	8.06	15.76	30.96	66.96	114.54
SMG	2	3.99	7.97	15.24	29.76	56.09	104.17
Hyb	2	4	7.98	15.87	31.09	56.76	108.63

Table 10.9: *Speed-up for Matrix execution on IITAC*

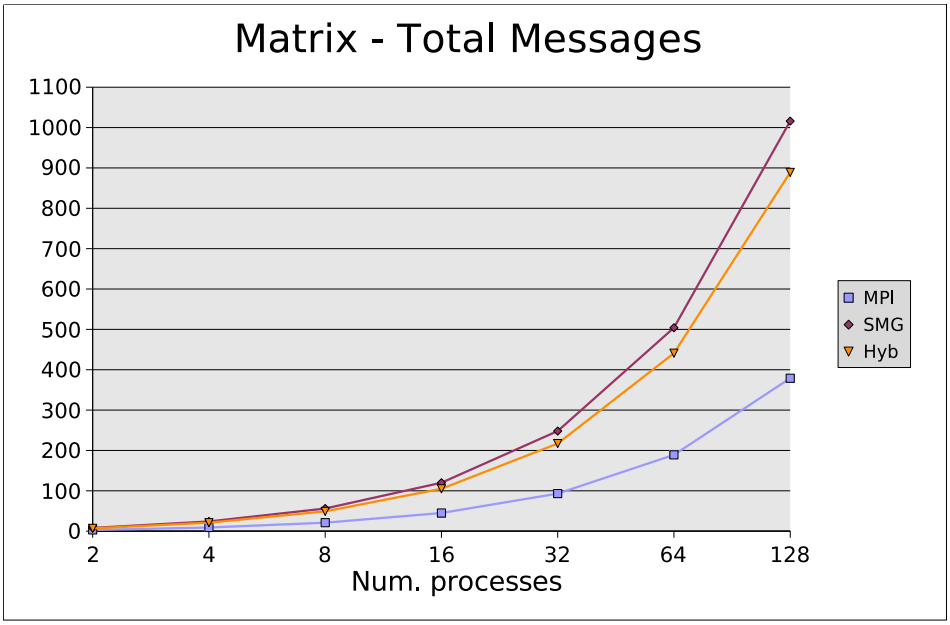


Figure 10.19: *Matrix - Total data with MPI/SMG hybrid version*

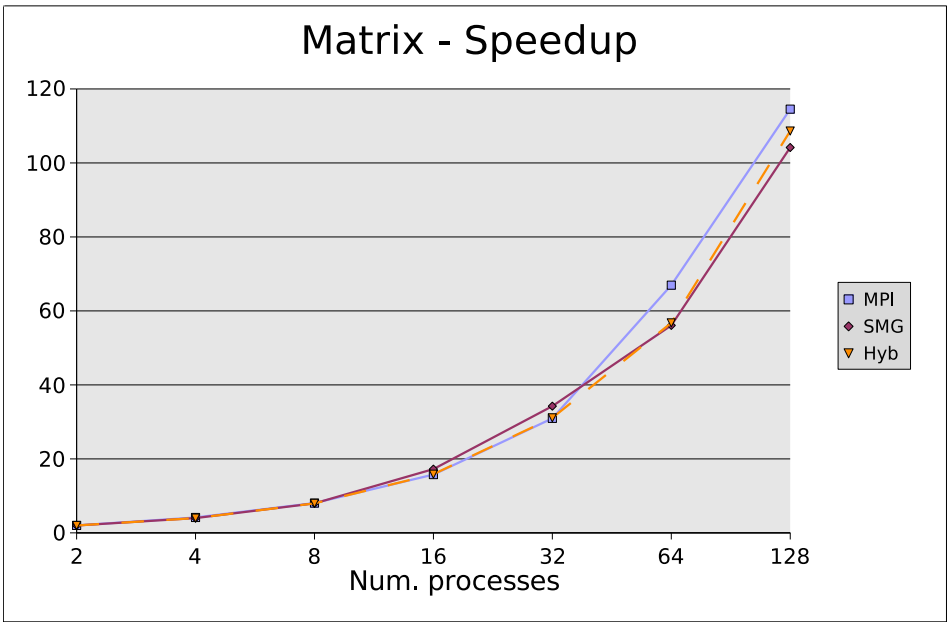


Figure 10.20: *Matrix - Speedup of MPI/SMG hybrid version*

Laplace

Like the other iterative application in the test suite, SOR, the hybridisation process identified the main iterative loop as the prime candidate for hybridisation. Disabling DSM controls and replacing the barrier with message passing calls results in a dramatic reduction in the volumes of data transferred as shown in Table 10.10, while the number of individual messages being transferred remaining unaffected.
Some minor speedup results when the application is run with small numbers of processors. Again the computation to communication ratio is a barrier that is unable to be overcome given the stated application problem dimensions and the volume of data transferred as a result of DSM activity during the initialisation and finalisation stages of the application.

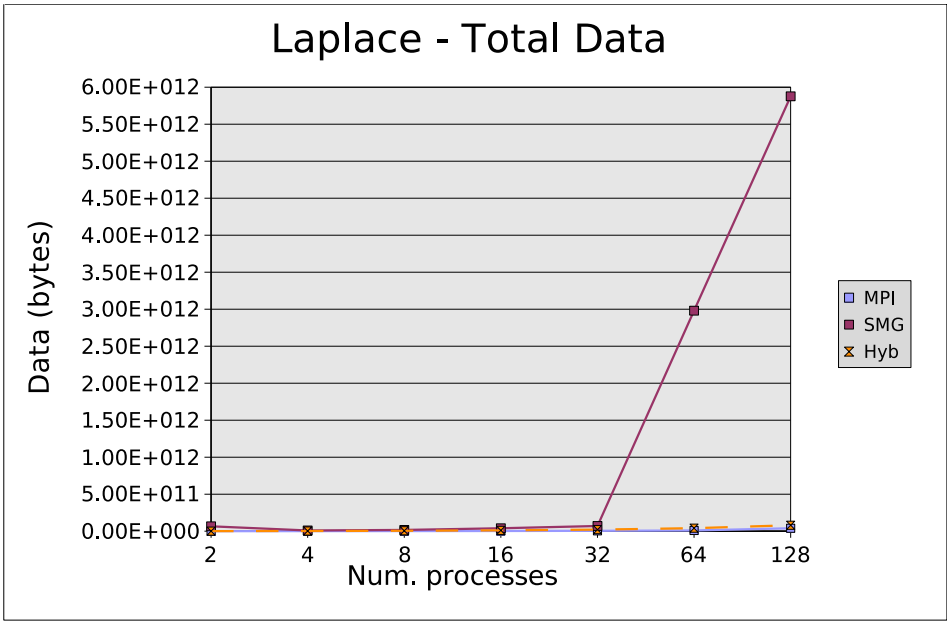


Figure 10.21: *Laplace - Total Message payload with MPI/SMG hybrid version*

No. Procs	4	8	16	32	64	128
MPI	9.36E+8	1.32E+9	2.55E+9	4.97E+9	9.81E+09	3.96E+10
SMG	9.65E+9	1.71E+10	3.99E+10	7E+10	2.98E+12	5.88E+12
Hybrid	2.44E+09	5.39E+9	1.08E+10	2.11E+10	4.12E+10	8.04E+10

Table 10.10: *Message payload (in bytes) for hybrid Laplace*

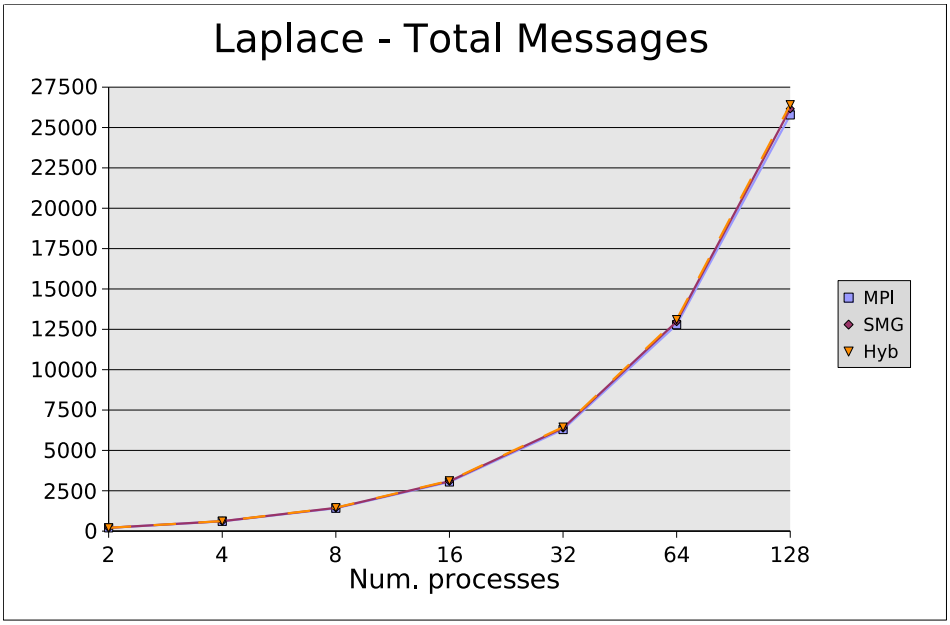


Figure 10.22: *Laplace - Total Messages with MPI/SMG hybrid version*

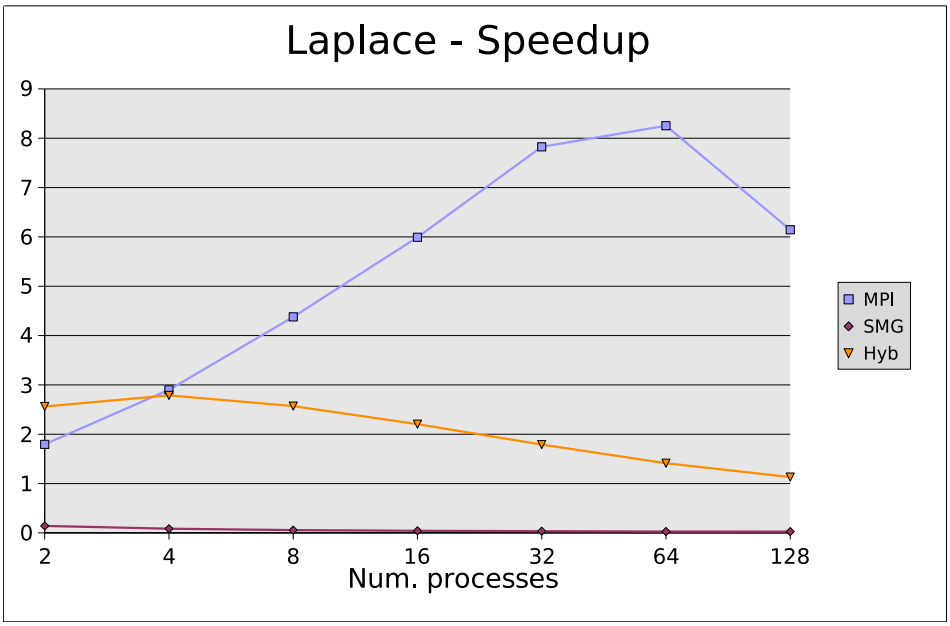


Figure 10.23: *Laplace - Speedup with MPI/SMG hybrid version*

SOR

The hybridisation engine identified some potential candidates for hybridisation in the SOR application, most notably in the main iterative loop. The hybridised application results in a significant decrease in the volume of data transferred by the application as enumerated in Table 10.11, with no change in the number of messages generated.
Like the Laplace application a small increase in speedup results, see Figure 10.26 when the application is executed with a low processor count, as the effects associated with the increase in coherence communication, as shown in Figure 10.24, become dominant resulting in performance decreases dramatically in all cases (computation-to-communication ratio doesn't support execution across a large number of processes).

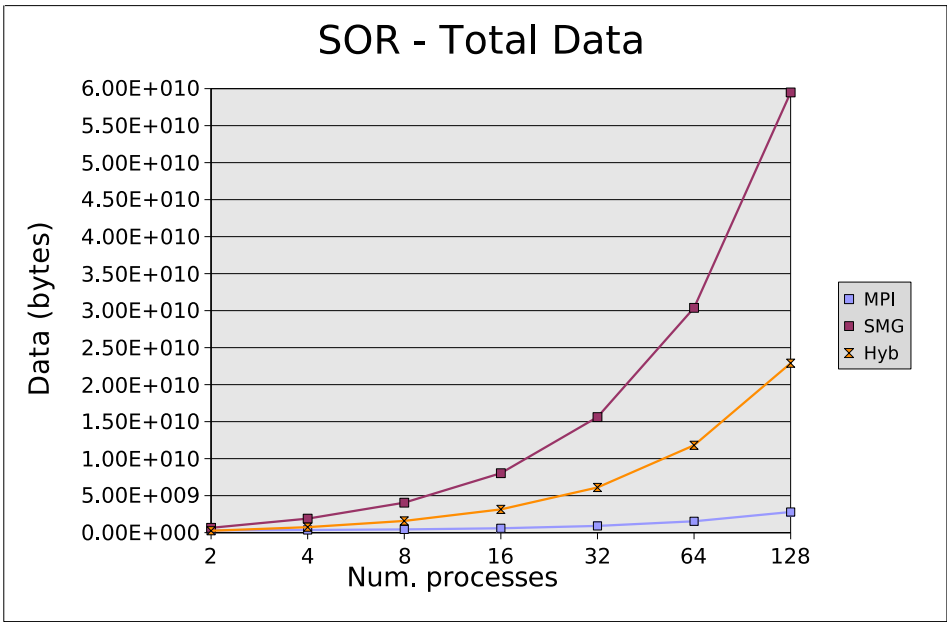


Figure 10.24: *SOR - Total Message payload with MPI/SMG hybrid version*

No. Procs	4	8	16	32	64	128
MPI	3.61E+8	4.39E+8	5.96E+8	9.10E+8	1.54E+9	2.79E+9
SMG	1.89E+9	4.06E+9	8.03E+9	1.56E+10	3.04E+10	5.95E+10
Hybrid	7.43E+8	1.59E+9	3.14E+9	6.11E+9	1.18E+10	2.29E+10

Table 10.11: *Message payload (in bytes) for hybrid SOR*

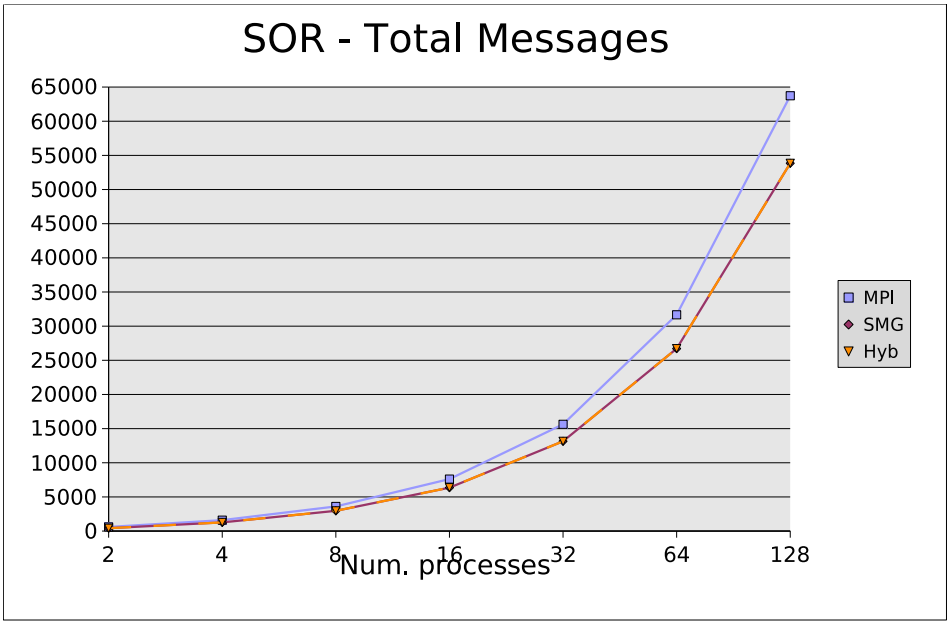


Figure 10.25: *SOR - Total Messages with MPI/SMG hybrid version*

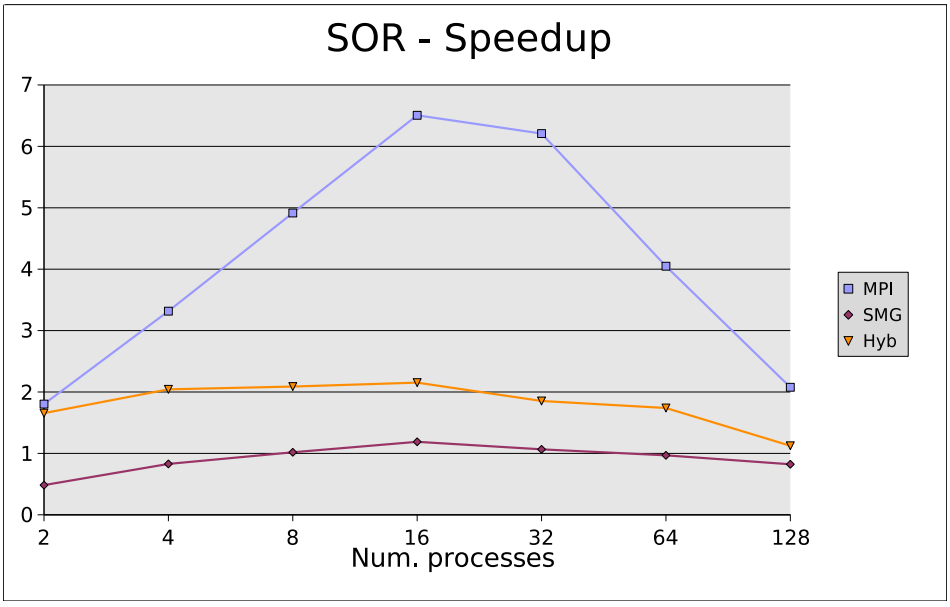


Figure 10.26: *SOR - Speedup with MPI/SMG hybrid version*

10.4 Grid Performance of Applications

For the grid simulations there are S (= 4) sites, with N/S processors per site. The initial processor count is taken to be N = 16, as the benefits of the information & monitoring system are only demonstrable at this scale. Note that because the grid environment is simulated, it does not exhibit many of the real properties of grids, such as non-determinism, or errors and failures of nodes, processes or communications.

EP

When executed on the simulated Grid, EP does not exhibit any slowdown compared with execution on a cluster with a similar configuration (apart from the simulated latencies), assuming the same update coherence protocol, see Table 10.12. This is only to be expected for an embarrassingly parallel application.

No. Procs (N)	16	32	64	128
SMG Grid-aware	14.94	27.77	52.71	84.13
SMG Grid-not aware	14.07	27.6	51.21	77.72

Table 10.12: *EP - speedup for execution on Grid with S=4 sites*

Matrix

The matrix application experiences some slow-down (Table 10.13), but this is not too significant as the hierarchy awareness is able to ensure that only the minimum number of messages required are sent between sites no matter how many processes in the application, see Figure 10.27.

No. Procs (N)	16	32	64	128
SMG Grid-aware	15.78	30.91	57.46	103.78
SMG Grid-not aware	14.87	29.63	55.04	93.14

Table 10.13: *Matrix - speedup for execution on Grid with S=4 sites*

Laplace

Laplace, when executed on a grid, experiences even greater slowdown than the cluster version due to the latencies associated with transferring large quantities of data over the simulated inter-site interconnect, see Table 10.14. This slowdown is made worse still without the presence of topology information that allows for the reduction in the

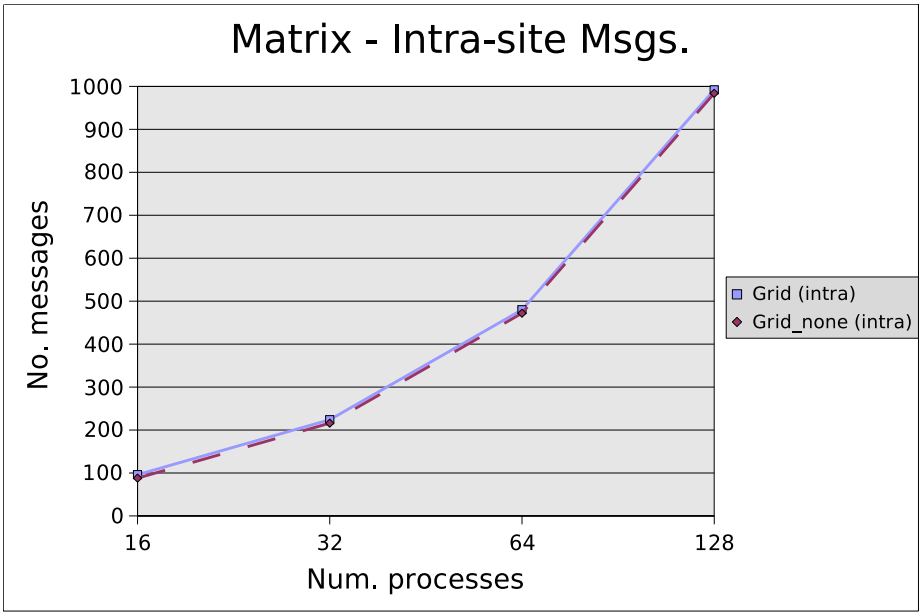


Figure 10.27: *Matrix - Total inter-site message count*

number of inter-site messages (see Table 10.15). It must be noted that the relevance of the number of inter-site messages reduces with respect to the number of intra-site messages as the processor count increases.

No. Procs (N)	16	32	64	128
SMG Grid-aware	3847.09	5415.66	6021.69	6922.78
SMG Grid-not aware	3921.83	5539.42	6204.94	7111.77

Table 10.14: *Laplace - Execution times on simulated Grid with $S = 4$ sites*

No. Procs (N)	16	32	64	128
SMG Grid-aware	618	618	618	618
SMG Grid-not aware	824	824	824	824

Table 10.15: *Laplace - Inter-site messages on simulated Grid with $S = 4$ sites*

SOR

The SOR application demonstrates the advantages of using knowledge of the grid topology to reduce the number of inter-site messages. The benefit of this approach is eroded for systems composed of large numbers of processors as the ratio of inter- to intra-site messages becomes less favourable. The choice of grid application parameters obfuscates the potential advantages as the distribution of processes across the S sites is evenly balanced. If this were not the case the values for the execution time with no awareness would be further increased as shown in Table 10.16. See Table 10.17 for the values of inter-site message counts when the number of processors are not so evenly distributed.

No. Procs (N)	16	32	64	128
SMG Grid-aware	104.84	115.96	119.47	151.20
SMG Grid-not aware	109.66	123.19	142.86	166.38

Table 10.16: *SOR - Execution times on simulated Grid with $S = 4$ sites*

No. Procs (N)	16	32	64	128
SMG Grid-aware	1272	1272	1272	1272
SMG Grid-not aware	1696	1696	1696	1696

Table 10.17: *SOR - Inter-site messages on Grid with $S = 4$ sites*

The table below shows the difference in inter-site messages for the SOR application with & without an information system available to provide data for it to generate a topology tree (thus enabling a more efficient barrier implementation). These values demonstrate the potential increase when the distribution of processes is not uniform across sites.

No. Procs (N)	Grid-aware	Grid-not aware
Messages 16 procs (3/4/5/4)	1272	1272
Messages 16 procs (3/3/5/5)	1272	2120
Messages 32 procs (6/6/10/10)	1272	2968
Messages 64 procs (8/8/24/24)	1272	2968
Messages 128 procs (25/25/39/39)	1272	4664

Table 10.18: *Difference in Inter-site Messages for SOR using SMG in a simulated Grid, with & without the information system*

10.5 Analysis

The mechanisms developed in the course of this thesis have demonstrated their potential to reduce the communication overheads imposed by the use of a DSM. Overall the SMG DSM engine incurs small amounts of overhead: initialisation and termination times are reasonably comparative to MPI. Slight overhead results in the termination phase due to the issues regarding shutdown of the multi-threaded communication implementation (discussed in Section 6.4.3); for an application composed of P processes this can be approximated to $5log_2P$ seconds.

10.5.1 SMG overheads

The overhead caused by the DSM is substantial, but for the right type of application with a large computation to communication ratio then this overhead becomes negligible. The largest component, by far, is ultimately due to communication, either waiting for a blocking request or the time to actually send request/response messages over the wire (although the latter is reduced where the DSM engine is multi-threaded, even more so if the communication subsystem is itself multi-threaded). The SMG startup and shutdown times will vary depending upon both the communication implementation and the information & monitoring system.

10.5.2 SMG vs MPI

The SMG DSM tries to compete with message passing versions as best it can, however the coherence update model results in many simple and trivial applications becoming communication bound. Clearly the update protocol is only suitable for data that is infrequently modified, and in those situations all modifications are required by all other processors. The most apparent situation is where data belongs to the read category of shared memory that was discussed in Section 4.1. The most common scenario is where data is initialised at the start of an application and is only ever read for the duration of the application; this happens in the matrix benchmark.
The subscription protocol improves dramatically on the update protocol but still cannot match the efficient use of the communication system by MPI. The scenarios where this can occur are illustrated by the nearest neighbour class of applications (SOR, Laplace) where the reduction is not as great due to the process mapping between the application and the barrier algorithm. For efficient performance the subscription protocol relies on the filtering effect of coherence data in the barrier tree. If a neighbour in the SOR algorithm is not a parent or child then any coherence data cannot be filtered as it is required by the parent. This situation is illustrated in Figure 10.28, where for the default barrier layout the neighbour pairs 2-3 and 3-4 communicate application data via intermediaries 1 and 5 respectively. A solution to this problem would be non-trivial as it would require a different barrier algorithm implementation that would take account of the sharing patterns of the application. This is difficult as a number of shared memory regions may require the barrier to maintain consistency, and each may have differing

requirements.
The current SMG subscription protocol implementation imposes extra communication overhead as coherence data sent by a child process in the release phase will be returned in the acquire phase of the barrier. This occurs in the current implementation at the cost of consuming less memory resources for maintaining separate coherence messages (extra processing time is also consumed filtering these events). If memory resources were less of a concern then this overhead component could be eliminated.

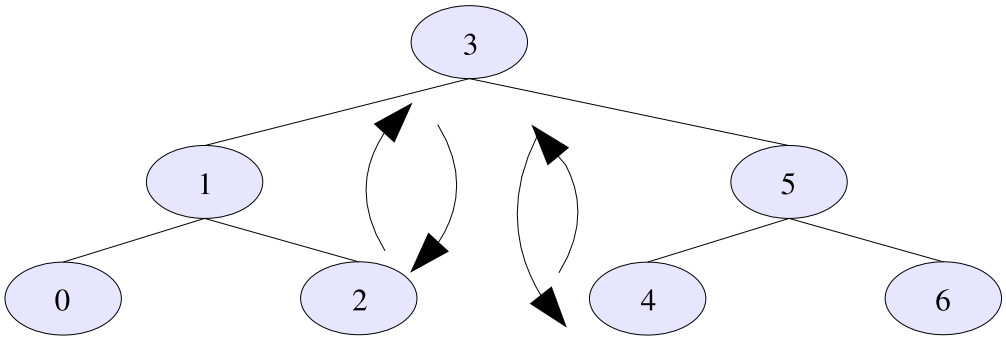


Figure 10.28: *Nearest Neighbour Problem-Barrier Mapping*

10.5.3 DSM variants

Although many variations can be applied to the SMG DSM, nonetheless the write trapping (Section 7.3.1) and write collection mechanisms (Section 7.5), and the barriers (Section 8.3.2), locating lock owners (Section 8.2.2) and lock queueing (Section 8.2.3) synchronisation variants are effectually inconsequential. It is clear that the only important variation is the combination of consistency model and coherence protocol, summarised anecdotically by Equation 10.4.

$$Communication = Consistency \times Coherence \quad (or \quad C = C^2) \tag{10.4}$$

10.5.4 Effects of Incremental hybridiation

We have seen the approaches used to enable DSM to execute efficiently, but in the best of cases the subscription protocol struggles. The best option is to hybridise the application by replacing the areas responsible with message passing. In some circumstances this is trivial as the communication pattern is regular and not too much extra burden is placed on the programmer. In the cases, where the communication pattern is irregular then the hybridisation process can prove irksome. In some cases the inefficiencies caused by the DSM are insurmountable.

In all cases where hybridisation is employed candidates will be identified, whether or not they truly are of relevance. The developer must further evaluate whether MPI code will indeed result in performance gains. The only metrics identified in the course of this thesis that could discriminate the effects were communication (latency & bandwidth) and DSM engine overhead (number of page-faults being the easiest for a user); these enable the developer to ascertain if it is potentially profitable to perform hybridisation.

10.5.5 SMG on Grid

The benefits of providing topology information are far less than originally expected. If the subscription protocol is employed then additional efficiencies are obtainable due to its multiplier effect in the reduction in the volume of data.
With the use of the information system the performance of synchronisation operations is improved. All barrier operations use only the minimum possible number of inter-site messages, i.e. for an application executing across S sites, then this number is $2 \times (S-1)$. The chosen simulated grid parameters, (i.e. evenly distributing the processors among all sites thus creating an efficient barrier even with no topology support) were used to gauge the benefits of topology awareness against the most efficient case where no topology information was present. This is evident in that the message volumes for the SOR application in Table 10.18 can be remarkably different even for small differences in the processor allocation where different numbers of processors are available at the four sites. In all cases the use of topology information results in the minimum of inter-site messages.
However, it must also be noted that the influence in the reduction of inter-site messages, in this way, reduces significantly as the processor count is increased. This is highlighted in the case of the Laplace and SOR applications by reference to Figures A.14 to A.15 and A.21 to A.22 respectively.

CHAPTER 11

Review

The premise of grid computing is that distributed sites make resources available for use by remote users. A grid job may be run on one or more sites and each site may consist of a heterogeneous group of machines. Grids are composed of geographically distributed sites, and traditional shared memory programs may not execute on processors located at different sites. DSM offers a potential solution, but numerous hurdles must be overcome before there will be an efficient implementation on the Grid. According to [24], if the paradigm could be made available, then grid programming would be reduced to optimising the assignment and use of threads, and the communication system.

Software distributed shared memory (DSM) aims to provide the illusion of a shared memory environment when physical resources do not allow for it. This thesis explored a possible realisation of this execution model for the grid. Typically a DSM run-time incurs substantial overheads that result in severe degradation in performance of an application with respect to a more efficient message passing implementation. This thesis examined mechanisms that have the potential to increase DSM performance by minimising high-latency inter-process messages and data transfers. Relaxed consistency models are investigated, as well as the use of a grid information system to ascertain topology information. The latter allows for hierarchy-aware management of shared data and synchronisation variables. The process of incremental hybridisation, where more efficient message-passing mechanisms can incrementally replace those DSM actions that adversely effect performance, is also explored.

Many obstacles were present, the foremost being the perception that grid-based DSM was a flawed concept. The SMG DSM was conceived as a tool to explore whether shared memory programming for grid application development was (a) feasible and (b) worthwhile. The initial objectives were stated in Chapter 1; this thesis has succeeded in achieving most of them.

The primogenial goal that *the DSM system should be constructed in a way that it is highly portable by ensuring the DSM uses only standard open libraries* has been achieved. The DSM is able to execute in a Grid environment as a result as a standard message-passing library, MPI, can be used for communication. Wherever the message passing library is

available then so too can the SMG DSM be. One difficulty is that SMG execution across multiple grid sites require a MPI library that is both multi-threaded and grid-enabled; currently existing libraries are one or the other, but not both. SMG offers a flexible API that enables it to be target-able by a future OpenMP source-to-source compiler. The SMG DSM goes further by supporting user multi-threaded applications.
The objective that for efficient application execution *the DSM should make use of information/monitoring systems where available to allow for the optimisation of user application code* is achieved, as SMG makes use of topology information that is accessible to the DSM at run-time. Communication should thereby be minimised, with a lowering of message count being preferable to total data transfer. Additionally, SMG implements a novel coherence protocol allowing further efficiencies to be obtained.
The stipulation in the third set of goals: *that a metric should be derived that illustrates the potential benefits. The user should be provided with an intuitive interface where the hybridisation process can be directed* has been partially achieved. Hybridisation was evaluated and demonstrated some promising results. The formal definition of a metric that defines the benefits of hybridisation has been achieved in a somewhat trivial manner, i.e. hybridisation can be rated in the reduction in DSM messages and/or the volume of data that would be transferred as these have a direct effect on the overall performance.

11.1 Contributions

A number of contributions were made during the work of this thesis:

1. **Extensible DSM:** the primary motivation for employing a DSM was the need for a tool that would assist exploration of the shared memory paradigm on the Grid. The many and varied disadvantages of previous DSMs prompted the development of a new DSM, SMG. Although yet another DSM implementation, it does provide for flexible extension via new coherence protocols, consistency models, and different communication mechanisms. SMG supports large numbers of processes; use of 128 processes has been demonstrated in this thesis, and 256 has also been attained. Other DSMs are limited to only 32 processes, or 64 in the case of the modified Treadmarks DSM used to implement Intel's Cluster OpenMP (described in Section 5.3.2).

2. **Grid-aware DSM:** SMG is the first grid-aware DSM. The DSM was designed to enable optimisations in the DSM engine by being hierarchy aware. Although message passing systems employ such techniques, this was the first DSM to integrate an information and monitoring system to enable the use of environmental information in this manner. The synchronisation primitives benefit from this approach as less inter site communication (messages and/or payload) results.

3. **Dynamic Subscription Protocol:** was developed to redress the deficits of the traditional protocols. This protocol performs very well for its intended audience

of iterative applications where multiple writers modify shared memory regions and where global barriers are used for synchronisation between iterations.

4. **Shared Memory Size:** the SMG DSM supports very large addressable shared memory regions. Previous DSMs were limited in the amount of data that could be shared (32MiB in the case of Treadmarks), and in most cases a shared memory region could take up this valuable space even if not required at a particular process. The maximum shared memory supported by SMG will be dictated by the limits of the virtual address space and the physical resources available. The former limit will be resolved with the move to a platform supporting a larger virtual address space (32 $\rightarrow$ 64-bit), while a possible solution to the latter can be solved with further work (once the move to 64-bit has been made).

5. **Communication:** SMG was the first DSM to use a multi-threaded flavour of MPI to provide communication. Other DSMs that use MPI for communication are still confined to using single-threaded MPI [168].

6. **Real Hybridisation:** the SMG DSM was the first to support true hybrid programming, enabling interleaving of DSM shared memory code with message passing schemes. Previous hybrid schemes just used different programming models in different phases, e.g. MPI for the regular data transfers, and when this was complete then the shared memory model for data accesses.

7. **Incremental Hybridisation:** the process of incrementally hybridising an application was explored. From these initial explorations it can be concluded that such a process places far too much burden on the developer. New burdens augment the normal burdens associated with message-passing model. These additional duties are required in order to maintain consistency of the shared memory region by validating the region's use once finished. However, it is possible that, in the future, it will be feasible for this task to be performed by a compiler.

8. **Identification of Application Suitability for the Grid-enabled DSM:** although seemingly obvious, this work has demonstrated, although through simulation, that an application with a high computation to communication ratio, such as those found in linear algebra applications, can be effectively executed in a grid environment using a DSM.

9. **Target for OpenMP translator:** SMG was designed mindful of the fact that another DSM API is not desirable. To this end cognisance was made of the growing acceptance of the OpenMP model for parallel programming. With little further development, support could be provided for enabling the DSM to form the target of a source-to-source translator.

11.2 Further Work

There are numerous topics for further work in the area covered by this thesis. In the area of the SMG DSM itself these include:

SMG Development

- **Non-blocking sync calls:** in a similar manner to the non-blocking routines provided by MPI, a similar provision by SMG could prove a valuable addition in certain circumstances.

- **General API Development:** support for the group-barrier implementation with shared memory functionality. Shared memory might only be allocated on processes that will be involved in the barrier operation, or maybe just at the local site, or just as a caveat in the use of this primitive.

- **Engine Optimisation:** while it is easy to observe that the DSM engine could be further optimised, results in the previous chapter illustrate that there is much scope for optimisation of the DSM engine in terms of support for multi-threaded applications. The root of these problems stems from the requirement for synchronised access to the internal DSM shared data structures.

- **Synchronisation:** it would be useful to implement a tree like structure such that when an all-to-one receive occurs then the root (receiver) can receive qualified votes from the child nodes, so that if the grid comprises a number of clusters then one node can be elected on each cluster to respond to the root with a head count of votes; otherwise there would possibly be n (n = nodes in cluster) messages traversing the network.
  Dynamic construction of Hierarchy Optimised Trees (HOT) for barrier use that adapt during the job execution to a (possibly) changing environment (i.e. the communication link attributes change) would be useful in grid environments. A variation on this theme would provide for the dynamic variation of the number of children a process has in the barrier primitive, i.e. a solution that changes these values (e.g. initial worst case scenario of N=2, that can increase based on workload/memory requirements).
  Support for alternate user lock queueing algorithms would allow various contention situations to be handled.

- **Memory views:** many of the current DSM implementations, SMG included, are implemented as a monolithic user level library, and as such are deficient in the provision of large memory configurations [169]. By porting the DSM to systems with a larger virtual address space, it would be possible with some optimisation for the DSM to support shared memory regions on a massive scale, i.e. disjoint sections of a large shared memory regions may be present as required. The subscription protocol will therefore facilitate scaling across large numbers of processors as the

impact caused by the bottleneck at the communication interconnect would be reduced.

- **Write trapping:** support of other consistency models will require modification to the page fault handler. With LRC, data is not guarded by a particular synchronisation primitive as is the case with EC. For this reason the page protection mechanisms employed are insufficient, as the possibility exists that while a page is being modified by the DSM system, a user thread may try to access the page. The method employed by previous DSMs was to have a user mapping and a system mapping to the same physical memory page, therefore having two sets of page protections, allowing for the system to update the page while maintaining page access protections by the user. Other approaches are described in [28].

- **OpenMP compiler:** one of the most valuable future objectives would be to target an OpenMP compiler[1] at SMG. This could entail constructing it from scratch, but a better option would be modifying one of the existing (open-source) source-to-source translators: Omni [170], OMPi [171], and OdinMP [172].

- **Hybridisation:** the current SMG hybridisation tool is basic. It was constructed to perform the simple task of identifying candidates for hybridisation. Constructing a component for a system such as Tau [155], would allow the exploitation of the other tools currently available from this (open-source) project (probes for OpenMP and MPI). Incremental hybridisation could be made easier to the user as tools are supplied to manage applications, and the strong focus of this system towards performance evaluation and tuning would be extremely beneficial to the hybridisation process.

General DSM

- **Coherence models:** the benefits of the per-site optimised home-less update protocols has been demonstrated t, but there are also advantages to the home-based approach [85].Per-site home based protocols should be explored. Dynamic adaptation between the update and subscription protocol is also an interesting possibility, e.g. some scientific applications can be composed of different phases: initially a preconditioner phase (like the nearest neighbour problem), followed by a compute intensive phase. Shared memory that is classified under the enumerated conventional type of Section 4.1 does not fit well in SMG, so additional coherence protocols such as one of the invalidate variants should be developed to aid in this area.

- **Subscription protocol:** further work is necessary on the optimisation of the subscription protocol implementation for supporting transient memory access patterns. When a fault occurs to a page that is invalid a process with a valid copy must be found. The current method of resolving a suitable location may generate

[1] It must be reiterated that in reality this would be a source-to-source translator rather than a full-blown compiler

a number of messages, resulting in higher latency, i.e. diametrically opposed to the goal of hiding latency.

- **New Consistency Models:** entry consistency is currently implemented with the option that a synchronisation object can be bound to the use of all barrier implementations (this is easy). If this was extended to all locks (not so easy), and then all synchronisation primitives (easy if the former two are present), then the result is *release consistency*. When coupled with the currently implemented update protocol this becomes *Eager-Release Consistency*. Upon development of an invalidate protocol (which would involve substantial work) lazy-release consistency could be supported for shared memory regions. Coupled with the subscription based protocol this will then be a very interesting prospect.

- **Write Trapping:** It was noted long ago that the page-fault access times are getting slower relative to processor clock rates [81] (Section 4.4). The page location mechanism for an invalid page in the subscription protocol is largely responsible. It is possible to direct the request at the barrier root, but this may create a bottleneck as there is currently a static barrier, so a dynamic distributed barrier management algorithm may need to be developed to overcome this.

- **Prior Application Use:** A case-based reasoner solves new problems by adapting solutions that were used to solve old problems [173]. As there are a number of applications with different memory access patterns, a case base could be built, and this could be used to evaluate and suggest possible candidates for hybridisation. This is an example of an adaptive strategy. Research has been performed on a similar theme involving the use of generic application skeletons for constructing efficient parallel applications [174]. This approach has previously been performed at compilation, i.e. hidden from the view of the developer.

- **Write Collection interoperability:** Currently all processes must use the same write collection protocol for a given shared object. This may not be an optimal strategy as in certain circumstances the write collection strategy should be different, i.e. one raw, one diff. This would require write collection interoperability, which itself is an admirable objective.

Supporting DSM in a Grid

- **Evaluation in a real grid:** with non-deterministic errors and failures not exhibited by the grid simulations of this thesis.

- **Communication:** Currently SMG will, by default, send inter-site messages before the local intra-site ones. Priority queueing for DSM messaging, as described in [133], would help. A possible further enhancement would be to make use of network link information, obtained dynamically from the information system, in this process.

- **Network Information Assisted:** the availability of network information would allow the optimal memory management strategy (i.e. coherence) to be chosen.
- **Fault tolerance:** the availability of fault tolerance will be a major influence [136, 135, 64]. Although fault tolerance is not supported by the SMG system, all the communication is through the MPI layer, so there is a basis for a fault tolerant DSM. Applications could be check-pointed at a user-specified global barrier.
- **Load balancing:** Grids will in the future be a heterogeneous mix of machines with varying performance characteristics. This is an important consideration, and so there is a need for run-time performance analysis methods so that load imbalances may be corrected. SMG offers a basic approach, however the burden is placed on the developer to utilise the functions; this job should ultimately be included in the tasks performed by any future compiler.
- **Heterogeneity support:** with SMG, the user does not instigate data transfers directly, so the problem of data transfers between differing architectures is made more difficult. As previously mentioned, the solution requires the developer to 'type' the data when allocating it. There are some DSM projects built using object-orientated technologies that use object reflection mechanisms provided by the language to do this work when transferring shared data [175]. This becomes very difficult when SMG is used as a target for OpenMP. The user would need to declare the type 'map', using a function for example:

```
int SMG_shmem_malloc_map(int num_structs, int num_types,
                         type_1, type_2, ..)
```

  Future support for heterogeneity is a noble objective. This problem is non-trivial, as shared data would need to be strongly typed, and tremendous issues abound with developing the write collection mechanism. This is one area where compiler support would prove beneficial, however this problem may be intractable.
- **Support for tightly coupled architectures:** for the most tightly coupled architectures, CPUs, the numbers of processing cores per processor package is expected to increase significantly in the near future. The memory bus will become even more of a bottleneck that will prevent further scaling up of multi-processor systems. The only way of effectively overcoming this will be to partition the system, so ending up with a distributed memory system. The other extreme of tightly-coupled architectures, low-latency grids (such as DAS-3 [47]), are by their nature distributed memory systems. DSM could provide an effective programming model for both cases.
- **Latency hiding:** ultimately DSM (with relaxed consistency models) tries to hide the latencies resulting from sharing memory among many distributed processes. Virtually sharing memory in this way despite the latencies between grid sites makes the task more difficult. Even weaker consistency models may be required.

11.3 Review

The seminal DSM implementations such as Treadmarks, Munin, and Midway were written for compute clusters, and not for Grid computing. Previous research has explored hierarchy-aware DSM consistency protocols [176], as well as OpenMP on distributed memory machines [177]. There have been other attempts at providing a shared memory model for wide area computing [178], and efforts are underway to implement the MPI-2 standard (which includes specifications for remote memory access and one sided communication) for grid computing.

This thesis presents an approach to supporting shared memory applications in a Grid environment. A new DSM implementation was required, as previous implementations had inherent deficits in terms of scalability, and their monolithic and constrained nature allowed little room for extension in terms of development of new protocols. Treadmarks, the exception to the constraints, was, and is, not publicly available. Implementing a new DSM and enabling integration with a grid information & monitoring system allowed for grid-optimised synchronisation primitives.

Open and mature standards such as MPI and OpenMP will eventually become the preferred foundation of parallel programming, and increasingly they will become more integrated to provide an optimal solution for developing applications for the grid. While an OpenMP compiler targeting the Grid has not yet been developed, DSM, and SMG in particular, will allow this to happen (when a source-to-source OpenMP to SMG translator has been developed). The SMG DSM attempts to demonstrate the potential advantages when message-passing and DSM programming paradigms are combined in the grid environment. The goal is to reduce the programming burden, and allow it to be followed by incremental optimisation. If this is achieved, it will promote the use of grids by allowing the exploitation of the very large collection of existing shared memory codes, and allow for easier parallelisation/grid-enabling of serial codes through the use of the OpenMP standard. The SMG coherence overheads are comparable with other DSM implementations but this will be improved when environmental awareness techniques and alterable write trapping/collection techniques are further improved.

This thesis has presented the concept of *Incremental hybridisation*, a method by which the regions of a shared memory application responsible for relatively poor performance could be targeted for conversion to a more efficient message passing style. It has proven to be interesting but not feasible for use by the application developer; the burden becomes overwhelming at times.

This thesis has also sought to highlight that an OpenMP compiler could make use of the hybrid scheme for implementing some particular areas of the OpenMP standard where it would be beneficial to the DSM, notably for some of the data scoping clauses, see Appendix D page 230, and in particular the *reduction* operations that occur at the conclusion of a parallelised region. MPI has such routines, potentially optimised, that can be used to achieve this more efficiently than can be achieved through DSM.

One of the most notable features of this exploratory thesis is that the potentially huge addressable memory space created by a DSM such as SMG is possibly its most valuable

attribute; it will enable scientists to scale up their problems to an extent that is otherwise impossible, i.e. it will raise the bar for tractability. This has been made more possible by the *Dynamic Subscription* protocol, which was developed in the course of this work. A second discovery of some interest is that, for some applications, the DSM performs well in comparison to a real SMP, which could point to good performance when executing on multi-core CPUs. In this case a unified programming model would be presented to the developer. A third discovery, although obvious, is that the DSM performs well on a Grid (albeit simulated) for applications with high computation to communication ratios (i.e. high temporal locality), like the matrix multiply and other linear algebra routines, which are a very important component of many scientific applications.
Grids are starting to impact on mainstream parallel computing. If this trend is set to continue then improved tools and development environments must be implemented. On the occasions where applications execute in a grid environment, and especially so in a multi-site context, they will require better support for monitoring and analysis; this thesis has made a small contribution in this regard. It is anticipated that there will be numerous approaches to constructing grid applications, be it message passing, shared memory, or DSM. Clearly none of these in isolation will provide the perfect fit, but rather an ensemble. Cluster computing has existed for a number of years but in the past decade its popularity has grown considerably, as illustrated in Figure 11.1 (or Table 11.1 for breakrown for June '01 & June '06).
Computer architecture is evolving towards processors with multi-cores that will exploit multi-threaded applications and these are already becoming the predominant processor architecture for cluster computing nodes. Clearly software design will have to adapt to exploit this new technology. Developing efficient applications for this three layer hierarchy ([grid]-[cluster]-[multi-core-processor]) will be a considerable challenge. The imminent inclusion of graphics processing units (GPUs) together with traditional CPU core(s) will further complicate things [179], this is evidenced by the difficulties in utilising the Cell Processor for general purpose scientific applications [180]. Presenting a unified programming model to a developer will be one way to achieve this. If an application could be developed using one programming model and could execute transparently in any manner on this three layer hierarchy then that would be a big step forward. One must believe that some of the first steps have been taken in this thesis.

Architecture	Count	Share %	Processor Totals
Constellations	118/38	23.60/7.60	20,590/46,534
MPP	319/98	63.80/19.60	116,220/414,962
Cluster	32/364	6.40/72.80	14,351/412,367
SMP	31/0	6.20/0	2,906/0
Totals	500	100	152863/873,863

Table 11.1: *Architecture Breakdown of top500 (June-2001/June-2006)*

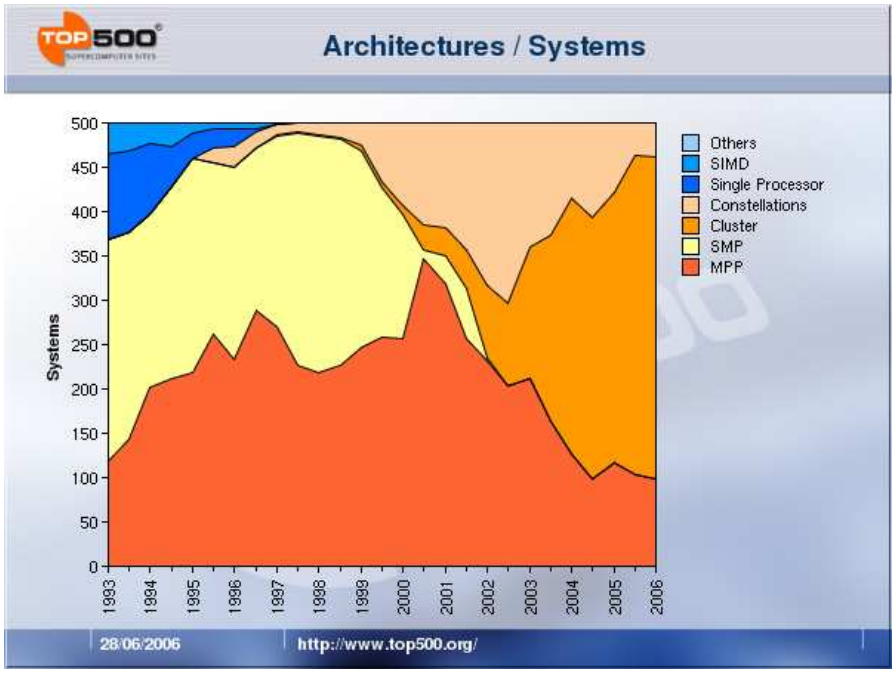


Figure 11.1: *Breakdown of Top 500 computer architecture, ©TOP500.org [166]*

11.4 Conclusions

With the arrival of multi-core processors and the increased performance of machine interconnects the combination of OpenMP and MPI will become irresistible. The scale of applications will expand to take advantage of these new developments, and in consequence it is probable that accessing memory will become more of a bottleneck than it currently is. This thesis has highlighted some of the numerous flaws with DSM and has explored various methods whereby they could be eliminated. For the future of parallel computing it is realistic to conclude that DSM will not prove suitable, but not for the reasons previously published in the literature. In many of these cases either it was inappropriate to attempt parallel computing or unsuitable benchmarks were employed.
It is a contention from the work of this thesis to reaffirm that use of DSM will not become significant in mainstream parallel computing because the classes of suitable applications are too scarce, and those that are are relatively easily ported to message passing, i.e. they have simple sharing patterns. The overheads arising from maintaining consistency can be (relatively) considerable, but not prohibitively so. Additionally the communication overheads with DSM are a factor, but increasingly the memory requirement to support virtual sharing is the real prohibitive factor. The demands for a memory region will be at least three times the actual shared region size. Unfortunately these are not something that can be overcome by using superior coherence protocols like the *Subscription* protocol, or by employing hybrid programming models in a localised fashion.

APPENDIX A

SMG Results

The following tables are presented: Table A.1 to A.3 present various benchmarks for obtaining effective bandwidth and latency information under different conditions for the *IITAC* cluster. Basic benchmarks for fundamental operations that effect the performance of the DSM are given in Table 10.4. After that the performance results are given for the EP, Matrix, Laplace and SOR test applications.
Tests of SMG applications employing user multi-threading (2 threads per process) and involving two processors, which results in each user thread having a dedicated processor, were not performed as this would result in the running of a single SMG process executing on one dual-processor compute node. No communication would result, so experiments of this type were of no interest.

Point-to-point Communication Benchmarks

This section presents three sets of communication benchmarks for the two systems employed for running the experiments in this thesis. The three benchmarks: *pingping*, *pingpong*, and *MPI_Sendrecv* measure different aspects of a point-to-point communication system and are based based on the Intel MPI suite of benchmarks [181]. Collective operations are not of express interest as many MPI libraries compose these operations from point-to-point calls.
pingping measures startup and throughput of single messages, with the crucial difference that messages are obstructed by oncoming messages. For this, two processes communicate (MPI_Isend/MPI_Recv/MPI_Wait) with each other, with the MPI_Isend's issued simultaneously.
pingpong is the classical pattern used for measuring startup and throughput of a single message sent between two processes.
MPI_Sendrecv based on MPI_Sendrecv, the processes form a periodic communication chain. Each process sends to the right and receives from the left neighbour in the chain.

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
8	2.5	3.2
16	5.0	3.2
32	8.7	3.7
64	11.6	5.5
128	13.8	9.2
256	15.3	16.7
512	16.2	31.6
1,024	16.8	61.1
2,048	17.1	119.9
4,096	17.2	238.7
8,192	17.3	473.1
16,384	17.3	948.5
32,768	17.3	1,897.7
65,536	17.3	3,797.0
131,072	17.3	7,591.0
262,144	17.1	15,364.6
524,288	17.1	30,613.5
1048,576	17.2	61,135.7
2097,152	17.2	122,265.2
4194,304	17.2	244,480.3
8,388,608	17.2	488,749.0
16,777,216	17.2	977,995.2
33,554,432	17.1	1,957,098.5
67,108,864	17.2	3,912,441.5
134,217,728	16.7	8,013,714.7

Table A.1: *pingping results for IITAC cluster*

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
8	0.1	56.3
16	0.3	57.0
32	0.6	56.8
64	1.1	57.6
128	2.1	62.0
256	3.9	66.5
512	6.3	81.6
1,024	10.3	99.8
2,048	16.6	123.3
4,096	19.6	208.5
8,192	25.4	322.0
16,384	27.1	604.7
32,768	28.0	1,171.0
65,536	29.1	2,253.7
131,072	29.7	4,413.1
262,144	29.2	8,977.9
524,288	29.6	17,739.5
1,048,576	29.8	35,178.1
2,097,152	29.9	70,189.1
4,194,304	29.9	140,306.9
8,388,608	29.9	280,547.4
16,777,216	29.7	563,985.3
33,554,432	29.9	1,121,106.0
67,108,864	29.8	2,249,938.0
134,217,728	29.9	4,485,807.9

Table A.2: *pingpong results for IITAC cluster*

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
4	0.1	32.1
8	0.2	32.4
16	0.5	32.6
32	1.0	32.8
64	1.9	33.1
128	3.8	34.0
256	6.9	36.8
512	12.1	42.4
1,024	18.8	54.3
2,048	33.1	61.9
4,096	32.6	125.5
8,192	34.1	240.2
16,384	33.2	493.7
32,768	31.7	1,032.7
65,536	31.4	2,084.4
131,072	31.9	4,103.5
262,144	32.0	8,194.0
524,288	32.1	16,321.6
1,048,576	32.4	32,398.3
2,097,152	32.4	64,731.7
4,194,304	32.5	129,060.3
8,388,608	32.5	258,196.0
16,777,216	32.3	520,100.1
33,554,432	32.4	1,034,866.8
67,108,864	32.4	2,073,620.3
134,217,728	32.3	4,152,477.4

Table A.3: *MPI_Sendrecv benchmark for IITAC cluster*

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
8	0.6	13.1
16	3.6	4.5
32	6.3	5.1
64	13.9	4.6
128	20.9	6.1
256	33.3	7.7
512	41.3	12.4
1024	43.1	23.8
2,048	45.1	45.4
4,096	47.8	85.7
8,192	45.5	180.0
16,384	46.6	351.7
32,768	45.2	725.7
65,536	47.6	1,376.3
131,072	43.0	3,045.2
262,144	44.0	5,959.9
524,288	44.0	11,906.4
1,048,576	44.5	23,548.4
2,097,152	50.1	41,873.3
4,194,304	43.4	96,611.4
8,388,608	44.8	187,207.8
16,777,216	46.0	364,866.7
33,554,432	45.5	737,815.0
67,108,864	53.6	1,252,040.5
134,217,728	56.7	2,366,012.0

Table A.4: *pingping results for Molch cluster*

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
8	0.1	86.4
16	0.2	81.1
32	0.4	84.9
64	0.8	85.2
128	1.4	88.3
256	2.6	97.2
512	5.1	100.9
1,024	7.8	130.6
2,048	13.1	156.2
4,096	19.9	205.6
8,192	29.4	278.6
16,384	35.5	462.1
32,768	40.8	802.5
65,536	47.8	1,370.8
131,072	49.8	2,633.2
262,144	45.5	5,761.8
524,288	47.3	11,093.9
1,048,576	43.5	24,128.5
2,097,152	50.0	41,949.5
4,194,304	61.8	67,921.6
8,388,608	64.4	130,311.3
16,777,216	47.1	355,886.3
33,554,432	47.6	704,317.8
67,108,864	50.5	1,328,461.9
134,217,728	90.8	1,477,681.6

Table A.5: *pingping results for Molch cluster*

Message Size (Bytes)	Bandwidth (MB/s)	Latency (us)
8	0.2	35.2
16	0.5	35.3
32	1.0	33.6
64	1.6	38.8
128	3.5	36.2
256	7.0	36.5
512	10.5	48.8
1,024	17.4	58.7
2,048	25.8	79.3
4,096	44.9	91.3
8,192	70.4	116.3
16,384	96.5	169.7
32,768	107.8	304.0
65,536	82.5	794.2
131,072	89.6	1,462.3
262,144	112.6	2,327.4
524,288	116.8	4,489.9
1,048,576	98.2	10,676.7
2,097,152	100.3	20,913.0
4,194,304	116.1	36,134.4
8,388,608	109.7	76,456.9
16,777,216	87.3	192,232.5
33,554,432	87.1	385,124.6
67,108,864	91.2	735,545.4
134,217,728	88.2	1,522,441.8

Table A.6: *MPI_Sendrecv benchmark for Molch cluster*

EP

No experiment has been performed for the threaded variant of the EP application for a processor count = 2, as this would result in the two processors being allocated to just one process, no inter-process communication, and so not of interest. So suitable candidates were identified where hybridisation would be beneficial.

No. Procs	2	4	8	16	32	64	128
Time (s)	7719.5	3839.48	1919.69	1064.56	551.36	286.44	179.13
Messages	1	3	7	15	31	63	127
Data (bytes)	96	288	672	1440	2976	6048	12192

Table A.7: *EP using MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	8182.69	4099.77	2048.77	1031.17	525.41	279.48	176.43
Messages	2	18	42	90	186	378	762
Data (bytes)	64	2624	9052	31520	112756	420212	1598276
DSM pagefaults	2	4	8	16	32	64	128

Table A.8: *EP SMG update protocol*

No. Procs	2	4	8	16	32	64	128
Time (s)	8210.49	4116.82	2186.22	1036.08	563.37	299.22	204.18
Messages	2	18	42	90	186	378	762
Data (bytes)	144	2784	9372	32128	114036	422708	1603396
DSM pagefaults	4	8	16	32	64	128	256

Table A.9: *EP using SMG with subscription*

No. Procs	2	4	8	16	32	64	128
Time (s)	X	4092.95	2046.49	1028.76	546.51	276.81	177.42
Messages	X	2	18	42	90	186	378
Data (bytes)	X	64	2624	9050	31520	112756	420210

Table A.10: *EP using SMG with multiple user threads*

No. Procs	16	32	64	128
Time (s)	1033.13	555.97	292.88	183.51
Messages - Intra	72	168	360	744
Messages - Inter	18	18	18	18
Data - Intra (bytes)	2.385E+4	9.803E+4	3.910E+5	1.540E+6
Data - Inter (bytes)	7.092E+3	1.295E+4	2.477E+4	4.808E+4

Table A.11: *EP using SMG in a simulated Grid with information*

No. Procs	16	32	64	128
Time (s)	1097.47	559.41	301.51	198.64
Messages - Intra	66	162	354	738
Messages - Inter	24	24	24	24
Data - Intra (bytes)	2.20E+4	9.518E+4	3.864E+5	1.532E+6
Data - Inter (bytes)	9.524E+3	1.758E+4	3.383E+4	6.589E+4

Table A.12: *EP using SMG in a simulated Grid without information*

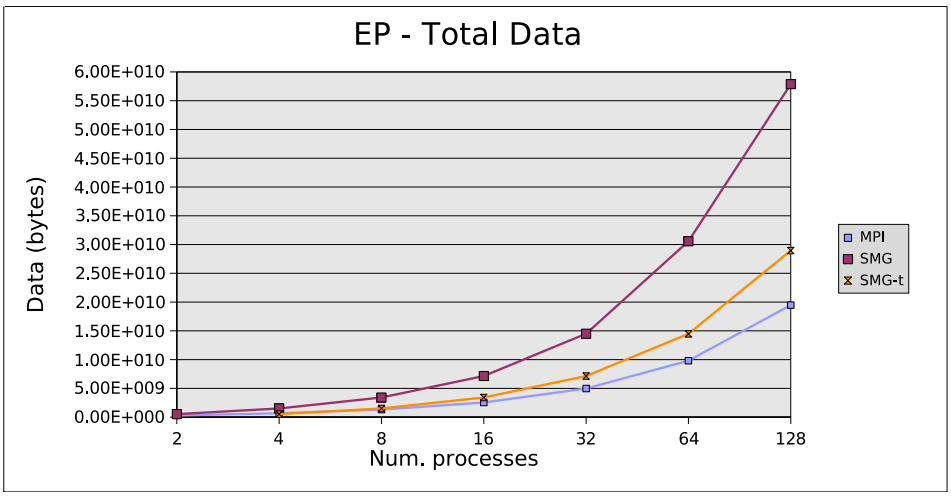


Figure A.1: *EP (with user multi-threads) - Total Data sent*

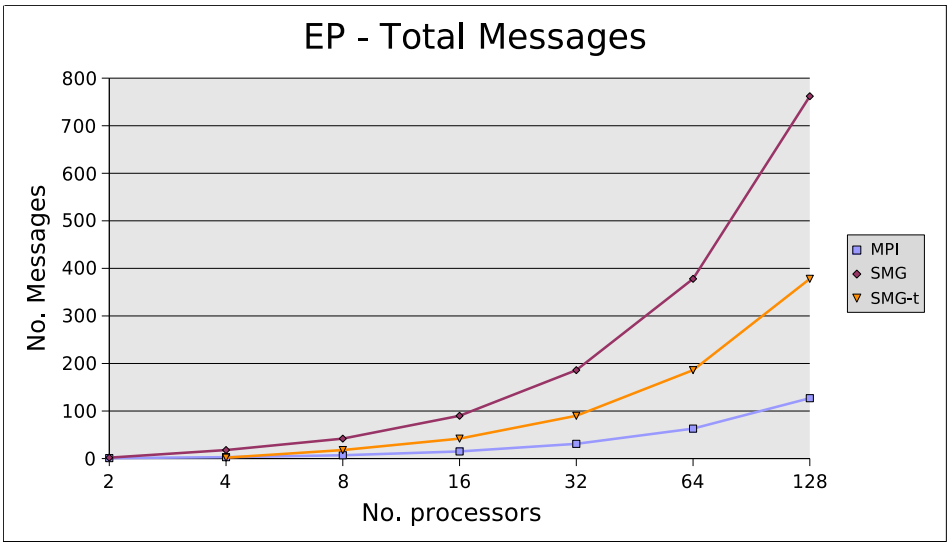


Figure A.2: *EP (with user multi-threads) - Total Messages sent*

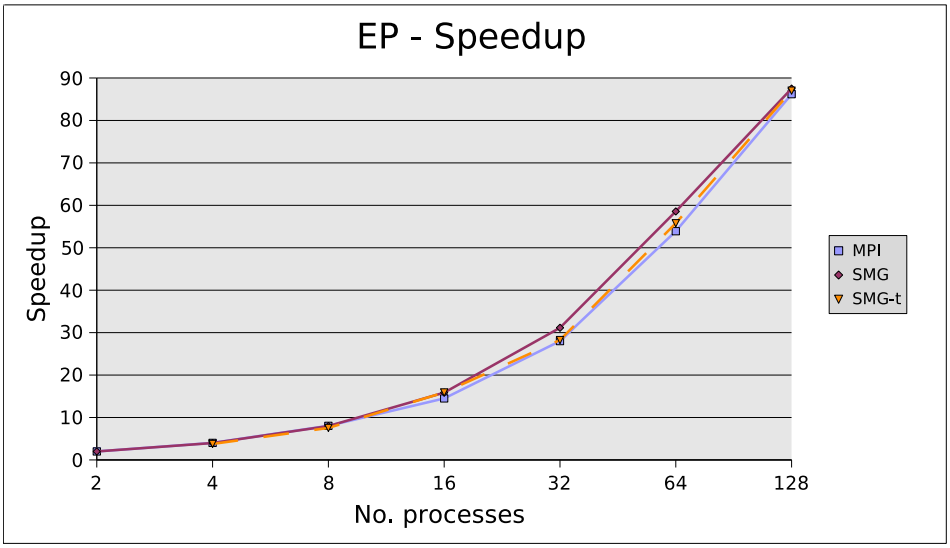


Figure A.3: *EP (with user multi-threads) - Speedup*

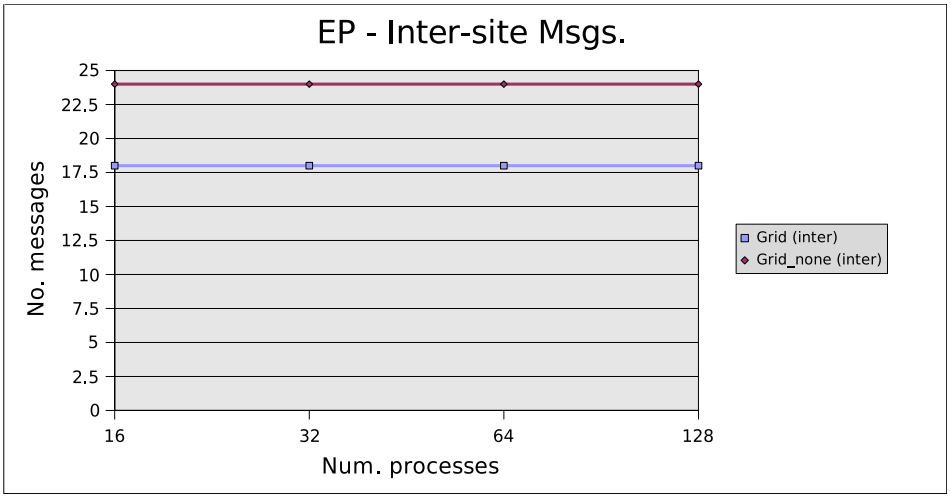


Figure A.4: *EP - Total inter-site message count*

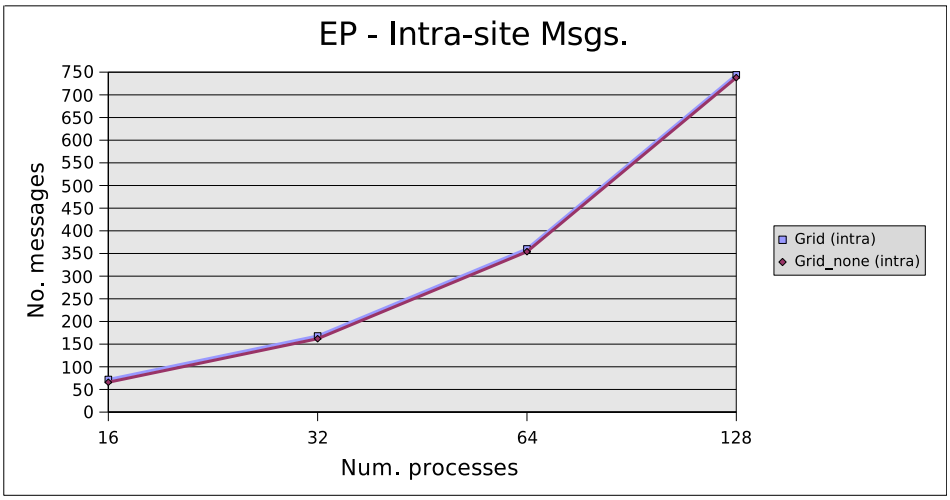


Figure A.5: *EP - Total intra-site message count*

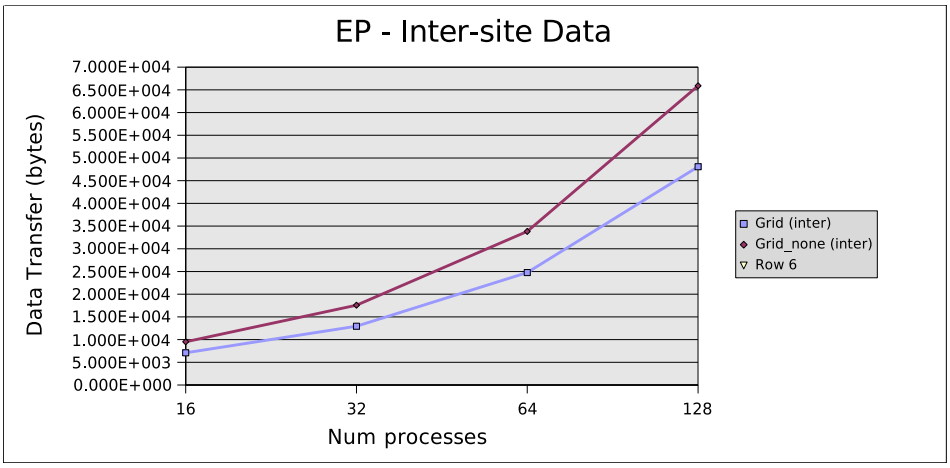


Figure A.6: *EP - Total inter-site data volumes*

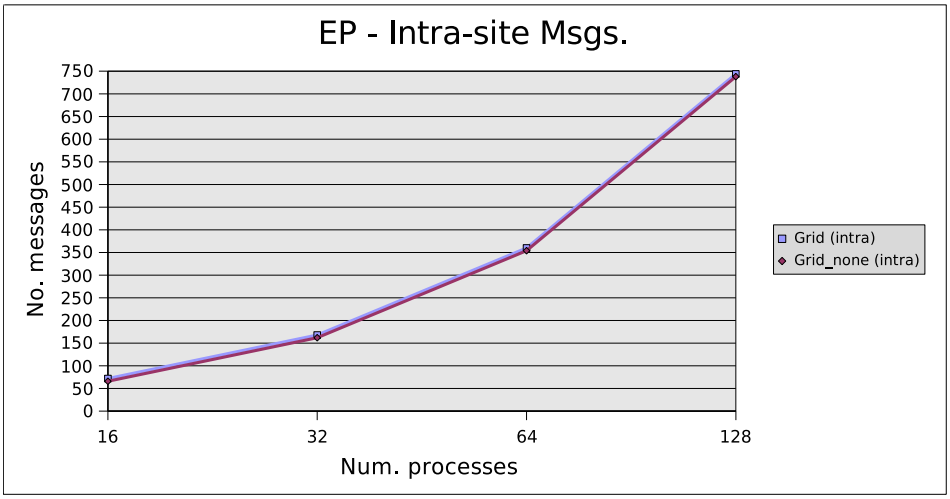


Figure A.7: *EP - Total intra-site data volume*

Matrix

No experiment has been performed for the threaded variant of the Matrix application for a processor count = 2, as this would result in one process, and so not of interest.

No. Procs	2	4	8	16	32	64	128
Time (s)	17547.41	8475.97	4383.68	2241.89	1141.10	527.51	308.39
Messages	3	9	21	45	93	189	379
Data (bytes)	3.2E+8	6.79E+8	1.32E+9	2.55E+9	4.97E+9	9.81E+9	1.95E+10

Table A.13: *Matrix Multiplication implemented using MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	17661.98	8849.21	4429.98	2048.77	1031.17	629.82	339.10
Messages	8	24	56	120	248	504	1016
Data (bytes)	5.28E+8	1.51E+9	3.41E+9	7.15E+9	1.45E+10	3.06E+10	5.79E+10
DSM pagefaults	110580	110568	110544	110496	110400	110208	109824

Table A.14: *Matrix SMG update protocol*

No. Procs	2	4	8	16	32	64	128
Time (s)	17661.71	8841.66	4426.36	2226.44	1136.03	622.30	325.18
Messages	7	21	49	105	217	252	889
Data (bytes)	3.02E+8	9.81E+8	2.25E+9	4.67E+9	9.51E+9	1.92E+10	3.85E+10

Table A.15: *Matrix using a hybrid of SMG & MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	17804.56	8868.76	4479.36	2255.27	1173.20	630.44	327.80
Messages	8	24	56	120	248	504	1016
Data (bytes)	5.28E+8	1.51E+9	3.41E+9	7.15E+9	1.45E+10	3.06E+10	5.79E+10
DSM pagefaults	184320	313344	599055	1184286	2361662	4719864	9438013

Table A.16: *Matrix using SMG with subscription*

No. Procs	2	4	8	16	32	64	128
Time (s)	X	8825.33	4187.52	2140.74	1097.02	603.42	327.80
Messages	X	8	24	56	120	248	500
Data (bytes)	X	5.28E+8	1.51E+9	3.41E+9	7.14E+9	1.45E+10	2.89E+10

Table A.17: *Matrix using SMG with multiple user threads*

No. Procs	16	32	64	128
Time (s)	2238.52	1142.93	614.80	340.37
Messages - Intra	96	224	480	992
Messages - Inter	24	24	24	24
Data - Intra (bytes)	5.582E+9	1.291E+10	2.744E+10	5.626E+10
Data - Inter (bytes)	1.508E+9	1.507E+09	1.504E+09	1.497E+09

Table A.18: *Matrix using SMG in a simulated Grid with information*

No. Procs	16	32	64	128
Time (s)	2374.99	1192.24	641.77	379.25
Messages - Intra	88	216	472	984
Messages - Inter	32	32	32	32
Data - Intra (bytes)	5.129E+9	1.248E+10	2.703E+10	5.586E+10
Data - Inter (bytes)	2.017E+9	2.025E+9	2.025E+9	2.019E+9

Table A.19: *Matrix using SMG in a simulated Grid without information*

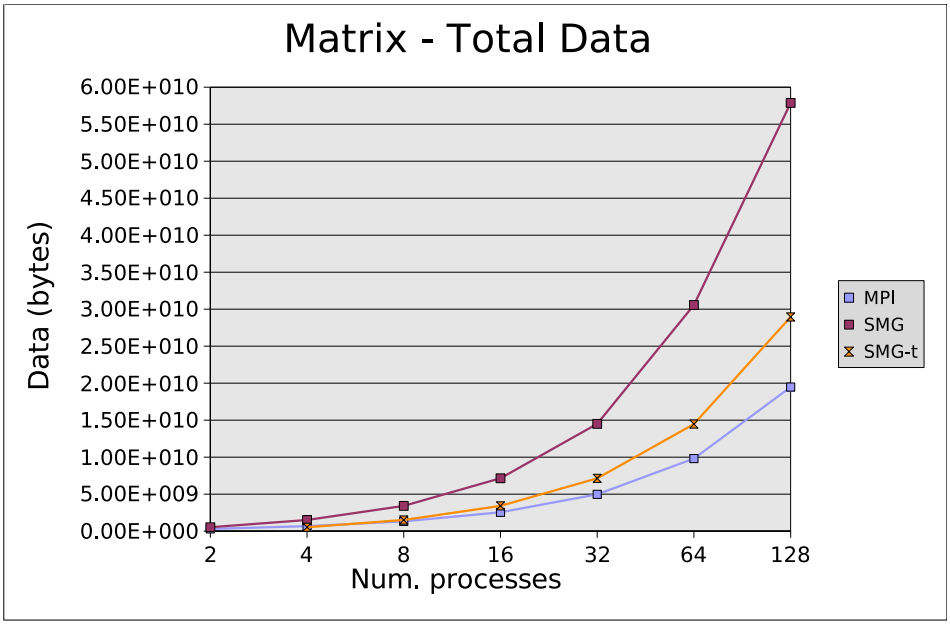


Figure A.8: *Matrix (with user multi-threads) - Total Data sent*

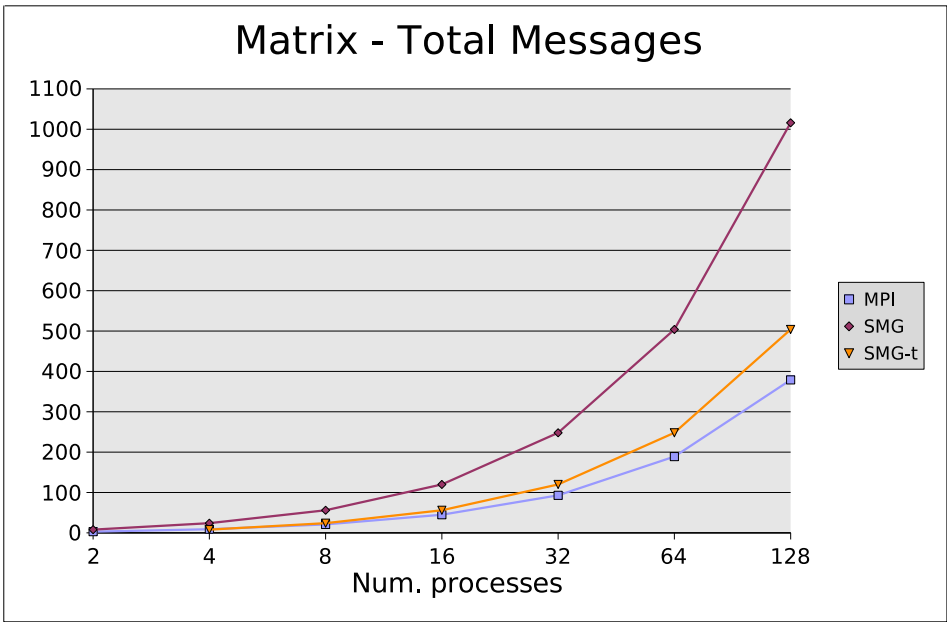


Figure A.9: *Matrix (with user multi-threads) - Total Messages sent*

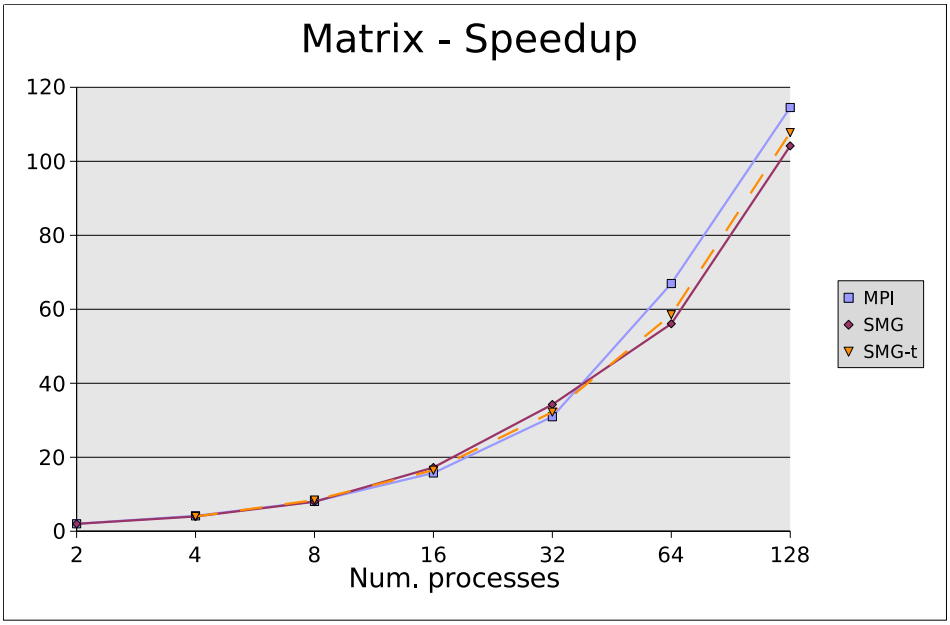


Figure A.10: *Matrix (with user multi-threads) - Speedup*

Laplace

No. Procs	2	4	8	16	32	64	128
Time (s)	86.55	53.58	35.48	25.92	19.85	18.82	25.28
Messages	203	609	1421	3045	6293	12789	25781
Data (bytes)	3.12E+8	9.36E+8	1.32E+9	2.55E+9	4.97E+9	9.81E+09	3.96E+10

Table A.20: *Laplace implemented using MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	1099.56	1833.80	2775.02	3650.29	4471.79	5638.20	6045.73
Messages	206	618	1442	3090	6386	12978	26162
Data (bytes)	6.63E+10	9.65E+9	1.71E+10	3.99E+10	7E+10	2.98E+12	5.88E+12
DSM pagefaults	7518034	7518236	7518640	7519448	7521064	7524296	7530760

Table A.21: *Laplace SMG update protocol*

No. Procs	2	4	8	16	32	64	128
Time (s)	90.65	55.72	60.44	70.47	86.77	109.95	137.27
Messages	208	624	1456	3120	6448	13104	26416
Data (bytes)	7.64E+8	2.44E+09	5.39E+9	1.08E+10	2.11E+10	4.12E+10	8.04E+10

Table A.22: *Laplace using a hybrid of SMG & MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	96.01	70.26	46.76	46.53	66.46	90.07	101.77
Messages	206	618	1442	3090	6386	12978	26162
Data (bytes)	3.19E+8	9.46E+8	2.28E+9	5.17E+9	1.19E+10	2.87E+10	7.65E+10
DSM pagefaults	9095369	9095611	9096096	9097066	9099005	9102883	9110640

Table A.23: *Laplace using SMG with subscription*

No. Procs	2	4	8	16	32	64	128
Time (s)	X	1498.71	2249.61	2789.54	3716.10	4339.80	5264.85
Data (bytes)	X	6.63E+10	1.64E+11	3.23E+11	6.4E+11	1.32E+12	2.66E+12
Messages	X	206	618	1442	3090	6386	12978

Table A.24: *Laplace using SMG with multiple user threads*

No. Procs	16	32	64	128
Time (s)	3847.09	5415.66	6021.69	6922.78
Messages - Intra	2472	5768	12360	25544
Messages - Inter	618	618	618	618
Data - Total (bytes)	7.53E+11	1.49E+12	2.95E+12	5.84E+12
Data - Intra (bytes)	5.761E+11	1.314E+12	2.769E+12	5.661E+12
Data - Inter (bytes)	1.772E+11	1.773E+11	1.775E+11	1.780E+11

Table A.25: *Laplace using SMG in a simulated Grid with information*

No. Procs	16	32	64	128
Time (s)	3921.83	5539.42	6204.94	7111.77
Messages - Intra	2266	5562	12154	25338
Messages - Inter	824	824	824	824
Data - Total (bytes)	7.70E+11	1.52E+12	2.98E+12	5.88E+12
Data - Intra (bytes)	5.320E+11	1.276E+12	2.737E+12	5.633E+12
Data - Inter (bytes)	2.381E+11	2.410E+11	2.426E+11	2.439E+11

Table A.26: *Laplace using SMG in a simulated Grid without information*

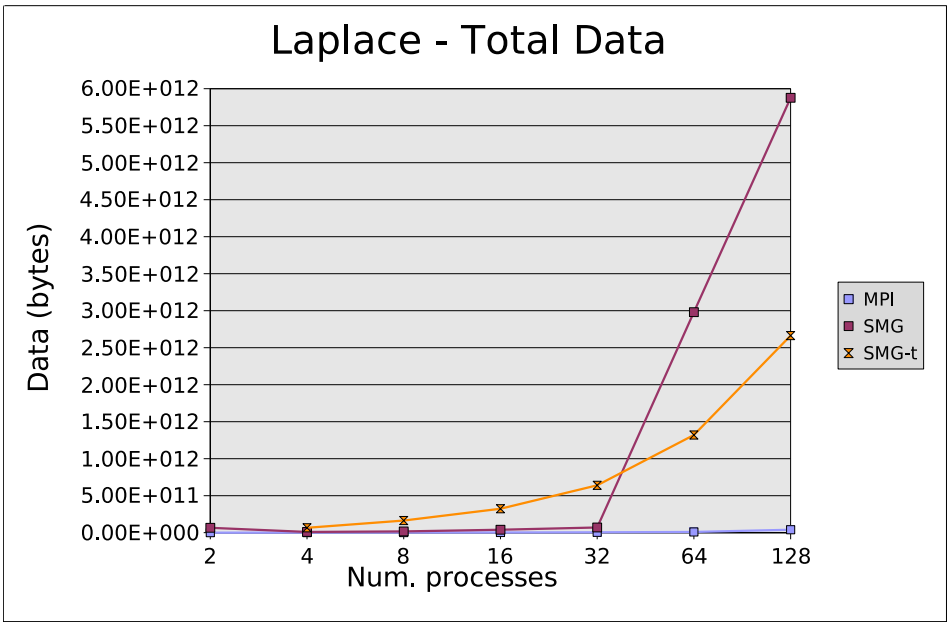


Figure A.11: *Laplace (with user multi-threads) - Total Data sent*

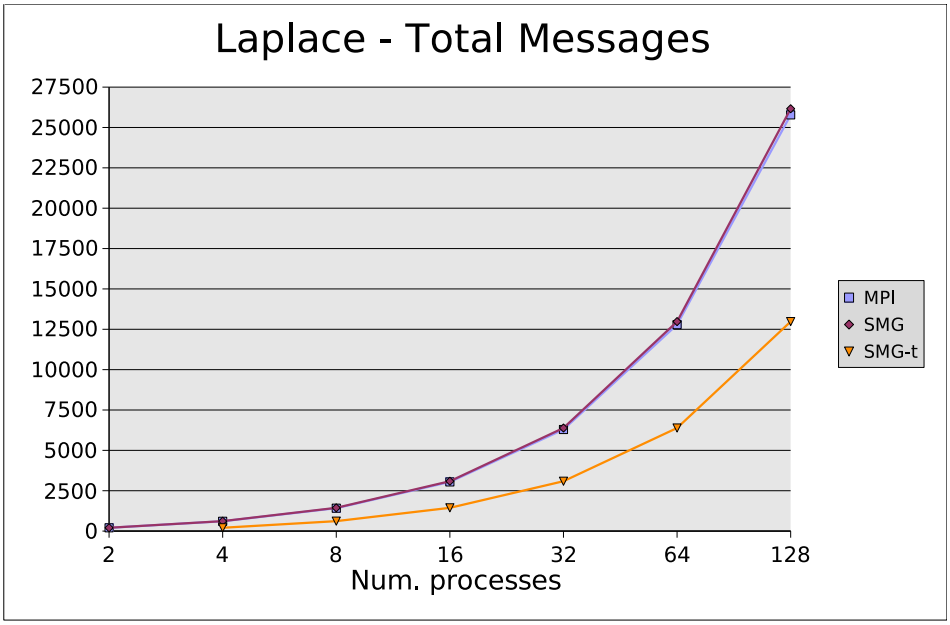


Figure A.12: *Laplace (with user multi-threads) - Total Messages sent*

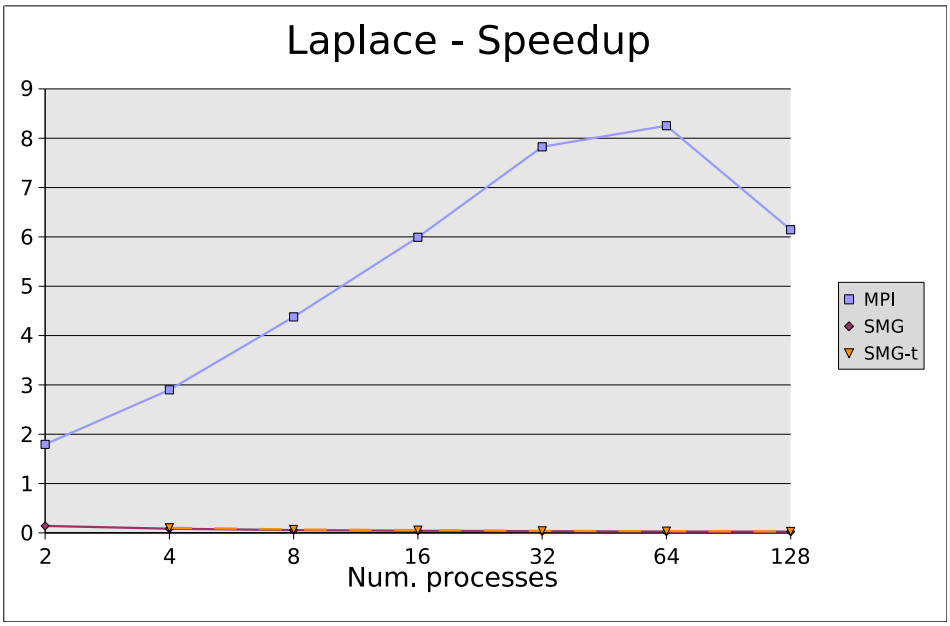


Figure A.13: *Laplace (with user multi-threads) - Speedup*

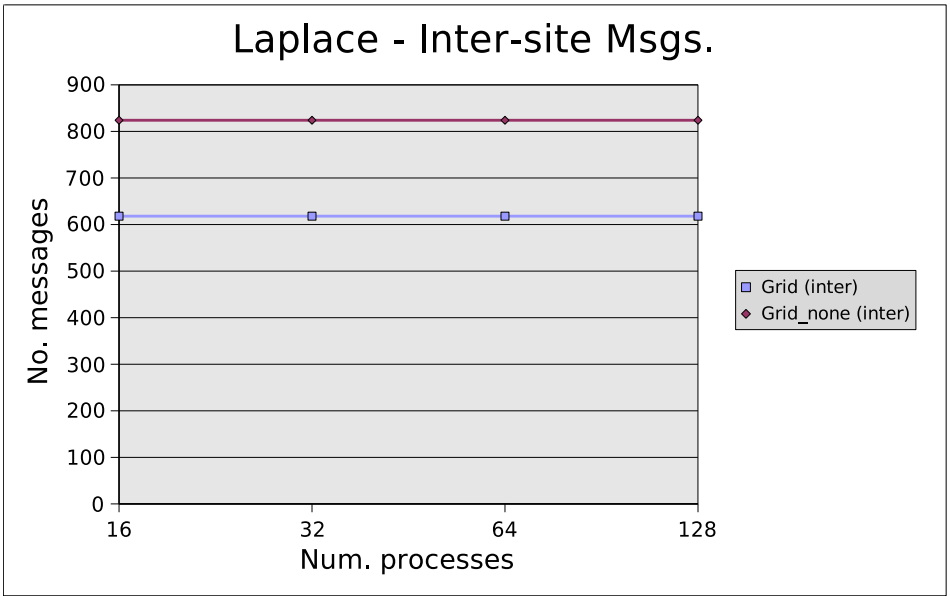


Figure A.14: *Laplace - Total inter-site message count*

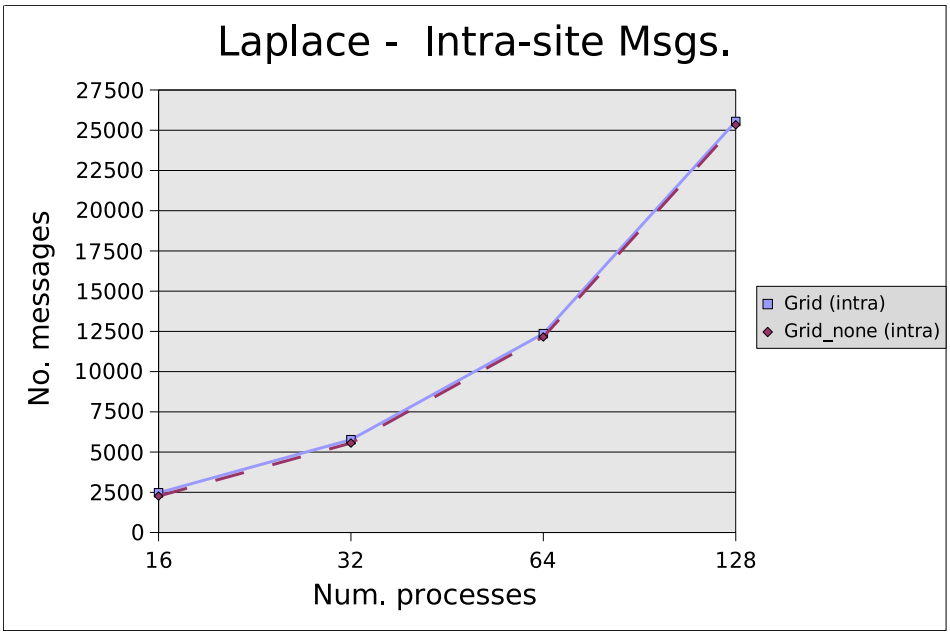


Figure A.15: *Laplace - Total intra-site message count*

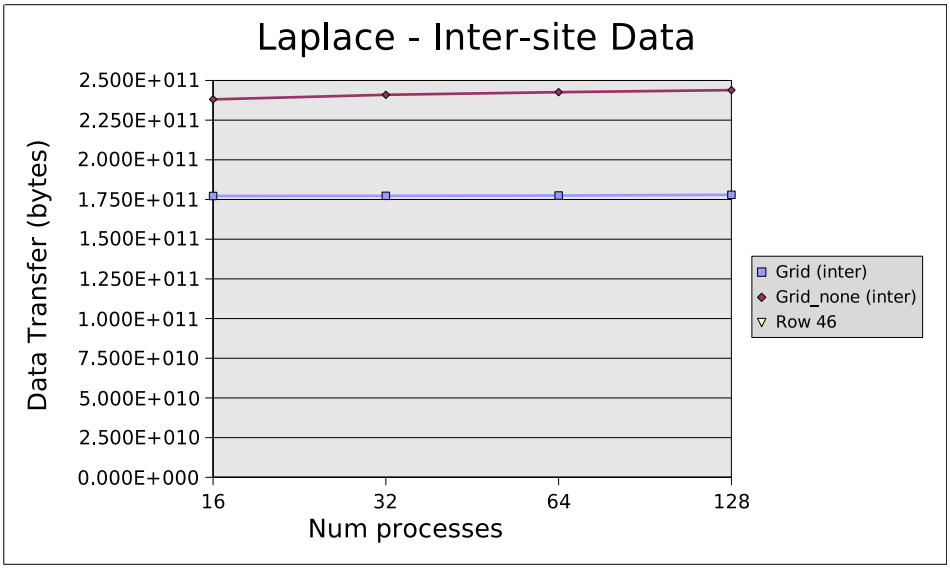


Figure A.16: *Laplace - Total inter-site data volumes*

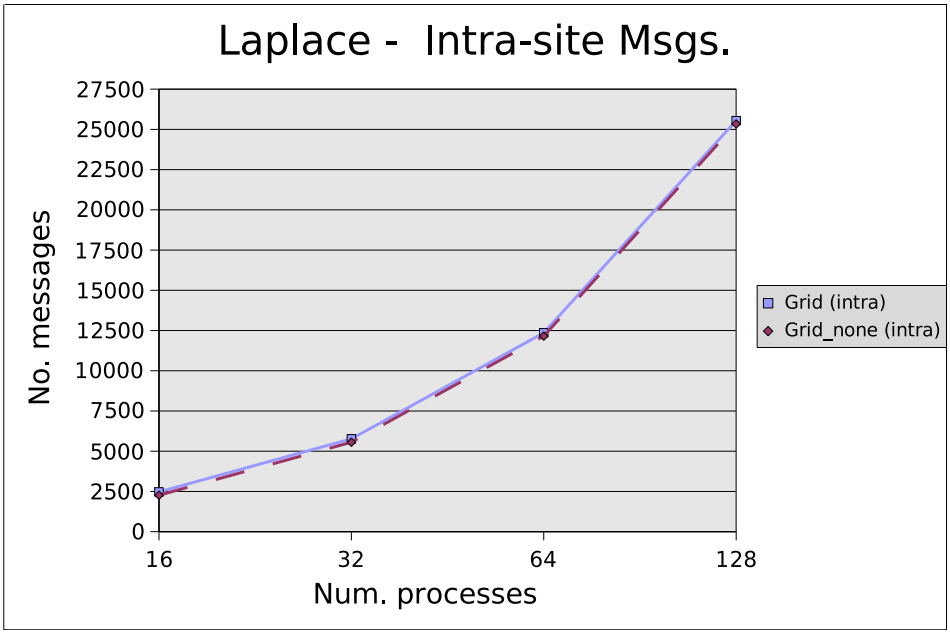


Figure A.17: *Laplace - Total intra-site data volume*

SOR

No experiment has been performed for the threaded variant of the Matrix application for a processor count = 2, as this would result in one process, and so not of interest.

No. Procs	2	4	8	16	32	64	128
Time (s)	63.19	34.35	23.18	17.51	18.35	28.14	54.85
Messages	602	1604	3608	7616	15632	31664	63728
Data (bytes)	3.21E+8	3.61E+8	4.39E+8	5.96E+8	9.10E+8	1.54E+9	2.79E+9

Table A.27: *SOR implemented using MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	235.49	146.40	111.84	95.83	106.83	117.54	138.34
Messages	424	1272	2968	6360	13144	26712	53848
Data (bytes)	6.54E+8	1.89E+09	4.06E+9	8.03E+9	1.56E+10	3.04E+10	5.95E+10
DSM pagefaults	15478480	15478880	15479680	15481280	15484480	15490880	15503680

Table A.28: *SOR SMG update protocol*

No. Procs	2	4	8	16	32	64	128
Time (s)	68.78	55.75	54.54	52.92	61.46	65.50	101.35
Messages	424	1272	2968	6360	13144	26712	53848
Data (bytes)	3.62E+8	7.43E+8	1.59E+9	3.14E+9	6.11E+9	1.18E+10	2.29E+10

Table A.29: *SOR using a hybrid of SMG & MPI*

No. Procs	2	4	8	16	32	64	128
Time (s)	62.74	41.21	42.45	44.32	61.11	74.31	99.76
Messages	424	1272	2968	6360	13144	26712	53848
Data (bytes)	4.09E+7	5.40E+8	6.40E+08	1.18E+09	3.34E+09	7.71E+09	1.63E+10
DSM pagefaults	18721632	18722112	18723072	18651264	18728832	18736512	18751872

Table A.30: *SOR using SMG with subscription*

No. Procs	2	4	8	16	32	64	128
Time (s)	X	211.99	144.12	116.01	126.50	125.59	175.48
Messages	X	466	1398	3262	6990	14446	29358
Data (bytes)	X	8.32E+8	2.19E+9	4.58E+9	9.32E+9	1.98E+10	3.93E+10

Table A.31: *SOR using SMG with multiple user threads*

No. Procs	16	32	64	128
Time (s)	104.84	115.96	119.47	151.20
Messages - Total	6360	13144	26712	53848
Messages - Intra	5088	11872	25440	52576
Messages - Inter	1272	1272	1272	1272
Data - Intra (bytes)	5.925E+9	1.346E+10	2.819E+10	5.728E+10
Data - Inter (bytes)	1.894E+9	1.894E+9	1.894E+9	1.895E+9

Table A.32: *SOR using SMG in a simulated Grid with information*

No. Procs	16	32	64	128
Time (s)	109.66	123.19	142.86	166.38
Messages Total	6360	13144	26712	53848
Messages - Intra	4664	11448	25016	52152
Messages - Inter	1696	1696	1696	1696
Data - Intra (bytes)	5.615E+9	1.318E+10	2.794E+10	5.705E+10
Data - Inter (bytes)	2.419E+9	2.432E+9	2.439E+9	2.443E+9

Table A.33: *SOR using SMG in a simulated Grid without information*

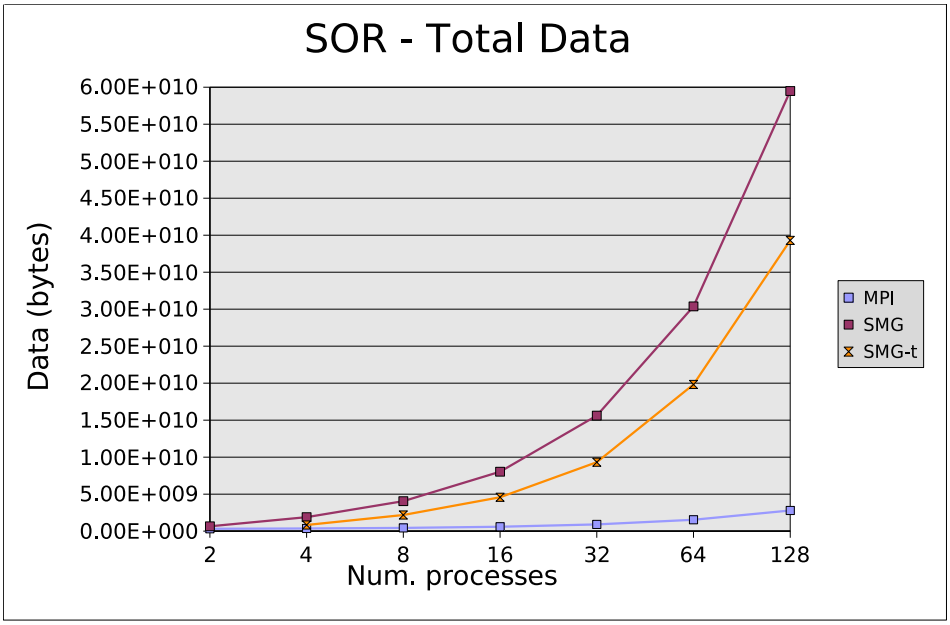


Figure A.18: *SOR (with user multi-threads) - Total Data sent*

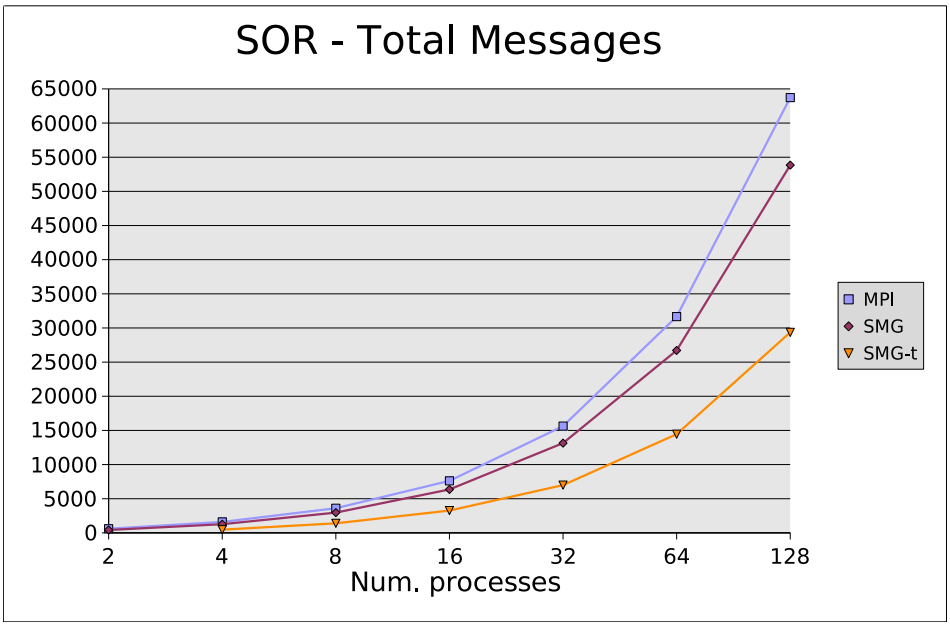


Figure A.19: *SOR (with user multi-threads) - Total Messages sent*

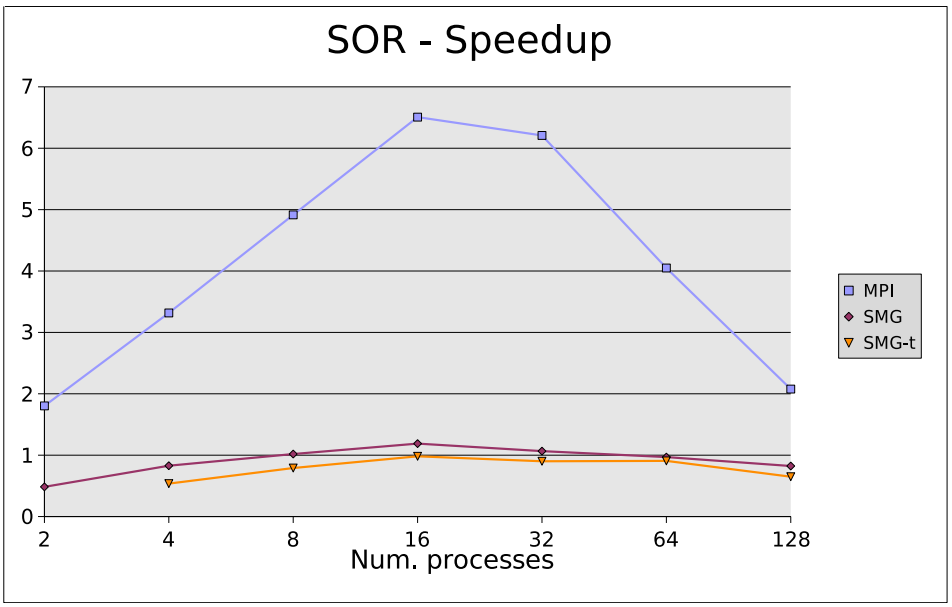


Figure A.20: *SOR (with user multi-threads) - Speedup*

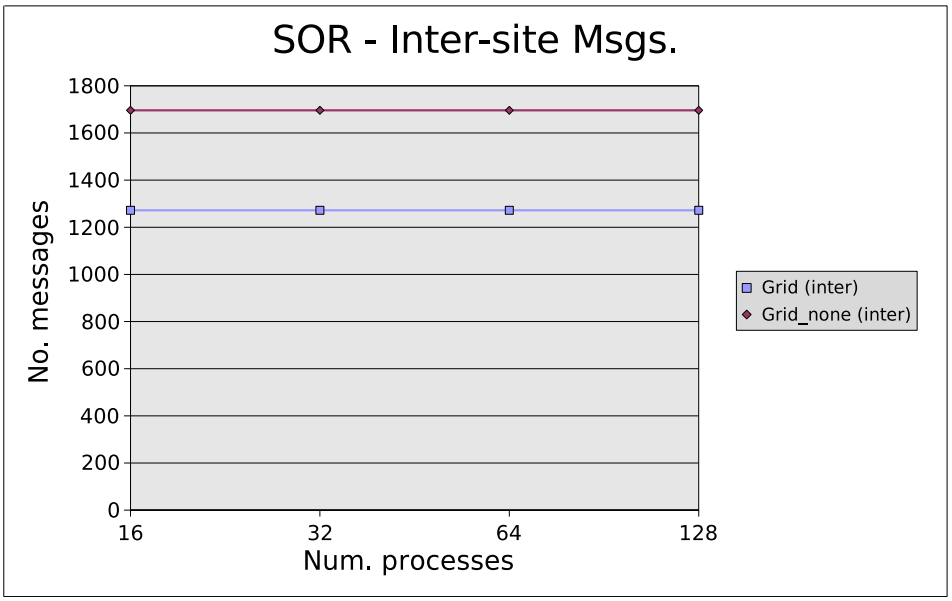


Figure A.21: *SOR - Total inter-site message count*

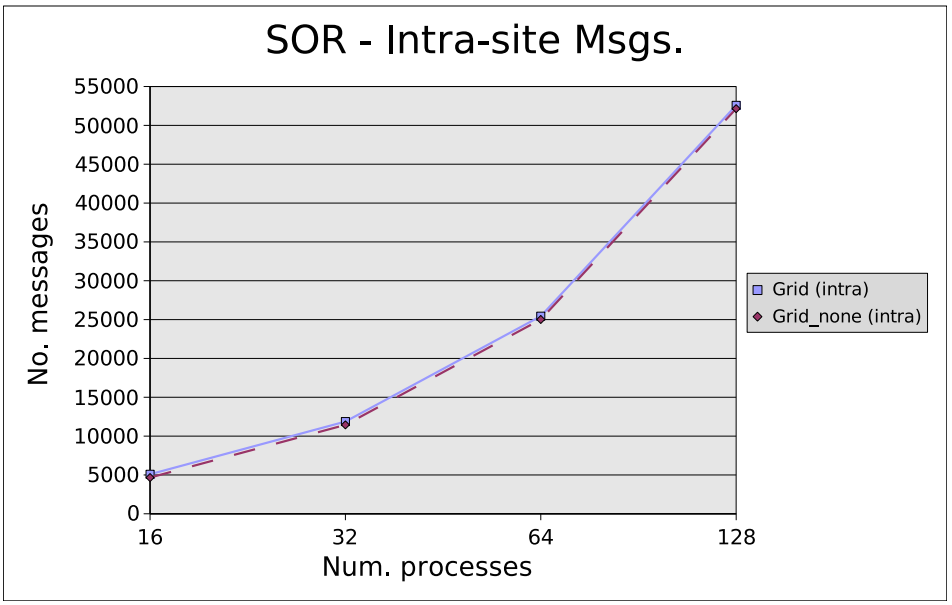


Figure A.22: *SOR - Total intra-site message count*

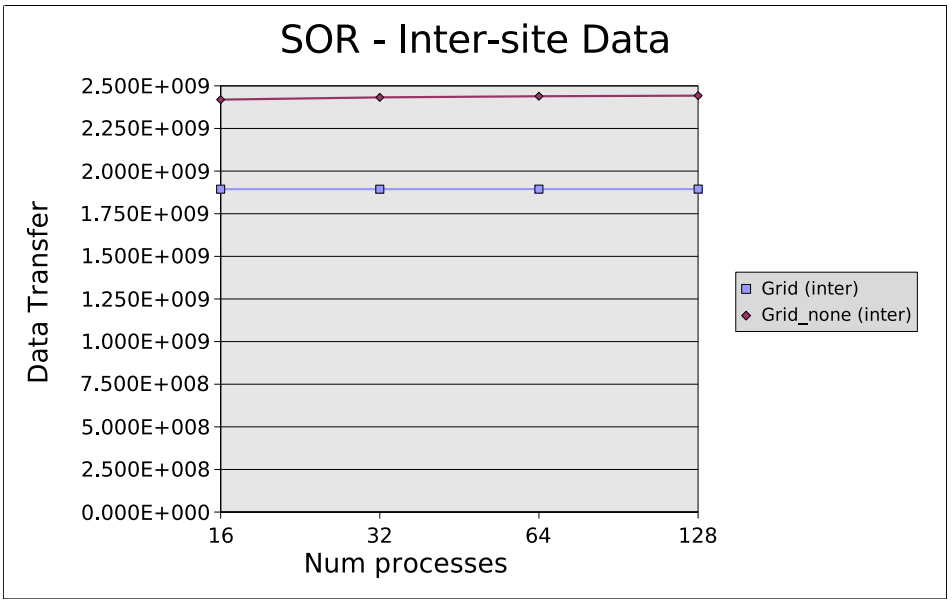


Figure A.23: *SOR - Total inter-site data volumes*

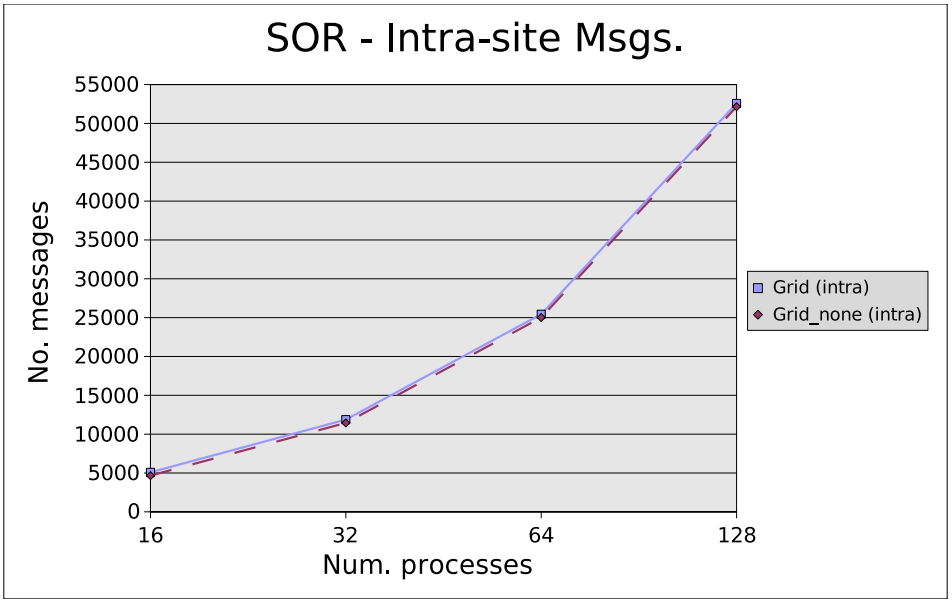


Figure A.24: *SOR - Total intra-site data volume*

APPENDIX B

DSM Reference

This Appendix gives a synopsis of the characteristics of various DSM implementations.

Name	*Type of Algorithm*	*Consistency Model*	*Granularity Unit*	*Coherence Policy*
Dash	MRSW	release	16 bytes	invalidate
DDM	MRSW	sequent16	bytes	invalidate
KSR1	MRSW	sequential	128 bytes	invalidate
Memnet	MRSW	sequential	32 bytes	invalidate
Merlin	MRMW	processor	8 bytes	update
RMS	MRMW	processor	4 bytes	update
SCI	MRSW	sequential	16 bytes	invalidate

Table B.1: *Hardware DSM implementations*

Name	*Type of Algorithm*	*Consistency Model*	*Granularity Unit*	*Coherence Policy*
Alewife	MRSW	sequential	16 bytes	invalidate
FLASH	MRSW	release	128 bytes	invalidate
Galactica Net	MRMW	multiple	8K bytes	update/invalidate
Hybrid DSM	MRSW	release	variable	invalidate
PLUS	MRMW	processor	4K bytes	update
SHRIMP	MRMW	AURC,scope	4Kbytes	update/invalidate
Typhoon	MRSW	custom	32 bytes	invalidate custom

Table B.2: *Hybrid DSM implementations*

Name	*Type of Algorithm*	*Consistency Model*	*Granularity Unit*	*Coherence Policy*
Blizzard	MRSW	sequential	32-128 byte	invalidate
Cashmere-2L	MRMW, LRC variant	invalidate	page	
Clouds	MRSW	inconsistent, sequential	8Kbyte	inval when unlocked
IVY	MRSW	Sequential	1Kbyte	invalidate
Jiajia	MRMW	LRC	Page	Home-based Invalidate
Linda	MRSW	sequential	variable (tuple size)	impl. dependent
Mermaid	MRSW	Sequential	1Kbyte, 8Kbyte	invalidate
Midway	MRMW	entry, release, processor	4Kbyte	update
Mirage	MRSW	sequential	512 byte	invalidate
Munin	SRSW,MRSW	release	Variable	delayed update, invalidate
Orca	MRSW	sync dependent	shared object size	update
TreadMarks	MRMW	lazy release	4Kbyte	update, invalidate

Table B.3: *Software DSM implementations*

APPENDIX C

DSM APIs

The various APIs of DSMs referred to in this thesis are listed in this Appendix.

Midway API

```
/* initialize, acquire, and release a lock primitive */
extern midway_lock_t midway_lock_alloc()
extern void midway_lock_acq(midway_lock_t lockn,
                            midway_acquire_t mode);
extern void midway_lock_rel(midway_lock_t lockn)


/* Barrier synchronisation routines */
extern midway_barrier_t midway_barrier_alloc()
extern void midway_cross_barrier(midway_barrier_t bname, int num)
extern void midway_flush_barrier(midway_barrier_t bname, int num)


/* bind shared memory to sync objects*/
extern void midway_bind_synch_object(midway_lock_t obj,void *addr,
                        u_long len)
extern void midway_rebind_synch_object( midway_lock_t obj)
extern void midway_create_thread( void(*func)(), u_long arg)
extern void midway_wait_for_end(u_long n)


/* Memory management */
extern shared char *shmalloc(unsigned n, unsigned linesz)
extern void shfree(shared char *mem)


/* DSM system routines */
extern int midway_start_profile (void);
extern int midway_stop_profile (void);
extern void midway_exit(int rc);
```

Munin API

```
/* Initialisation & finalisation*/
user_init()   // User implements these functions if required
user_done()

/* Synchronisation routines */
AcquireLock()
ReleaseLock()
CreateBarrier()
WaitAtBarrier()
WaitCondition()
SignalCondition()
BroadcastCondition()

/* Shared memory operations */
AssociateDataAndSynch()
PhaseChange()
ChangeAnnotation()
PreAcquire()
Invalidate()
SingleObject()
```

CRL API

```
/* Initialisation & finalisation*/
extern void}     crl_init(char *);
extern void}     crl_exit(void);

/* Memory management */
extern rid_t  rgn_create(unsigned);
extern void     rgn_destroy(rid_t);
extern void    *rgn_map(rid_t);
extern void     rgn_unmap(void *);
extern rid_t   rgn_rid(void *);
extern unsigned rgn_size(void *);

/* Memory access operations */
extern void     rgn_start_read(void *);
extern void     rgn_end_read(void *);
extern void     rgn_start_write(void *);
extern void     rgn_end_write(void *);
extern void     rgn_flush(void *);

/* Collective operations */
extern void     rgn_barrier(void);
extern void     rgn_bcast_send(int, void *);
extern void     rgn_bcast_recv(int, void *);
extern double   rgn_reduce_dadd(double);
extern double   rgn_reduce_dmin(double);
extern double   rgn_reduce_dmax(double);
```

Treadmarks API

```
/* Application variables */
extern unsigned Tmk_nprocs
extern unsigned Tmk_proc_id

/* Initialisation & finalisation*/
void Tmk_startup(int argc, char **argv)
void Tmk_exit(int atatus)

/* Synchronisation routines */
void Tmk_barrier(unsigned id)
void Tmk_lock_acquire(unsigned id)
void Tmk_lock_release(unsigned id)
void Tmk_lock_cond_broadcast( unsigned lock_id, unsigned cond_id );
void Tmk_lock_cond_signal( unsigned lock_id, unsigned cond_id );
void Tmk_lock_cond_wait( unsigned lock_id, unsigned cond_id );

/* Memory management */
char *Tmk_malloc(unsigned size)
void Tmk_free(char *ptr)
```

CVM API

```
/* Initialisation & finalisation*/
void cvm_startup();
void cvm_create_procs(void *func());
void cvm_exit(char *,...);
void cvm_finish();

/* Environment */
int cvm_pid();
int cvm_tid();

/* Memory management */
void *cvm_alloc(int sz);
void *cvm_alloc_char(int sz);
void *cvm_alloc_short(int sz);
void *cvm_alloc_int(int sz);
void *cvm_alloc_float(int sz);
void *cvm_alloc_double(int sz);
void *cvm_typed_alloc(int sz, ...);

/* Synchronisation routines */
void cvm_lock(int id);
void cvm_unlock(int id);
void cvm_barrier(int id);
void cvm_probe();

/* Shared memory environment */
extern void cvm_assign_ownership(int pid, char *from, int sz);
extern void cvm_distribute_ownership(char *from, int sz, int how);
```

APPENDIX D

Enabling Technology

This Appendix briefly summarises technologies relevant to this thesis.

Message Passing Interface (MPI) Implementations

There are many MPI version 1 implementations, plus varying levels of completeness amongst a number of implementations of MPI version 2. There are many proprietary and commercial implementations. All the main HPC providers (IBM, HP, Sun, Intel, SGI, to name but a few) have their own implementation, as well as those offered by commercial software providers (MPI/Pro, WMPI-II, Scali). However, there are some robust publicly available implementations; the most prevalent are MPICH and LAM/MPI (although the latter is currently being replaced by the more recent OpenMPI). There are two distinct versions of MPICH: MPICH-1.2 and MPICH2. The latter is now the actively developed project, while the former remains maintained due to its large installed base.

MPICH Version-1.X

MPICH is an open-source implementation of MPI [182]. Currently the MPI 1.2 standard is supported, while some sections of MPI-2 are also implemented. The overall design is built around the concept of an abstract device, whereby in order to port MPICH to new communications hardware all that needs to be done is to implement a new abstract device interface (ADI). For example, there are implementations for the MPICH ADI for the following devices:

- **ch_p4** This device allows communication across networks using sockets and the TCP/IP suite of protocols. A typical cluster with a commodity Ethernet network will utilise this device for the underlying communications layer.
- **shmem** allows for processes to communicate using physical shared memory, thus largely removing the overhead of communication. The *shmem* device is used in MPI processes running on the same SMP system, so processes can communicate

simply by performing a memory copy without having to incur the latencies of the protocol stack associated with the *ch_p4* device.

- **Globus** is a communications device for a grid environment. It's difference to the *ch_p4* device is its integration with the ubiquitous Globus toolkit [183, 49]. There are also optimisations that allow for a significant performance increase with the use of collective calls by making use of topology information [52].
- **SCI** & **Myrinet** devices allow for the exploitation of these low-latency and high-bandwidth communication interconnects. These interconnects can then be used to enable the construction of a high performance and cost effective compute cluster.

Exploring the use of MPICH [184] for the communication between processes executing on distributed nodes allows for the exploitation of an optimised and stable message passing library, and leveraging of useful MPI resources such as profiling tools and debuggers. Its use also use insulates the system from platform dependencies and will ease porting to other architectures and platforms in the future.
Unfortunately the current Grid enabled version of MPICH, MPICH-G2 [49], is based on a MPICH distribution (currently version 1.2.5) that has no support for multi-threaded applications. This makes optimising applications hard, as a DSM system thread requires a MPI communication channel, and so can only be used in MPI_THREAD_FUNNELLED mode. MPI implementations exist that provide thread safety, however they are not grid-enabled. Future MPI implementations can be expected to support both multi-threading and grid.

MPICH-2

MPICH2 is an implementation of the MPI-2 standard that evolved and extends the efforts of MPICH-1.2.x described above. Although currently there is no communication device implemented that supports multi-site grid applications, importantly for this thesis there is robust support provided for multi-threaded applications.
The main difference between MPICH2 and the previous MPICH-1.X, apart from supporting implementing the MPI-2 standard, is that the former separates the actual management of application processes (i.e. start-up) from the applications that use MPI. This allows for different job managers to be developed and used without affecting the MPI applications themselves. Different types of process manager exist depending upon the execution environment: *gforker* for starting process locally on a machine (for testing), *mpd* for non-dedicated clusters, and *spmd* for Microsoft Windows platforms. Also third party managers are easily supported, e.g. support is available [185] for managed clusters such as those managed by systems like PBS [186] and Torque[187].

LAM/MPI

LAM/MPI provides a complete implementation of the MPI 1.2 standard, and provides significant portions of the MPI-2 standard [114]. It is constructed on top of LAM

which is a programming environment and development system for a message passing multicomputer [188]. The use of clusters of SMPs is supported, with optimisations to take advantage of the hardware to perform the messaging between two processes located on the same node. Such optimisations can occur with little on no modifications to the user application code. LAM/MPI also includes support for grid environments with close integration with the Globus toolkit.
LAM/MPI provides the ability to checkpoint an application, therefore allowing for the restart of an application at a later stage if problems arise. Check-pointing support in LAM/MPI requires the use of an external third party support such as [189]. Like the MPICH implementation, LAM/MPI currently has no support for multi-threaded applications.

PAC-X

PAC-X is a MPI implementation that enables MPI in a metacomputing environment; applications can execute across a number of distributed sites without a change in source code [48]. To the developer the system appears to be a single resource. Within each of the sites different vendor MPI implementations may be employed to take advantage of increased performance offered by the local interconnect.
External communication between the resources can be achieved via any supported protocol, such as TCP/IP using the socket library, see Section 5.1.1. At each site in the meta-computer a proxy node is used to transport messages between sites (this node requires a routable IP address, while the rest may have non-routable private addresses). When a message from a process on one site needs to be sent to a process on another it will be directed to the local proxy, which will deliver the message to the proxy on the remote site, which will in turn deliver it to the correct destination node. This approach will introduce extra latency due to the proxy-ing approach.

FT-MPI

Fault Tolerant MPI (FT-MPI) is a relatively new MPI implementation. It is implemented with a view to solving one of the principle deficiencies of current MPI standards, namely lack of proper support for fault tolerance [36]. Current standards only allow for the graceful exit of MPI applications once a fault has occurred. To achieve fault-tolerance the implementers of this project had to deviate slightly from the MPI standards. The application developer must attach fault handlers at the initialisation stage. If an error occurs the fault handler is called, thus allowing files to be closed and other basic actions to occur. FT-MPI overcomes this by allowing the user application to periodically checkpoint its progress and also through the usage of naming and notification services. Once a communication has an error state it can recover by rebuilding it.

OpenMPI

OpenMPI is an amalgamation of the previous three implementations, aiming to combine beneficial technologies and resources from these projects [190]. Currently this implementation is not suitable for production environments. Due to the limited multi-threaded support in the implementation (it is still in development) it could not be considered for use in this thesis.

The Parallel Virtual Machine (PVM)

PVM is more than just a message passing implementation [191]. It is a complete system that combines multiple machines to form a virtual machine that is available to users to run parallel message passing applications on. Each machine can be different.
A PVM daemon (pvmd) will run on each node of the virtual machine. A user connects to a PVM console where the applications can be created and managed from. The PVM console in turn interacts with the *pvmd* daemons.
A small subset of the PVM API is given in Listing D.1. Many of its novel features also appear in later MPI versions. PVM has started to show its age, no support for user multi-threaded applications is available because the API doesn't support them. While still popular in academic institutions it is quickly being phased out by the ever increasingly ubiquitous nature of MPI.

```
pvm_mytid()      // Returns the task identifier of the process
pvm_gsize()      // Get the number of precesses in the group
pvm_parent()     // Obtain the identifier of the parent
pvm_spawn(..)    // Spawn a new PVM process
pvm_exit(id)     // kill a PVM process
pvm_send(..)     // send a message to another process
pvm_recv(..)     // receive a message from a remote process
pvm_barrier(..) // synchronise with other processes
```

Listing D.1: *Subset of the PVM API*

Monitoring Implementations

R-GMA is described in Section 5.2.1, and so will not be covered here.

MDS

The Monitoring and Discovery Service (MDS) component of the Globus Toolkit [108] provides a grid information service accessible through a single interface. It is highly scalable and can handle static and dynamic data. It was constructed in order to provide a standard mechanism where the status of resources and configuration information could

be easily accessed by a user. It also integrates the Globus Grid Security Infrastructure (GSI) allowing for restricting access to data [148].
MDS is built upon the hierarchical Lightweight Directory Access Protocol (LDAP). It consists of three components i) the *Grid Index Information Service* (GIIS) is a directory allowing for the aggregation of data sources lower down in the hierarchy, ii) *Grid Resource Information Service* (GRIS) acts as the content portal for the resource, and iii) *Information Providers* (IPs) provide the interface between the source of data and the GRIS. As depicted in Figure D.1 a GRIS will register with its associated GIIS. Caching of data is used extensively to minimise the communication overhead. IPs exist that can publish data according to the GLUE schema and also to a MDS core schema.
It has been shown that when MDS is integrated with a MPI implementation to derive a topology map, dramatic improvements are obtainable in the performance of a number of MPI collective operations through the usage of hierarchy awareness [51, 52].

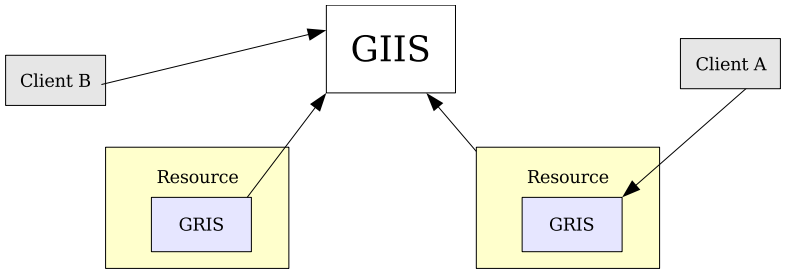


Figure D.1: *MDS Architecture*

Netlogger

Netlogger is a system that provides for the monitoring of the entire system at the host, networking, and application levels [192]. Its primary application domain is in the performance analysis of distributed applications. A special feature is the ability to correlate application performance with network information, e.g. the number of dropped packets, allowing for the identification of intermittent poor communication performance.
The primary concern is the requirement for the provision of global time so that all processes are synchronised. This requires that all nodes be synchronised using a network time protocol (NTP) server, or have access to a time source such as a GPS device. Netlogger output logging data conforms to the IETF Universal Logger Message (ULM) format. These logging messages can be directed to user/system files (using syslogd), to memory buffers, and across networks to remote netlogd servers. A netlogging daemon (*netlogd*) is located at a central host and is used to collect logging data from all application processes, Figure D.2 illustrates the potential downside of this approach as a bottleneck may arise at the *netlogd* daemon. The Netlogger API consists of only 6 functions:

```
NetLoggerOpen()      // Create a netlogger connection
NetLoggerClose()     // Close the netlogger connection
NetLoggerWrite()     // Send a netlogger message
NetLoggerGTWrite()   // Send a message, timestamped (gettimeofday)
NetLoggerFlush()     // Flush the buffer of any messages
NetLoggerSetLevel()  // Set the verbosity level of ULM messages
```

Listing D.2: *Netlogger API*

Netlogger provides a front-end tool for the viewing of logging data, however it is not suitable for monitoring many concurrent events (with a frequency < 20ms). Netlogger is not suitable for analysing MPI programs as the resolution of potential events is too low. In addition there is no support for an application to query the logging data, such as might be required when a topology graph is required. If the information requirement is not present then it is feasible to implement the monitoring system using this library. Future Netlogger releases aim to provide support for a *publish & subscribe* model, akin to that provided by R-GMA (described in Section 5.2.1).

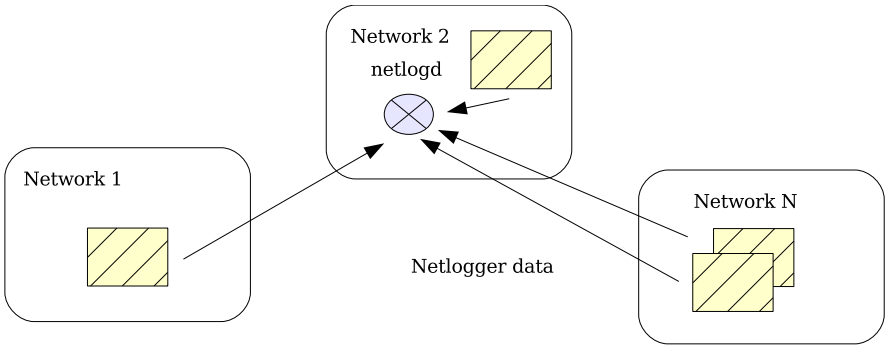


Figure D.2: *Netlogger Architecture*

OpenMP

The complete specifications for all OpenMP versions is available at [4].

OpenMP Compiler Directives

An OpenMP directive consists of compulsory terms and optional terms. No other comments may appear on the same line of the directive. First there is the declaration that identifies to the compiler that it is an OpenMP directive. In the C version of the standard this is **#pragma omp**. A valid OpenMP directive (or combination of) will follow next, and optional clauses to the directive can then be specified. In C, an OpenMP directive has the following form.

#pragma omp *name-of-directive(s) [clauses,..]*

Parallelisation

Parallelisation in OpenMP is achieved via the use of the **parallel** directive signals. This signals the start of a code segment that is capable of being executed concurrently by a number of threads. The implementation will fork the number of threads that is specified either by the specified API function (**omp_set_num_threads()** (page 232), an environment variable (**OMP_NUM_THREADS**), or by a clause (**NUM_THREADS**). For SMG, a thread count in excess of the number of processes that the SMG job was started with could result in the surplus threads being allotted in a round robin fashion per process. Current OpenMP implementations implement the *parallel* directive by encapsulating all code within its scope as a distinct function.

Work-sharing

The three directives below provide for sharing work among the team of threads that were created using the *parallel* directive (i.e. they must be enclosed in the parallel region). It is important to note that they must be encountered by all members of the thread team or none at all [193]. The first two work-sharing directives may be combined with the *parallel* directive, enabling a shorthand version in the following manner:

#pragma omp *parallel for.*

- **for** divides the iterations of a for loop among the the team of threads (akin to data decomposition described in Section 2.3). The method by which the number of iterations can be apportioned to each thread can be specified using one of the scheduling clauses listed on page 231.
- **sections** allocates different sections of work to the members of the thread team (i.e. functional decomposition described in Section 2.3)
- **single** allows for a region of parallel code to be serialised thus allowing only one thread to execute it. This allows for regions of code within the parallel region that may not be suitable for all threads to execute, i.e. not thread safe

The scheduling clause in the *for* directive can be accommodated using the SMG work division call described in Section 9.2.1. The latter two directives are easily provided for using simple flow-control statements.

Synchronisation

There are a number of directives that are available that enable different thread synchronisation operations for a team.

- **master** allows only the master thread to execute the code region. All other members of the thread team will skip it. It can be trivially implemented by SMG using a simple if-then-else structure to ensure that only one thread executes the code block.
- **critical** allows all threads to execute a given code region, but only one at a time. The functionality may be provided by guarding the region with an exclusive lock (see Section 2.4.1) that is accessible to all threads in the team.
- **barrier** ensures that all threads in the team become synchronised in the same manner described in Section 2.4.2.
- **atomic** allows the single statement following the directive to be executed in an atomic fashion, i.e. similar to the *critical* directive, but allowing implementation optimisations.
- **flush** identifies a synchronisation point in code that requires that a consistent view of memory, so this directive has a particular resonance with the memory consistency model provided by the SMG system.
- **ordered** requires that the code block is executed in the same order as it would execute in a sequential process.

Data Environment

The **threadprivate** directive is the only directive providing data scope control. It provides for the scoping of data during execution of the application, i.e. it allows for each thread to have a private copy of any global variables. This directive may be specified outside the range of a *parallel* directive.

OpenMP Data Scoping

In OpenMP all variables are shared by default. This may not be suitable in all cases, so data scoping directives are provided to change the scope of a variable if it should be different from the norm.

- **private** is similar to the *threadprivate* directive, allowing for each thread to have a private copy of a variable, but only if specified by the clause, i.e. it has a dynamic nature, but is non-persistent between scopes.
- **firstprivate** is a derivation of the *private* clause but specifies that each thread's variable is initialised from the original variable.
- **lastprivate** specifies that the last thread to leave the scoped region will copy their version of the variable to the global version.
- **shared** specifies that the variable is shared (only relevant if the default sharing policy is no sharing).

- **default** specifies that the variable will have the default sharing scope.
- **reduction** allows the (implicit) private copies of variables for all threads to be combined in a specified reduction operation e.g. +, -, *, /.
- **copyin** applies only to variables declared using the *threadprivate* directive. All threads have their private copy of the variable initialised from the original at the beginning of the scoped region.
- **copyprivate** allows for a private variable to be shared among other members of the team.

OpenMP Scheduling Clauses

- **STATIC** scheduling divides up the iterations among the threads in a chunk of a given size.
- **DYNAMIC** assigns each thread a chunk of work, the size of which is set dynamically. Once the thread is finished it may request another chunk. This allows for load-balancing of the work to be performed.
- **GUIDED** is similar to *DYNAMIC*, but allows for the chunk size to be reduced.
- **RUNTIME** allows for the scheduling to be delayed until run-time.

OpenMP Environment Variables

There are four OpenMP environment variables:

- **OMP_SCHEDULE** is used to declare the default run-time scheduling parameter. It specifies the type (a scheduling clause) and a parameter, e.g. for *static* scheduling a chunk size can be specified, such as *'static 10'*.
- **OMP_NUM_THREADS** declares the default number of threads to use during a parallel region. If unset, the default value is equal to the number of processes * 1. This can be overwritten throughout the execution of the application (it can be higher or lower).
- **OMP_DYNAMIC** specifies if dynamic adjustment in the number of threads is allowed, i.e. allows the previous variable to be overridden.
- **OMP_NESTED** specifies if nested parallelism is allowed (or provided for by the implementation).

OpenMP Library Functions

The APIs calls are mostly self explanatory. Most enable the dynamic setting of the behaviour of the OpenMP application like the clauses and environmental variables described above. For a complete definition see the OpenMP specification [128].

Thread data

```
void omp_set_num_threads(int num_threads)
int omp_get_num_threads(void)
int omp_get_max_threads(void)
int omp_get_thread_num(void)
int omp_get_num_procs(void)
```

Environment querying

```
int    omp_in_parallel(void)
void   omp_set_dynamic(int threads)
int    omp_get_dynamic(void)
void   omp_set_nested(int nested)
int    omp_get_nested(void)
```

Lock functions

```
void   omp_init_lock(omp_lock_t *lock)
void   omp_destroy_lock(omp_lock_t *lock)
void   omp_set_lock(omp_lock_t *lock)
void   omp_unset_lock(omp_lock_t *lock)
void   omp_test_lock(omp_lock_t *lock)
```

Timing routines

```
double  omp_get_wtime(void)
double  omp_get_wtick(void)
```

Linux information

SEGV Pagefault Structure: the structure presented to the signal handler on the generation of a SEGV access fault, and therefore passed to the SMG fault handler is:

```
siginfo_t {
  int      si_signo;  /* Signal number */
  int      si_errno;  /* An errno value */
  int      si_code;   /* Signal code */
  pid_t    si_pid;    /* Sending process ID */
  uid_t    si_uid;    /* Real user ID of sending proc. */
  int      si_status; /* Exit value or signal */
  clock_t  si_utime;  /* User time consumed */
  clock_t  si_stime;  /* System time consumed */
  sigval_t si_value;  /* Signal value */
  int      si_int;    /* POSIX.1b signal */
  void *   si_ptr;    /* POSIX.1b signal */
  void *   si_addr;   /* Mem location which caused fault */
  int      si_band;   /* Band event */
  int      si_fd;     /* File descriptor */
}
```

Listing D.3: *Structure Passed to Handler on SEGV*

APPENDIX E

SMG Reference Manual

The SMG DSM was developed to be a user-friendly system that would require minimum preparatory work in order to get the uninitiated developer creating shared memory application, and with little effort. This reference will give a quick introduction to building and configuring the SMG DSM.
This Appendix gives a brief introduction to developing with the SMG API, Appendix F (page 287) gives more comprehensive examples with code for locks, barriers, and hybridised SMG/MPI application. A detailed SMG API reference and related documentation is also included, see page 252.

Prerequisites

To build the SMG DSM library from source, there are a number of requirements for applications to be installed on the build machine:

autotools *autotools* package (which provides autoconf, automake) is required for the build process, although SMG can be (painstakingly) build by hand if the tool-chain is not available.

MPI SMG requires an underlying communication system, the default is to make use of *MPI*, with a suggested feature that it support multi-threaded calls in the *MPI_THREAD_MULTIPLE*.

pthreads The *pthreads* threading package is also frequently used in SMG , although many modern operating systems provide this package as standard, or is readily available.

R-GMA If the *R-GMA* information & monitoring system modules are required to be build for SMG , then it is a requirement that it is installed. The current implementation that comes as standard with the *gLite Version 3.0* middleware distribution is supported.

Building SMG

Building SMG is a relatively trivial matter when using the supplied configure scripts. Once the source code has been unpacked into a suitable location, then the `autogen.sh` script should be run. This script invokes the standard `autoconf` tools which generates a configure script `configure` which will be used to generate makefiles. This script has a number of options, some of the salient one are listed in table E.1.
Once the configure script has completed successfully, then SMG can be built by running the `make` command within the top-level source directory. Subsequently `make install` can be invoked, if required, to install the SMG libraries, header files, and helper scripts into the the default location, or as specified by the *–prefix* option to the configure script.

Option	Description
–prefix=<path>	Location where to install SMG
–comm-type=<type>	The default communications system to use (Default: MPI)
–with-mpi=<path>	This specifies the path to the MPI installation to use
–mpi-threaded=<yes/no>	Specifies if the MPI library supports multi-threaded calls
–with-cc=<mpi compiler>	The command name to invoke compiler (Default: *mpicc*)
–with-rgma=<path>	Path to the R-GMA library installation
–with-smginfo=<path>	Path to a prebuild SMG information system module
–with-smginfotype=<type>	The default information module type to use
–with-smgmon=<path>	Path to a prebuild SMG monitoring system module
–with-smgmontype=<type>	The default monitoring module type to use

Table E.1: *SMG DSM Core Engine Code*

Compiling a SMG application

A SMG application is compiled in the same manner as one would when compiling similar applications using the underlying system services, e.g. if MPI is to be used for communication, then the application should be compiled with the same compiler and link to the same libraries as required by the underlying MPI distribution. The only additional requirement is for the SMG libraries to be specified. Consider a simple 'helloworld' application below:

```
/* Include the SMG API definitions with no consistency support. */
#include "smg.h"

int main (int argc, char *argv[]){
  int error, flag = (NO_INFORMATION | NO_MONITORING);

  /* Initialise the SMG environment application without support for
     information and monitoring services. */
  error = SMG_init(&argc, &argv, flag, NO_CONSISTENCY);
  if(error != SMG_SUCCESS){
    printf("Cannot initialise SMG! exiting\n");
    exit(-1);
  }

  printf("Helloworld! I'm process %d of %d", SMG_rank, SMG_size);

  error = SMG_finalise();

  return error;
}
```

Listing E.1: *Hello World with SMG*

When the *configure* script is executed it creates a `smgcc` script that facilitates the development process with SMG by acting as a wrapper around the underlying compiler. When the developer makes use of this script, the paths of SMG header files and libraries are automatically included for the developer. However, the developer may also directly specify alternative modules, if others than the default are required, this allows different communication/information/monitoring modules to be easily specified at the linkage time of the application. The *helloworld.c* application above can be compiled in either manner below using the commands:

```
> smgcc -Wall -c -o helloworld.o helloworld.c
> smgcc -Wall -o helloworld helloworld.o
                                  **  or  **
> mpicc -Wall -I$SMG_HOME/include -c -o helloworld.o helloworld.c
> mpicc -o helloworld helloworld.o -L$SMG_HOME/lib -lsmg   \
            -L$SMG_HOME/modules/comm/lib -lcomm_mpi_t      \
            -L$SMG_HOME/modules/info/lib -linfoSMG_null    \
            -L$SMG_HOME/modules/mon/lib  -llogg_Basic      \
            -L$SMG_HOME/modules/util/lib -lsmg_util        \
            -lm -lpthread -ldl
```

Running SMG applications

The manner of executing a SMG application is dependent on the system services that the application was compiled with, so if MPI is used for the DSM communication, then the application should be executed in a similar manner that a regular MPI using the same MPI distribution would be.
The *helloworld* application created in the previous section, which was compiled to use MPI for DSM engine communication, can be executed using the relevant command, `mpiexec`, and MPI options such as the number of processes to start the application with `-np 4`, for MPI communication the command is demonstrated below in Figure E.1, together with the output from the execution the application.

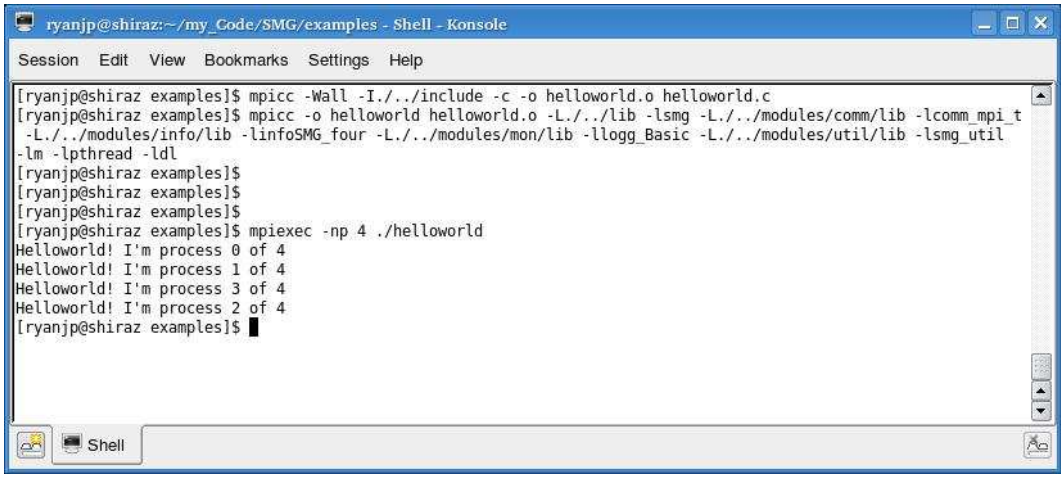


Figure E.1: *Execution of helloworld.c*

Command-line options

The amount of command line options is limited at present to just three, but is expected to expand in the future. The three options:

- `-i <identifier>` to specify the job identifier
- `-o <identifier>` the output directory for system files
- `-s` is the option to gather more detailed statistics than is available with the output of *SMG_print_state*

The identifier option, *-i* is obligatory when the application uses a R-GMA based information and/or monitoring system, when a SMG application is submitted as a grid job then middleware job identifier (given by the `$EDG_WL_JOBID` environment variable for *gLite* middleware) has been used successfully in this field, see page 87 for an example of this use.

E.1 Developing with SMG

The majority of SMG API definitions are located in the *smg.h* header file which is detailed below on page 243. Additional definitions which are required for the use of entry consistency (EC) are given in the *smg_ec.h* header file described on page 276. When developing SMG application it is sufficient to include the latter if EC is required. The API can be broadly classified under four sections System management, shared-memory management, synchronisation, and utility functions.

E.1.1 SMG API Quick-Reference

Listed below are the SMG API commands provided for quick-reference, more comprehensive API documentation is provided on page 243.

System Operations

```
int SMG_init(int *argc, char **argv[], int flags, int type_consist);
int SMG_finalise();
int SMG_get_rank(int *rank);
int SMG_get_size(int *size);
```

Synchronisation Operations

```
int SMG_lock_declare(int lock_id, int flags);
int SMG_read_lock_acquire(int lock_id);
int SMG_write_lock_acquire(int lock_id);
int SMG_lock_unlock(int lock_id);
int SMG_barrier_declare(int barrier_id, int type);
int SMG_barrier(int barrier_id, int flags);
```

Shared Memory Management

```
int SMG_shmem_malloc(int id, int size, void **pointer, int type,
                     int lock_bind_to);
int SMG_shmem_make(int id, int size, void **pointer, int type,
                   int lock_bind_to);
int SMG_shmem_free(int id);
int SMG_shmem_map(int id, void **pointer);
int SMG_memory_get_start(int identifier, void **start);
int SMG_reserve(void *pointer_from, void *pointer_to);
int SMG_local_2_globalptr(void *pointer, int *global);
int SMG_global_2_localptr(void **pointer, int *global);
```

Hybridisation Routines

```
int SMG_shmem_share(int object_id);
int SMG_shmem_noshare(int object_id);
int SMG_shmem_valid(int object_id, void *start, int size);
int SMG_shmem_invalid(int object_id, void *start, int size);
```

User Multi-threading

```
int SMG_thread_create(pthread_t *tid, pthread_attr_t *attr,
                      void *start_routine, void *arg);
int SMG_thread_join(int tid, void **exit_val);
int SMG_thread_count();
int SMG_thread_exit(int return code);
```

Utility Functions

```
int SMG_print_state();
int SMG_user_tag(char *user_tag, int lineno, char *filename);
int SMG_work_distribution(int TYPE, void *from, void *to, int how,
                          void *my_from, void *my_to, void *param);
int SMG_module_load(int MODULE_TYPE, char *location);
int SMG_internal_get(int key, void *value);
int SMG_internal_set(int key, int value);
int SMG_get_comm_handle(void *comm_t);
```

E.1.2 Developing with SMG - An Extended Example

This example is a simple demonstration of the multi-writer support provided in SMG . Figure E.2 below shows the output of the application when executed with a given number of processes. All processes initialise the SMG environment (Line 21) and the allocate a shared memory region (Line 28). Some of the applications modify the shared region with a value synonymous with their process rank. The barrier that the shared memory region is bound to is subsequently invoked which results with all modifications from all processes becoming visible at all other processes. All process terminate, but all must first call the finalise routine (Line 81) to clean up the SMG environment and the underlying communication system(s).

```
  /* Include the SMG API definitions with entry consistency
2  * (extensions).
   */
4 #include "smg_ec.h"

6 #define N  2000
  #define A_GRID_ID 3
8 #define GRID_BARRIER 5

10
  int main (int argc, char *argv[]){
12   int *a_grid;
     int i, flag, error;
14
     flag = (NO_INFORMATION | NO_MONITORING);
16
```

```
18  /* Initialise the SMG environment application without support for
     * information and monitoring services. Specify EC is required.
20   */
    error = SMG_init(&argc, &argv, flag, ENTRY_CONSISTENCY);
22

24  /* Allocate a shared memory region, size = N integers, of type EC
     * that employs the barrier with identifier =  GRID_BARRIER for
26   * consistency operations
     */
28  error = SMG_shmem_malloc(A_GRID_ID, (N * sizeof(int)),
                    (void**)&a_grid, (ENTRY | NAMED_BARRIER),
30                  GRID_BARRIER);

32  /* Invoke a barrier to sync all processes
     */
34  error = SMG_barrier(GRID_BARRIER+1,0);

36
    /* All processes, except rank 0, perform non-conflicting writes, with
38   * value = (100 + process_rank), to the shared memory region A.
     */
40  if(SMG_proc_rank == 0){
      printf("(%2d) Other processes write to the memory region \
42                    (Local cache:%p)\n", SMG_proc_rank, a_grid);
    }else{
44    for(i=0; i < 2; i++){
        a_grid[(SMG_proc_rank * 3) + i] = 100 + SMG_proc_rank;
46    }

48    /* The process that modify the shared region print out a portion
       * of their version of it.
50     */
      printf("(%2d) Write 'A' ", SMG_proc_rank);
52    for(i=0; i < 17; i++){
        printf("%3d ",a_grid[i]);
54    }
      printf("\n");
56  }

58
    /* Invoke the barrier which causes coherence events to be generated
60   * for any modified shared memory regions
     */
62  error = SMG_barrier(GRID_BARRIER,0);

64  if(SMG_proc_rank == 0){
      printf("\nBarrier !\n\n");
66  }

68
    /* All processes again print out the same portion of the shared
70   * region
```

```
     */
72   printf("(%2d) Read 'A'  ", SMG_proc_rank);
     for(i=0; i < 17; i++){
74       printf("%3d ", a_grid[i]);
     }
76   printf("\n");

78
     /* Clean-up the SMG environment and exit.
80    */
     error = SMG_finalise();
82
     return error;
84 }
```

Listing E.2: *A more advanced SMG application*

This application can be compiled using the same procedure as outlined on page 237. When executed with 8 processes the output that is produced is shown below in Figure E.2. All processes except process with rank = 0, modify the shared memory region with a specified value. Once all processes have reached the barrier, which the shared memory region was bound to, then a portion of the shared region is printed by all processes, all printing the same values.

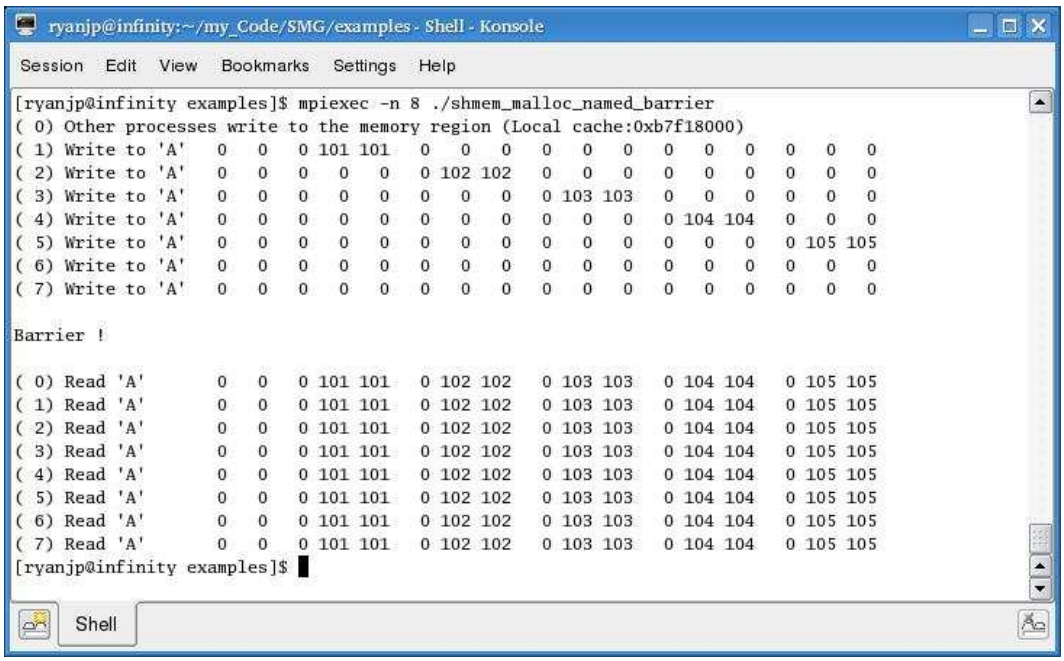
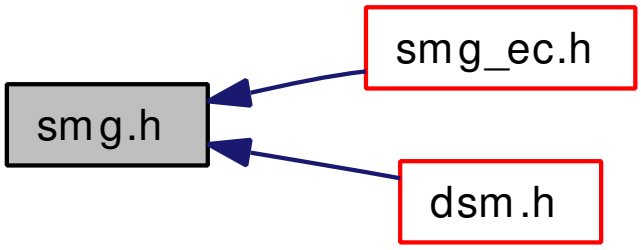


Figure E.2: *Multiple-Writer support in SMG*



E.2 smg.h File Reference

This file contains the main function prototypes for SMG DSM System.

```
#include <pthread.h>
#include <string.h>
#include <stdio.h>
```

Defines

- #define **SMG_FAILURE** 0

  Error status code signifying failure of a SMG function.

- #define **SMG_SUCCESS** 1

  Error status code returned by most of the SMG API calls signalling successful completion of the routine.

- #define **INFORMATION_FLAG** 0x00000001

  The definition of the flag that sets the system to use the information system.

- #define **MONITORING_FLAG** 0x00000002

  A definition of the flag that sets the system to use the monitoring system.

- #define **ABORT_FLAG** 0x00000004

  If requested system services such as information and/or monitoring functionality cannot be initialised, or not available, then the SMG application will abort.

- #define **NO_CONSISTENCY** 0x00000000

  No consistency specified.

- #define **SEQ_CONSISTENCY** 0x00000001

  Sequential consistency requested.

- #define **LAZY_CONSISTENCY** 0x00000002

  Lazy-release consistency.

- #define **ENTRY_CONSISTENCY** 0x00000004

  Entry consistency (EC) support required.

- #define **COHERENCE_NONE** 0x00000000

  No coherence option that can be specified when allocating shared memory, do the default coherence protocol will be employed.

- #define **COHERENCE_UPDATE** 0x00000100

  Update coherence option that can be specified when allocating shared memory.

- #define **COHERENCE_INVAL** 0x00000200

  Invalidation coherence option that can be specified when allocating shared memory.

- #define **COHERENCE_SUBSCRIBE** 0x00000800

  Subscription coherence option that can be specified when allocating shared memory.

- #define **LOCK_UNLOCKED** 0x0

  The unlocked status mode for the SMG lock synchronisation primitive.

- #define **LOCK_READ** LOCK_RO

  The read, or non-exclusive, status mode for the SMG lock synchronisation primitive.

- #define **LOCK_WRITE** LOCK_RW

  The write, or exclusive, status mode for the SMG lock synchronisation primitive.

- #define **LOCK_UNKNOWN** 0x4

  The status mode for the SMG lock synchronisation primitive when the status is unknown, usually because lock is at a remote process.

- #define **SMG_NULL** -1

  The SMG definition for a null memory reference.

Typedefs

- typedef long long **SMG_ptr**

  A type definition for SMG DSM shared-memory pointer.

Functions

- int **SMG_init** (int *argc, char **argv[], int flags, int type_consist)
  Initialise SMG DSM System.

- int **SMG_finalise** ()
  Terminate the SMG DSM environment.

- int **SMG_get_rank** (int *rank)
  Get the rank of the process within the SMG job.

- int **SMG_get_size** (int *size)
  Get the total number of SMG processes in the application.

- int **SMG_user_tag** (char *user_tag, int lineno, char *filename)
  Allow the user to create a tag for a specified line of user code.

- int **SMG_barrier_declare** (int barrier_id, int type)
  Declare the use of a barrier for future use.

- int **SMG_barrier** (int barrier_id, int flags)
  Invocation of a barrier operation for a specified barrier.

- int **SMG_sub_barrier** (int barrier_id, int who, int flags)
  Synchronise a subset of the processes in the application.

- int **SMG_barrier_coordinator** (int barrier_id)
  Obtain the processor rank of the coordinator for the specified barrier.

- int **SMG_lock_declare** (int lock_id, int flags)
  Declare the future use of a lock object with identifier lock_id.

- int **SMG_read_lock_acquire** (int lock_id)
  Acquire SMG lock in read (non-exclusive) mode.

- int **SMG_write_lock_acquire** (int lock_id)
  Acquire the specified SMG lock primitive in write (exclusive) mode.

- int **SMG_lock_acquire** (int lock_id, int mode)
  Generic lock acquisition function.

- int **SMG_lock_unlock** (int lock_id)

  Release a lock variable that was previously acquired.

- int **SMG_lock_status** (int lock_id, int *status)

  Obtain the local status of a lock primitive.

- int **SMG_shmem_malloc** (int id, int size, void **pointer, int type, int sync_bind_to)

  Allocate a block of shared memory.

- int **SMG_shmem_make** (int id, int size, void **pointer, int type, int sync_bind_to)

  Make an existing memory region sharable.

- int **SMG_shmem_free** (int id)

  free (locally) all the resources of a shared memory with the global identifier id.

- int **SMG_shmem_noshare** (int object_id)

  Disable the active sharing of a shared memory region.

- int **SMG_shmem_share** (int object_id)

  Enable the active sharing of a shared memory region.

- int **SMG_shmem_valid** (int object_id, void *start, int size)

  Validate a given section of a shared memory area.

- int **SMG_shmem_invalid** (int object_id, void *start, int size)

  Invalidate a given section of a shared memory area.

- int **SMG_shmem_flush** (int object_id)

  Flush modification to a shared memory region.

- int **SMG_memory_get_identifier** (void **ptr, int *identifier)

  Find the shared object identifier of a given memory location.

- int **SMG_memory_get_consistency** (int identifier, int *type)

  Get the consistency type for a specific shared memory region.

- int **SMG_memory_get_size** (int identifier, int *size)

  Obtain the size (bytes) of a shared memory region.

- int **SMG_memory_get_start** (int identifier, void **start)

  Get the start of the local cached version of the specified shared memory region.

- int **SMG_memory_get_owner** (int identifier, int *owner)

  Get the owner of a shared memory region.

- int **SMG_memory_set_granularity** (int identifier, int granularity)

  Set the granularity (in bytes) for the sharing of a shared memory region.

- int **SMG_memory_get_granularity** (int identifier, int *granularity)

  Get the sharing granularity for a given shared memory region.

- int **SMG_local_2_globalptr** (void *pointer, int *global)

  Convert a local memory reference to a global reference for passing between processes.

- int **SMG_global_2_localptr** (void **pointer, int *global)

  Convert a global memory reference to a local reference for passing references between processes.

- int **SMG_thread_create** (pthread_t *tid, pthread_attr_t *attr, void *start_routine, void *arg)

  Wrapper call around underlying thread creation routine.

- int **SMG_thread_count** ()

  Return the number of user threads currently alive.

- int **SMG_thread_exit** (int code)

  User thread exit routine.

- int **SMG_thread_join** (int tid, void **exit_val)

  SMG wrapper to underlying join (pthread_join) call.

- void **SMG_print_state** ()

  Print the current internal status of DSM engine.

- int **SMG_module_load** (int MODULE_TYPE, char *location)

  Load a SMG module that provides extra DSM functionality.

- int **SMG_set_default_consistency** (int consistency)

  Set the default consistency to be used with any memory allocation.

- int **SMG_get_default_consistency** ()

  Get the default memory consistency.

- int **SMG_have_consistency** (int consistency)

  Query the system to see if the specified consistency code is supported.

- void **SMG_local_response_time** ()

  Simple test to return the time for a local DSM system call.

- void **SMG_remote_response_time** (int remote)

  Simple test to return the time for a remote DSM system call.

- int **SMG_consistency_supported** (int type, int *supported)

  Test to see if a given consistency level is supported.

- int **SMG_internal_get** (int key, int *value)

  Get the value of an internal DSM attribute.

- int **SMG_internal_set** (int key, int value)

  Set the value of an internal DSM attribute.

- int **SMG_get_comm_handle** (int *handle)

  Set a communication handle, handle*, to the underlying communication infrastructure.*

- int **SMG_work_distribution** (int TYPE, void *from, void *to, int how, void *my_from, void *my_to, void *param)

  Divide workload among processes according to given criteria.

Variables

- int **SMG_proc_size**

  This specifies the number of processes that in the application.

- int **SMG_proc_rank**

  The rank of the process in the pool of processes.

E.2.1 Detailed Description

This file contains the main function prototypes for SMG DSM System.
The SMG library is a DSM implementation that allows for the development of applications in a shared memory programming style. This file ***smg.h*** *(p. 243)* specifies the API used to develop applications with the SMG DSM.

E.2.2 Define Documentation

#define SMG_FAILURE 0

Error status code signifying failure of a SMG function.

See also:

SMG_SUCCESS (p. 249)

Most SMG functions return an error status code signifying the success or failure of a given SMG function invocation. This error status code as its name suggests indicates an unsuccessful invocation of a SMG function.

#define SMG_SUCCESS 1

Error status code returned by most of the SMG API calls signalling successful completion of the routine.

See also:

SMG_FAILURE (p. 249)

Most SMG functions return an error status code signifying the success or failure of a given SMG function invocation. This error status code as its name suggests indicates successful invocation of the SMG function.

#define INFORMATION_FLAG 0x00000001

The definition of the flag that sets the system to use the information system.

Warning:

If the underlying system cannot initialise correctly, then the application will not abort, unless requested to do so.

This flag is used in the bit-field of operations provided to the *flags* parameter of the *SMG_init* routine results in the use of information system by the DSM being enabled, otherwise it is not used to devise a topology tree for DSM engine and communication optimisation. The information system used by SMG will be whatever is compiled with the application. A corresponding *NO_INFORMATION* flag also exists for requesting that no information system be used.

#define MONITORING_FLAG 0x00000002

A definition of the flag that sets the system to use the monitoring system.

Warning:

If the underlying monitoring system cannot initialise correctly, then the application will not abort, unless requested to do so.

The monitoring system allows for the system logging of monitoring data. This flag is specified in the bit-field of operations provided to the *flags* parameter of the *SMG_init* routine results in the use of monitoring system by the DSM being enabled, otherwise it is not used. The monitoring system employed will be whatever is compiled with the application. A corresponding *NO_MONITORING* flag also exists for requesting no monitoring system use.

#define ABORT_FLAG 0x00000004

If requested system services such as information and/or monitoring functionality cannot be initialised, or not available, then the SMG application will abort.
The system will abort if information monitoring is requested, and not available, otherwise the system will continue using a topology-unaware approach if the complementary *NO_ABORT* flag is used.

#define NO_CONSISTENCY 0x00000000

No consistency specified.
As consistences are specified in a bit-field, this is defined mainly for testing of DSM.

#define SEQ_CONSISTENCY 0x00000001

Sequential consistency requested.

Warning:

Although not supported in SMG, it is defined for future use.

#define LAZY_CONSISTENCY 0x00000002

Lazy-release consistency.

Warning:

Although not supported in SMG, it is defined for future use.

#define ENTRY_CONSISTENCY 0x00000004

Entry consistency (EC) support required.
Entry Consistency (EC) is default consistency model supported in SMG.

#define COHERENCE_NONE 0x00000000

No coherence option that can be specified when allocating shared memory, do the default coherence protocol will be employed.
This flag is specified in the bit-field of options to the *flags* field of *SMG_init*.

#define COHERENCE_UPDATE 0x00000100

Update coherence option that can be specified when allocating shared memory.
This flag is specified in the bit-field of options to the *flags* field of *SMG_init*.

#define COHERENCE_INVAL 0x00000200

Invalidation coherence option that can be specified when allocating shared memory.

Warning:

This option is still in development, and may produce errors!

This flag is specified in the bit-field of options to the *flags* field of *SMG_init*.

#define COHERENCE_SUBSCRIBE 0x00000800

Subscription coherence option that can be specified when allocating shared memory.
This flag is specified in the bit-field of options to the *flags* field of *SMG_init*.

#define LOCK_UNLOCKED 0x0

The unlocked status mode for the SMG lock synchronisation primitive.

#define LOCK_READ LOCK_RO

The read, or non-exclusive, status mode for the SMG lock synchronisation primitive.

#define LOCK_WRITE LOCK_RW

The write, or exclusive, status mode for the SMG lock synchronisation primitive.

#define LOCK_UNKNOWN 0x4

The status mode for the SMG lock synchronisation primitive when the status is unknown, usually because lock is at a remote process.

#define SMG_NULL -1

The SMG definition for a null memory reference.

E.2.3 Typedef Documentation

typedef long long SMG_ptr

A type definition for SMG DSM shared-memory pointer.
A DSM pointer has two components, an object id, & offset, that are required to be passed between two processes to correctly resolve potential conflicts that may arise with differences in location of shared memory regions at different processes. This primitive is used to encode both into a single primitive.

Warning:

Problems will arise with the use of the typedef in 64-bit environments

E.2.4 Function Documentation

int SMG_init (int * *argc*, char ** *argv*[ ], int *flags*, int *type_consist*)

Initialise SMG DSM System.

Parameters:

argc pointer to the argument count.

argv pointer to the list of arguments that were passed to main.

flags flags the set the default behaviour

type_consist the memory consistences required by the developer.

See also:

SMG_system_finalise()

Returns:

The SMG error status code

Warning:

This call should be called only once, repeated calls will produce undefined results.

Initialise the DSM system infrastructure, command line parameters are passed to this function. This function must be called before any other SMG call is invoked. The underlying communication infrastructure will be initialised through this function. The call will fail if the requested consistency type(s), or system services, are not available. The *argv* and *argc* are the arguments, and number of, that were passed into the *main* function of the application. The *flags* field allows the developer to specify a selection of default characteristics e.g. default coherence mechanism – *COHERENCE_UPDATE*. If a given flag can be passed in the *flags* field of this call, then this will expressly specified in that flag's documentation in this manual.

SMG_finalise ()

Terminate the SMG DSM environment.

Returns:

The SMG error status code

See also:

SMG_system_init()

Cleanly terminate the DSM environment in preparation for exiting of the process, this includes de-allocation of internal data structures and closing/removing internal files used by the DSM engine are closed. All processes must call this routine before exiting. The underlying communication mechanism(s) are also closed, so subsequently no further calls to communication handles may be performed that were obtained using *SMG_get_comm_handle*. This functions acts as an implicit barrier that ensures all processes have called this function before returning.

SMG_get_rank (int * *rank*)

Get the rank of the process within the SMG job.

Parameters:

rank pointer to location to store the process' rank

Returns:

The SMG error status code

See also:

SMG_get_size(int *size) (p. 253);

Warning:

No further *SMG* routines may be invoked subsequently to this routine.

This function will return the unique process identifier, or rank, of the calling process which will be in the range of 0..(N-1), where N is the total number of processes as returned by the call ***SMG_get_size()*** *(p. 253)*.

SMG_get_size (int * *size*)

Get the total number of SMG processes in the application.

Parameters:

size pointer to location to store the number of SMG processes.in the application.

Returns:

The SMG error status code

See also:

SMG_get_rank(int *rank) (p. 253)

Return the number of SMG processes participating in the application. This value is determined at run-time and should equal the number of processes the application is started with.

```
// This code snippet demonstrates the use of four basic SMG routines.

// Include the header file with SMG API definitions.
#include "smg.h"

int main (int argc, char *argv[]){
    int flag, error, i, j;

    // The request for information and monitoring systems are specified as
    // a bit-field in the flags field of the initialisation routine.
    flag = (INFORMATION_FLAG | MONITORING_FLAG);

    // Initialise the DSM, with no consistency model implementation.
    error = SMG_init(&argc, &argv, flag, NO_CONSISTENCY);
    if (error != SMG_SUCCESS){
        printf("SMG_init Unsuccessful!\n");
        exit(-1);
    }

    SMG_get_rank(&i);
    SMG_get_size(&j);
    printf("I am process %d of %d\n", i, j);

    error = SMG_finalise();
    if (error != SMG_SUCCESS){
        printf("Error while calling SMG_finalise!\n");
    }

    return error;
}
```

SMG_user_tag (char * *user_tag*, int *lineno*, char * *filename*)

Allow the user to create a tag for a specified line of user code.

Parameters:

user_tag String containing the user tag

lineno line number within the code file

filename Name for the file

Returns:

The SMG error status code

This function allows the user to specify a descriptive tag for a given line of code, then this tag can be referenced for instrumentation of code and by other components of the DSM such as the monitoring system. A C macro exists that wraps this call that uses the standard __LINE__ and __FILE__ macros.

SMG_barrier_declare (int *barrier_id*, int *type*)

Declare the use of a barrier for future use.

Parameters:

barrier_id the numerical identifier of the barrier

type the default characteristics of the barrier

Returns:

The SMG error status code

This function to initialise a barrier primitive is primarily used to change the default behaviour of a barrier primitive where the default is not appropriate, where this call is not performed then default behaviour will result. An important characteristic is the algorithm used to implement the barrier, possible choices include the *BINARY_TOPAWARE_BARRIER* or simpler *CENTRAL_SERVER_BARRIER* algorithms one of which can be passed in the *type* field.

SMG_barrier (int *barrier_id*, int *flags*)

Invocation of a barrier operation for a specified barrier.

Parameters:

barrier_id the numerical identifier of the barrier

flags Field specifies the local operation of the barrier within the local process.

Returns:

The SMG error status code

This function invokes a barrier synchronisation operation on the primitive specified by the *barrier_id* identifier. By default, a process in the SMG application will wait at this location until all process have reached the barrier. The default behaviour may be altered by specifying appropriate flags such as *FIRST_T_FIRE*, or ALL_T_FIRE that specify when the local barrier event may fire. For shared memory regions that employ a relaxed consistency model their related memory coherence events may occur in conjunction with this synchronisation call.

int SMG_sub_barrier (int *barrier_id*, int *who*, int *flags*)

Synchronise a subset of the processes in the application.

Parameters:

barrier_id the numerical identifier of the barrier

flags Specifies the local operation of the barrier within the

who Specifies what processes to take part in the operation local process.

Todo

Further development work to optimise this routine needs to be performed.

Warning:

No coherence operation occur with this variant of the barrier sync. primitive

Returns:

The SMG error status code

This function synchronises a subset of the processes in the SMG application, the processes take part in the 'waiting' section of operation are governed by the *who* parameter. Like the *SMG_barrier* call, the default behaviour may be altered by specifying appropriate flags such as *FIRST_T_FIRE*, or ALL_T_FIRE that specify when the local barrier event may fire. Memory coherence operations do not occur on a 'sub' barrier due to the potential for inconsistencies with shared memory region(s) to unintentionally arise.

SMG_barrier_coordinator (int *barrier_id*)

Obtain the processor rank of the coordinator for the specified barrier.

Parameters:

barrier_id the numerical identifier of the barrier

Returns:

The process rank of the barrier coordinator.

This function returns the rank of the process that coordinates the operation of barrier primitive, specified by the identifier *barrier_id*. This function can ultimately prove useful for barrier operations that will involve memory coherence operations allowing developers to more efficiently employ such mechanisms.

SMG_lock_declare (int *lock_id*, int *flags*)

Declare the future use of a lock object with identifier *lock_id*.

Parameters:

lock_id The numerical identifier of the lock

flags specifying the default behaviour of the lock

Returns:

The SMG error status code

Warning:

Repeated calls to this primitive after the lock has been utilised with the other lock routines may result in undefined behaviour.

SMG_lock_declare allows the user to predeclare the use of a lock and modify its default behaviour by specifying the *flags* field. If this function is to be used, then all process should invoke the call for a given lock before use of the other lock routines on the primitive.

SMG_read_lock_acquire (int *lock_id*)

Acquire SMG lock in read (non-exclusive) mode.

Parameters:

lock_id Identifier of the lock to acquire

See also:

SMG_write_lock_acquire(int lock_id) (p. 258)

Returns:

The SMG error status code

This function will acquire a lock primitive in read (non-exclusive) mode, this is a synchronous call and will return with status *SMG_SUCCESS* if the operation has succeeded, while any number of processes may concurrently hold a lock primitive in non-exclusive mode. A function will succeed if the primitive is not already in exclusive mode. Repeated invocations of the same primitive i.e. using the same *lock_id* will result in undefined behaviour. Consistency related operations for shared memory regions allocated with *SMG_shmem_malloc* may be performed in conjunction with this call.

SMG_write_lock_acquire (int *lock_id*)

Acquire the specified SMG lock primitive in write (exclusive) mode.

Parameters:

lock_id Identifier of the lock to acquire

See also:

SMG_read_lock_acquire(int lock_id) (p. 257)

Returns:

The SMG error status code

This function is similar in operation to SMG_write_lock_acquire, and is used to acquire a SMG lock in exclusive mode. This is a synchronous call and will return with status *SMG_SUCCESS* if the operation has succeeded. Only one process may hold a lock primitive in exclusive mode, the function will succeed if the primitive is not already acquired in either exclusive or non-exclusive mode. Repeated invocations of the same primitive i.e. using the same *lock_id* will result in undefined behaviour. Consistency related operations for shared memory regions allocated with *SMG_shmem_malloc* may be performed in conjunction with this call. If the lock is held in write mode it enjoys all the privileges accorded to read mode.

int SMG_lock_acquire (int *lock_id*, int *mode*)

Generic lock acquisition function.

Parameters:

lock_id Identifier of the lock to acquire in the specified mode.

mode the mode to acquire the lock in: read or write.

See also:

SMG_read_lock_acquire(int lock_id) (p. 257)
SMG_write_lock_acquire(int lock_id) (p. 258)

This function is provided as a utility function and is essentially serves as a wrapper around the primary lock acquisition functions: *SMG_write_lock_acquire* and SMG_read_lock_acquire. The mode in which the lock is acquired is specified via the *mode* field and is either *LOCK_WRITE* or *LOCK_READ*.

SMG_lock_unlock (int *lock_id*)

Release a lock variable that was previously acquired.

Parameters:

lock_id Identifier of the lock to release.

Returns:

The SMG error status code

This function will perform a release of a lock synchronisation primitive so that the status of the lock will be either: *LOCK_READ*, *LOCK_UNLOCKED* or *LOCK_UNKNOWN*. The lock will remain in *LOCK_READ* mode if the local process had delegated non-exclusive mode access to another process in the previous interval, while *LOCK_UNLOCKED* signifies that the lock is owned locally but is not held acquired in either exclusive or non-exclusive mode.

```
// This code snippet demonstrates the use of SMG synchronisation routines.
#include "smg_ec.h"

// Define the identifiers for the lock & barrier
#define FOO_LOCK    1
#define BAR_BARRIER 2
#define SHMEM_ID    1

int main (int argc, char *argv[]){
    int flag, error, my_rank;
    int *temp_int;

    // Request for information and monitoring systems
    flag = (INFORMATION_FLAG | MONITORING_FLAG);

    // Initialise the DSM, with no consistency model implementation.
    error = SMG_init(&argc, &argv, flag, NO_CONSISTENCY);
    if (error != SMG_SUCCESS){
        printf("SMG_init Unsuccessful!\n");
        exit(-1);
    }
    SMG_get_rank(&my_rank);

    // Allocate a small shared memory region, bind it to use with FOO_LOCK
    error = SMG_shmem_malloc(SHMEM_ID, (2 * sizeof(int)),
                        (void**)&temp_int, (ENTRY | BIND_LOCK), FOO_LOCK);

    // Acquire the lock, print out a message, and release
    SMG_write_lock_acquire(FOO_LOCK);
    *temp_int = my_rank;
    SMG_lock_unlock(FOO_LOCK);

    // All process then wait for all others to print out the message
    SMG_barrier(BAR_BARRIER, 0);

    // Processes find out which of them was last to acquire the lock!
    SMG_read_lock_acquire(FOO_LOCK);
    printf("Last process to write to shared variable: %d\n", *temp_int);
```

```
    SMG_lock_unlock(FOO_LOCK);

    error = SMG_finalise();
    if (error != SMG_SUCCESS){
        printf("Error while calling SMG_finalise!\n");
    }

    return error;
}
```

SMG_lock_status (int *lock_id*, int * *status*)

Obtain the local status of a lock primitive.

Parameters:

lock_id Identifier of the lock to obtain the status of.

status Pointer to location where to return status of a lock

Returns:

The SMG error status code

This utility function will return the local status of the specified lock primitive in the location pointed to by the status field. The value will either be *LOCK_WRITE*, *LOCK_READ*, *LOCK_UNLOCKED* or *LOCK_UNKNOWN*. The *LOCK_UNKNOWN* signifies that the local process is unaware of the status of a primitive as the owner is a remote process.

int SMG_shmem_malloc (int *id*, int *size*, void ** *pointer*, int *type*, int *sync_bind_to*)

Allocate a block of shared memory.

Parameters:

id the numerical reference for the block of shared memory to allocate

size the size of the shared memory area (bytes).

pointer location to store the reference of the shared region

type the characteristics of the shared memory region

sync_bind_to Is a parameter ultimately passed to the memory sharing mechanism

Warning:

This function should not be called repeatedly for the same shared memory identifier within the same synchronisation interval without the corresponding use of the free function ***SMG_shmem_free()*** *(*p. *261)* to release the resources

Returns:

The SMG error status code

This function allocates a block of shared memory of *size* bytes with a globally unique identifier *id*, and maps it into the local process address space at the location referenced by the *pointer* field. The *type* field is a bit-field that allows the developer to specify the characteristics of the shared memory region is to created with, these include consistency, and the write-collection technique that should be employed. Not all processes participating in the application are required to call the function, but all processes that call the routine with the same shared memory identifier should specify the same arguments. However, in the case there a process does not call the routine, the shared memory region will not be mapped into the local address space i.e. no local 'cache' exists.
The *sync_bind_to* parameter is type-specific and is ultimately passed to the collection of sharing mechanisms specified by the *type* field. For example, with entry consistency (EC), this specifies the identifier of the synchronisation primitive to bind to the use of the newly shared memory region.

SMG_shmem_make (int *id*, int *size*, void ** *pointer*, int *type*, int *sync_bind_to*)

Make an existing memory region sharable.

Parameters:

id the identifier of the shared region

size the size of the shared location

pointer reference to a pointer to the memory region to share

type the characteristics of the shared memory region

sync_bind_to Is a parameter ultimately passed to the memory sharing mechanism

Returns:

The SMG error status code

Warning:

the existing location must of been allocated using mmap() routine

This function is used to make a preallocated local memory region sharable. The memory region must be allocated on a page boundary, so it is highly advisable that the region was allocated using the *mmap* system call to ensure no further memory allocations use any part of its constituent virtual memory page(s).

SMG_shmem_free (int *id*)

free (locally) all the resources of a shared memory with the global identifier *id*.

Parameters:

id Identifier of the shared memory area to free.

Returns:

The SMG error status code

This function will free all resources consumed (twin area, diffs, internal handles) by the local process in the local cache of the shared memory area globally identified by *id*. The data may not be removed under certain circumstances such as when the local process is the memory region's owner and remote processes may require the use of the region in the future.

SMG_shmem_noshare (int *object_id*)

Disable the active sharing of a shared memory region.

Parameters:

object_id Identifier of region to share

Returns:

The SMG error status code

See also:

SMG_shmem_share(int object_id) (p. 262)

At certain stages in the execution of an application it may be desirable to turn off the global sharing (DSM management) for a given shared memory region. This functionality can be achieved using the *SMG_shmem_noshare* routine. The shared memory region to disable sharing on is specified in the *object_id* field. Once this call has successfully returned no further DSM consistency/coherence actions will result until active sharing of the region is re-enabled using the complementary function *SMG_shmem_share*. Once this function returns, the local cache can be modified as the developer sees fit.
This routine instructs the DSM engine to make a copy of the shared area for future comparison to be performed when DSM control of the area is re-enabled.

SMG_shmem_share (int *object_id*)

Enable the active sharing of a shared memory region.

Parameters:

object_id Identifier of region to share

Returns:

The SMG error status code

See also:

SMG_shmem_noshare(int object_id) (p. 262)

This routine is used to re-enable the active sharing of a given shared memory region. Any consistency related operations that need to take place will take place at the next appropriate point e.g. in the case of an entry consistent shared region this will be the next relevant synchronisation operation for that region. Before the routine is called the shared memory region needs to be properly validated using the provided routines *SMG_shmem_valid* or *SMG_shmem_invalid* to ensure that no conflicts arise with the cached copies of the shared memory region across all processes.
The current status of a shared area is compared with the copy that was created with the invocation of the *SMG_shmem_noshare* routine. Any differences that arise will be signalled to the DSM engine in order to take consistency related actions at the next appropriate point.

SMG_shmem_valid (int *object_id*, void * *start*, int *size*)

Validate a given section of a shared memory area.

Parameters:

object_id Identifier of region to share

start pointer to start of section to validate

size the size (bytes) to validate

Returns:

The SMG error status code

See also:

SMG_shmem_invalid(int object_id, void *start, int size) (p. 263)

The *SMG_shmem_valid* routine is used to ensure that no conflicts arise with the local cache of a shared memory region that may have arose intentionally/unintentionally when DSM sharing of the area was disabled. With this function these conflicts are removed by copying the specified section from the copy made of the original that was made when *SMG_shmem_noshare* was invoked. All other sections of the shared memory will be refreshed from the original copy made at the call to *SMG_shmem_noshare*.

SMG_shmem_invalid (int *object_id*, void * *start*, int *size*)

Invalidate a given section of a shared memory area.

Parameters:

object_id Identifier of region to share.

start Pointer to the beginning of the region to invalidate

size Number of bytes to invalidate

Returns:

The SMG error status code

See also:

SMG_shmem_valid(int object_id, void *start, int size) (p. 263)

This function is a complementary function to *SMG_shmem_valid*,

SMG_shmem_flush (int *object_id*)

Flush modification to a shared memory region.

Parameters:

object_id Identifier of region to share

Returns:

The SMG error status code

Warning:

Results of this function are non-deterministic where multiple processes call this function concurrently.

This function results in the local modifications to a specified shared memory region been flushed to all other processes without the use of error checking or event-ordering that would be available through normal use with relaxed-consistency mechanisms. The shared memory region may be left in an inconsistent state across all local caches if more than one process executes this routine concurrently.

int SMG_memory_get_identifier (void ** *ptr*, int * *identifier*)

Find the shared object identifier of a given memory location.

Parameters:

ptr memory location to query

identifier pointer where to store id, if it exists, -1 if no object maps to the specified location

Returns:

SMG error code

This utility function is provided to ascertain if a given memory location is within a SMG shared memory region. If the memory location specified by *ptr*, is within the shared memory region that was allocated using a routine like *SMG_shmem_malloc*, then that region's identifier will be stored to the location pointed to by *identifier*.

SMG_memory_get_consistency (int *identifier*, int * *type*)

Get the consistency type for a specific shared memory region.

Parameters:

identifier Numerical identifier of region to get consistency of.

type The memory region's consistency

Returns:

The SMG error status code

This routine will return the consistency type for a shared memory region specified by *identifier*. If the identifier maps to a valid shared memory region then the *type* field will correctly identify the consistency. If the shared memory region does not exist, or no local cache of the object is present then consistency type *NO_CONSISTENCY* will be set.

SMG_memory_get_size (int *identifier*, int * *size*)

Obtain the size (bytes) of a shared memory region.

Parameters:

identifier Identifier of shared memory region to obtain the size of.

size Location to store the size of the shared memory region.

Returns:

The SMG error status code

This utility function is provided to obtain the size of the shared memory region, if the region doesn't exist then the *size* will be set to 0.

SMG_memory_get_start (int *identifier*, void ** *start*)

Get the start of the local cached version of the specified shared memory region.

Parameters:

identifier Identifier of region to share

start A pointer to a reference giving the start of the shared memory area.

Returns:

The SMG error status code

This routine is provided to enable user code to obtain the start in the local address space for a shared memory region. Different processes within the same SMG application may return different values as a given shared memory region may be mapped at different locations in the virtual address spaces for different processes.

SMG_memory_get_owner (int *identifier*, int * *owner*)

Get the owner of a shared memory region.

Parameters:

identifier Numerical identifier of region to get the owner of

owner Location which to store the process rank of the region's owner

Returns:

The SMG error status code

All shared memory regions have an owner, which is responsible for the region's global management, this routine is provided to enable a user application to obtain this process identifier.

SMG_memory_set_granularity (int *identifier*, int *granularity*)

Set the granularity (in bytes) for the sharing of a shared memory region.

Parameters:

identifier Identifier of the shared region

granularity Granularity to set shared region to (in bytes)

Returns:

The SMG error status code

Warning:

Experimental code!

This routine enables the setting of the default granularity for the sharing of an object i.e. the smallest region at which write conflicts may be detected, the default is 4 bytes. It may prove highly beneficial in terms of write-collection performance, if the sharing granularity can be set to a value that better suits the application, as the default value may prove too small especially in circumstances where the default primitive used in the application is larger.

int SMG_memory_get_granularity (int *identifier*, int * *granularity*)

Get the sharing granularity for a given shared memory region.

Parameters:

identifier Identifier of region to get granularity of

granularity location where to store the requested item's granularity.

Returns:

The SMG error status code

See also:

SMG_memory_set_granularity (p. 266)

Complementary function to *SMG_memory_set_granularity* to get the granularity of a shared memory region specified by *identifier*.

SMG_local_2_globalptr (void * *pointer*, int * *global*)

Convert a local memory reference to a global reference for passing between processes.

Parameters:

pointer local reference to transform into a global reference

global the resulting global reference to a shared memory location

Returns:

The SMG error status code

See also:

SMG_global_2_localptr (p. 267)

It may be desirable to pass references to shared memory regions between processes of an application. Two main barriers to passing references directly are (i) shared memory regions may be mapped at different locations in the virtual address space of the two different processes, and (ii) memory references may be of incompatible sizes at different processes i.e. one process is 32-bit while the other has 64-bit addressing. This function solves this problem solves this by allowing the user to convert a local memory reference to a global shared memory reference.

SMG_global_2_localptr (void ** *pointer*, int * *global*)

Convert a global memory reference to a local reference for passing references between processes.

Parameters:

pointer pointer to local reference that the global reference is transformed to.

global the global reference to obtain the local reference of.

Returns:

The SMG error status code

See also:

SMG_local_2_globalptr (p. 267)

This is the complementary function that transforms a global shared memory reference created using *SMG_local_2_globalptr*, possibly at a remote process, to a pointer that is a valid local reference to a shared memory region.

SMG_thread_create (pthread_t * *tid*, pthread_attr_t * *attr*, void * *start_routine*, void * *arg*)

Wrapper call around underlying thread creation routine.

Parameters:

tid the thread identifier returned by the underlying pthread creation routine.

attr attributes passed to the thread creation

start_routine Function pointer to entry function for the new thread

arg Argument to the function, if required, to entry function of new thread.

Returns:

The error code code returned by the underlying thread package

This routine is a wrapper around the thread creation routine of the underlying threads packages (only pthreads currently supported). It is important for the DSM engine to be aware of thread creation/finalisation as certain functionality, namely SMG barriers, requires the DSM to know the count of user threads in the application. By wrapping the underlying threading package in this way this count can be achieved. The prototype of this function mimics that of the pthread thread creation call. If the user thread is not created using this call then it is invisible to the DSM engine.

SMG_thread_count ()

Return the number of user threads currently alive.

Returns:

The number of user threads alive in the local process.

This routine returns the number of threads created using *SMG_thread_create*, and that have not yet exited.

SMG_thread_exit (int *code*)

User thread exit routine.

Parameters:

code Exit code of the user thread

Returns:

The SMG error status code

This routine is user to return the number of user threads, created using *SMG_thread_create*, and that are running in the local process. If the user thread was not created using this call, then it is not visible and is not in the count returned.

SMG_thread_join (int *tid*, void ** *exit_val*)

SMG wrapper to underlying join (pthread_join) call.

Parameters:

tid Thread identifier

exit_val pointer to a reference where the exit code of the thread is set

Returns:

The SMG error status code

This routine is used to join a thread created using the *SMG_thread_create* routine. The exit value of the thread, with identifier *tid*, will be stored at the location pointed to by the reference *exit_val*. It forms a wrapper around the underlying thread package's join call.

```
// This code snippet demonstrates the use of SMG threading routines.

// Include the header file with SMG API definitions.
#include "smg.h"

int some_function(void *ptr){
    printf("I am thread #%d of SMG process %d",
                            (int)pthread_self(), SMG_rank);
    return 0;
}

int main (int argc, char *argv[]){
    int flag, error, i, j;

    // The request for information and monitoring systems are specified as
    // a bit-field in the flags field of the initialisation routine.
    flag = (INFORMATION_FLAG | MONITORING_FLAG);

    // Initialise the DSM, with no consistency model implementation.
    error = SMG_init(&argc, &argv, flag, NO_CONSISTENCY);
    if (error != SMG_SUCCESS){
        printf("SMG_init Unsuccessful!\n");
        exit(-1);
    }
```

```
    // Create a user thread that will be visible to the DSM, it
    // will execute some_function
    SMG_thread_create(&tid, &attr, &some_function, NULL)

    // Main user thread also calls the function
    some_function(NULL);

    // Wait for created thread to finish
    SMG_thread_join(tid, &exit_val);

    error = SMG_finalise();
    if (error != SMG_SUCCESS){
        printf("Error while calling SMG_finalise!\n");
    }

    return error;
}
```

void SMG_print_state ()

Print the current internal status of DSM engine.

Returns:

The SMG error status code

This routine will print to *stdout* a description of the internal state of the DSM engine which includes a count of all DSM resources used, and their status, by the user application - locks, barriers, threads, and shared memory regions.

SMG_module_load (int *MODULE_TYPE*, char * *location*)

Load a SMG module that provides extra DSM functionality.

Parameters:

MODULE_TYPE The type of DSM module to load

location Path to location of the object file with the required functionality

Returns:

The SMG error status code

The SMG DSM engine can be easily extended by developing extra functionality that implements the type of interface specified by *MODULE_TYPE*. Possible values for the module type are: SMG_MOD_CONSIST, SMG_MOD_COHEREN, SMG_MOD_COLLECT.

SMG_set_default_consistency (int *consistency*)

Set the default consistency to be used with any memory allocation.

Parameters:

consistency The default consistency type.

Returns:

The SMG error status code

This routine sets the default memory consistency type for any memory allocation not specifying a memory consistency type.

SMG_get_default_consistency ()

Get the default memory consistency.

Returns:

The default code for the default shared memory consistency.

SMG_have_consistency (int *consistency*)

Query the system to see if the specified consistency code is supported.

Parameters:

consistency The consistency to check to see if supported.

Returns:

The SMG error status code

SMG_local_response_time ()

Simple test to return the time for a local DSM system call.

Returns:

The time in milliseconds

Simple test that calculates the minimum response time to invoke a DSM operation in the local process (more/less the response time of the DSM engine). This utility function is again provided as a means for the user to perform simple benchmarking of the system.

SMG_remote_response_time (int *remote*)

Simple test to return the time for a remote DSM system call.

Parameters:

remote the remote process to take part in the call

Returns:

The time in milliseconds

Simple test that calculates the minimum response time to invoke an operation at a remote process with the identifier specified by *remote*. This utility function is provided as a means for the user to perform simple benchmarking of the system.

SMG_consistency_supported (int *type*, int * *supported*)

Test to see if a given consistency level is supported.

Parameters:

type

supported

Returns:

The SMG error status code

SMG_internal_get (int *key*, int * *value*)

Get the value of an internal DSM attribute.

Parameters:

key the internal item to obtain the value of

value location where to store the value of the specified item

Returns:

The SMG error status code

See also:

SMG_internal_set (p. 273)

This function is used to get the value of an internal DSM attribute specified by the *key* attribute. The internal attributes that can be queried govern the internal operation of the DSM engine, while really only use by for DSM protocol implementers, it is provided here for completeness.

SMG_internal_set (int *key*, int *value*)

Set the value of an internal DSM attribute.

Parameters:

key the internal item to obtain the value of

value the internal value to set the internal attribute to.

Returns:

The SMG error status code

Warning:

Use of this function with values unknown to the DSM system may have adverse side-effects.

See also:

SMG_internal_get (p. 272)

This routine together with its complementary function is used to query and set internal DSM engine attributes. Theses attributes or environment variables can change the default behaviour of DSM functionality where the default 'best case' does not prove adequate. This function while only for experienced developers and DSM implementers, is provided for completeness. E.g. this function can be used to alter the default number of internal DSM threads created by the engine.

SMG_get_comm_handle (int * *handle*)

Set a communication handle, *handle*, to the underlying communication infrastructure.

Parameters:

handle a pointer to the location where the communication handle is to initialised.

Returns:

The SMG error status code

Warning:

The underlying communication system should support multiple independent threads utilising the communication system concurrently, otherwise behaviour is undefined. Any handle returned by SMG_get_comm_handle ceases to valid once *SMG_finalise* is called by the application.

The function returns a communication handle for use with the underlying communication infrastructure that was initialised with the *SMG_init* call, and used internally by SMG DSM engine. This handle can be availed of in user code for accessing the communication system directly.

```
// This code snippet demonstrates use of the underlying communications

// Include the header file with SMG API definitions.
#include "smg_ec.h"

// Also include the
#include "mpi.h"

int main (int argc, char *argv[]){
    int flag, error, i, j, comm_handle;

    // The request for no information and monitoring systems are specified
    // as a bit-field in the flags field of the initialisation routine.
    flag = (NO_INFORMATION | NO_MONITORING);

    // Initialise the DSM, with entry consistency requested.
    error = SMG_init(&argc, &argv, flag, NO_CONSISTENCY);
    if (error != SMG_SUCCESS){
        printf("SMG_init Unsuccessful!\n");
        exit(-1);
    }

    error= SMG_get_comm_handle(&mpi_handle);
    if (error != SMG_SUCCESS){
        printf("Error obtaining communication handle!\n");
        exit(-1);
    }

    // Use the handle to access the underlying communication system.
    MPI_Barrier(comm_handle);

    error = SMG_finalise();
    if (error != SMG_SUCCESS){
        printf("Error while calling SMG_finalise!\n");
    }

    return error;
}
```

SMG_work_distribution (int *TYPE*, void * *from*, void * *to*, int *how*, void * *my_from*, void * *my_to*, void * *param*)

Divide workload among processes according to given criteria.

Parameters:

TYPE The primitive that the values *from* and *to* represent

from The start of the work distribution to divide among processes

to the end of the range of work to divide among processes

how the method of decomposing work among all processes

my_from The resulting start of the work distribution for the local process

my_to The end of the work distribution assigned to the local process

param Optional parameter to the work distribution function.

Returns:

The SMG error status code

Warning:

Due to rounding errors there may be some rounding errors associated with some implementations where the *TYPE* is a floating point number.

This routine is used to divide up the workload among all processes in the contiguous range *from* to *to* according to a specified strategy (specified using *how*, and can be one of the work distribution functions such as SMG_BLOCK or SMG_CYCLIC), and the support provided by the information system, if it was instantiated.

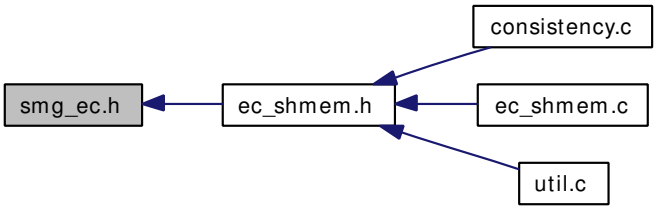
E.2.5 Variable Documentation

int SMG_proc_size

This specifies the number of processes that in the application.
This variable states the number of processes in the SMG DSM application, although the function *SMG_get_size* also exists, this variable is constant throughout the application i.e. this variable will specify the number of processes the application was started with. This will prove useful if dynamic process creation capabilities are added in the future.

int SMG_proc_rank

The rank of the process in the pool of processes.
This variable states the rank of the process in the SMG DSM application, although the function *SMG_get_rank* also exists, this variable is constant throughout the application i.e. this variable will specify the processes rank when the application was started with. This will prove beneficial if dynamic process creation capabilities are added in the future.



E.3 smg_ec.h File Reference

Additional definitions for use of Entry Consistency (EC) with SMG DSM System. This file also 'includes' the definitions in **smg.h** (p. 243).
`#include "smg.h"`

Defines

- #define **BIND_LOCK** 0x0000010

  A definition of the flag specified in a SMG_malloc *invocation that sets the system to bind memory to a lock synchronisation primitive.*

- #define **ALL_BARRIER** 0x0000020

  A definition of the flag that sets the system to bind memory to all barriers.

- #define **NAMED_BARRIER** 0x0000040

  The flag that sets the system to bind memory to a given barrier.

- #define **DYNAMIC_BARRIER** 0x0000080

  The flag that sets the system to bind memory to any barrier at run-time.

Functions

- int **SMG_memory_get_sync** (int identifier, int *type, int *prim_id)

  Get the bound synchronisation primitive for an EC shared memory region.

E.3.1 Detailed Description

Additional definitions for use of Entry Consistency (EC) with SMG DSM System. This file also 'includes' the definitions in **smg.h** (p. 243).
The SMG library is a DSM implementation that allows for the development of applications in a shared memory programming style. This file specifies extra flags necessary to develop shared-memory applications involving entry consistency. These extra definitions

specify what synchronisation primitive to bind to the use at the allocation of a shared memory region.

E.3.2 Define Documentation

#define BIND_LOCK 0x0000010

A definition of the flag specified in a *SMG_malloc* invocation that sets the system to bind memory to a lock synchronisation primitive.
Memory type allocation variables EC only

#define ALL_BARRIER 0x0000020

A definition of the flag that sets the system to bind memory to all barriers.

#define NAMED_BARRIER 0x0000040

The flag that sets the system to bind memory to a given barrier.

#define DYNAMIC_BARRIER 0x0000080

The flag that sets the system to bind memory to any barrier at run-time.

E.3.3 Function Documentation

int SMG_memory_get_sync (int *identifier*, int * *type*, int * *prim_id*)

Get the bound synchronisation primitive for an EC shared memory region.

Parameters:

identifier Numerical identifier of region to get information on.

type Type of synchronisation primitive that shared region bound to.

prim_id Primitive identifier of type specified by *type*.

Returns:

The SMG error status code

Warning:

The value obtained may be inconclusive if the process is not the owner of the lock when this function is invoked.

This routine obtains the identifier of the synchronisation primitive that the shared memory region is bound to. The type of the synchronisation primitive is specified by *type* field and could be one of *BIND_LOCK*, *ALL_BARRIER* etc.

E.4 SMG Source Manifest

The tables below (Tables E.2-E.7) give a brief synopsis of the source code files for the implementation of the SMG DSM, also given is a line count for each file. This manifest only includes SMG engine code from '.c' files, many of the files have corresponding '.h' header files (but are referred to in the include graphs below), but are omitted.
Figure E.3 below gives a pictorial representation of the include graph for *dsm.h*, the code for the main DSM engine core. Code graphs for other sections of the code are given below on page 282.

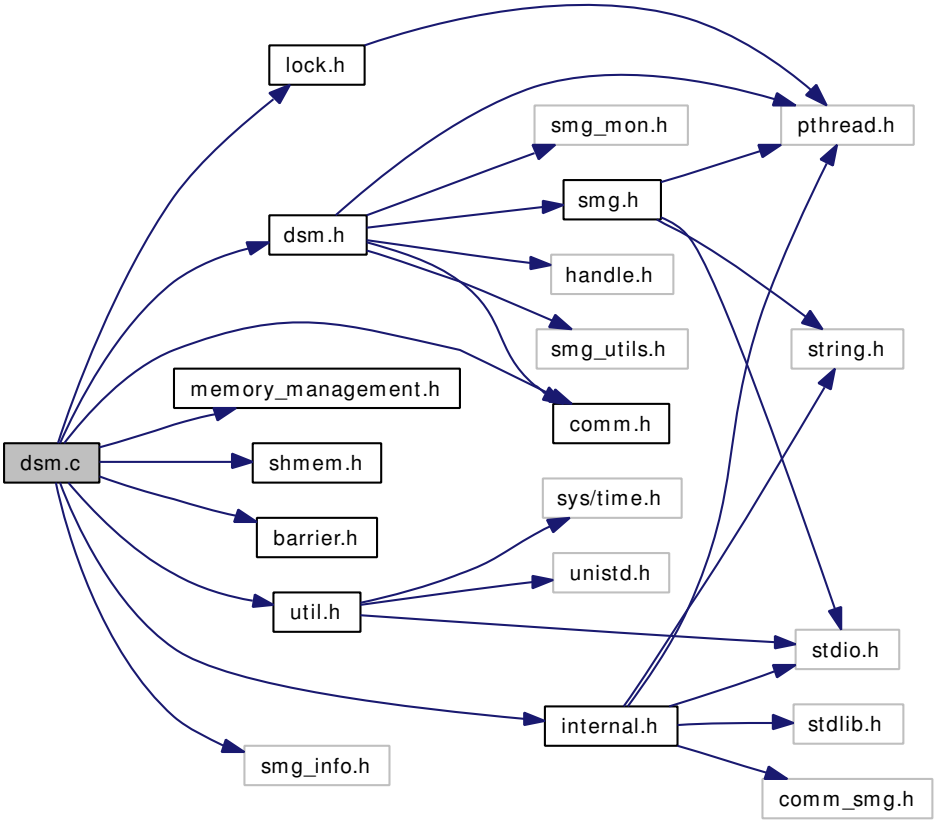


Figure E.3: *dsm.c source file include graph*

File Name	Description	Lines
abort.c	API Function implementations for DSM System abort routines	190
barrier.c	API Function implementations for DSM System barrier routines	1,430
coherence.c	Implementation for core DSM coherence operations	247
collection.c	Implementation for core DSM write-collection operations	225
comm.c	Communication management logic for the DSM engine	518
consistency.c	Implementation for core DSM consistency operations	220
dsm.c	Function implementations for DSM System engine entry exit	929
finalise.c	Implementation of finalisation API, plus cleanup routines for SMG	405
flush.c	Implementation of flush operation of shared memory	576
ft.c	Initial support for fault tolerance in SMG	160
info.c	Implementation of core information-system operations for SMG	466
init.c	Implementation of initialisation API, plus start-up routines for SMG	640
internal.c	Internal API implementation and functionality for DSM engine code	432
lock.c	Implementation of SMG lock API	1966
memory_management.c	Implementation of functions required for internal memory management performed by the DSM System engine	1003
pointer.c	Implementation of pointer/reference routines in the SMG API	194
shmem.c	Implementation of memory management routines of the SMG API	1759
thread.c	User multi-threading support in the SMG API	427
updates.c	Core DSM logic to process updates within the DSM engine	577
util.c	Generic routines utilised by many sections of the DSM implementation	929

Table E.2: *SMG DSM Core Engine Code*

File Name	Description	Lines
linux-2_6.c	Operating system abstraction layer implementation for Linux 2.4 and 2.6 kernels	458

Table E.3: *SMG OS independence*

File Name	Description	Lines
coher_inval.c	Implementation for invalidation coherence protocol	778
coher_subscr.c	Implementation of subscription protocol for SMG DSM System	1801
coher_update.c	Implementation of update coherence protocol for SMG DSM System	423
collect_raw.c	Implementation of 'raw' write-collection strategy in the SMG DSM System	594
diff.c	Implementation of 'diff' write-collection strategy in the SMG DSM System	1100
ec_shmem.c	Extra memory management routines providing the EC support in SMG	902

Table E.4: *SMG Consistency Implementations*

File Name	Description	Lines
comm_smg_Basic.c	Generic routines used by all communication implementations	202
comm_smg_mpi.c	Single-threaded MPI communication system implementation for DSM engine	826
comm_smg_mpi_t.c	Multi-threaded MPI communication system implementation for DSM engine	894
comm_smg_mpi_sim_t.c	Multi-threaded MPI communication system implementation with latency simulation for DSM engine	1087
comm_smg_pvm.c	Single-threaded PVM communication system implementation for DSM engine	720

Table E.5: *SMG communication*

File Name	Description	Lines
info-rgma-2.4.c	Information system implementation using R-GMA API from LCG-2.4	823
info-rgma-2.6.c	Information system implementation using R-GMA API from LCG-2.6	1174
info-rgma-2.7.c	Information system implementation using R-GMA API from LCG-2.7/gLite-3.0	1282
info-rgma-share.c	Shared generic routines used across R-GMA implementations	138
info-file.c	File-based information system implementation	974
info-null.c	A *NULL* back-end implementation for the information system	317
mon-rgma-2.4.c	Monitoring system implementation using R-GMA API from LCG-2.4	201
mon-rgma-2.6.c	Monitoring system implementation using R-GMA API from LCG-2.6	254
mon-rgma-2.7.c	Monitoring system implementation using R-GMA API from LCG-2.7/gLite-3.0	196
mon-file_t.c	Monitoring system implementation using log-files directly	210
mon-null.c	A *NULL* back-end implementation for the monitoring system	164
rgma tools	Common utility code for R-GMA based services	978

Table E.6: *SMG Information & Monitoring*

File Name	Description	Lines
bitarray.c	Bit-field/array utility functions for manipulating these structures	328
data_init.c	Routines for verifying integrity of shared memory (debugging)	196
handle.c	Generic structure used with the DSM engine and related support	576
proc_status.c	Routines for accessing process information(debug support	147
smg_utils.c	Timing routines used internally by the DSM (for debugging)	175

Table E.7: *SMG Utility Code*

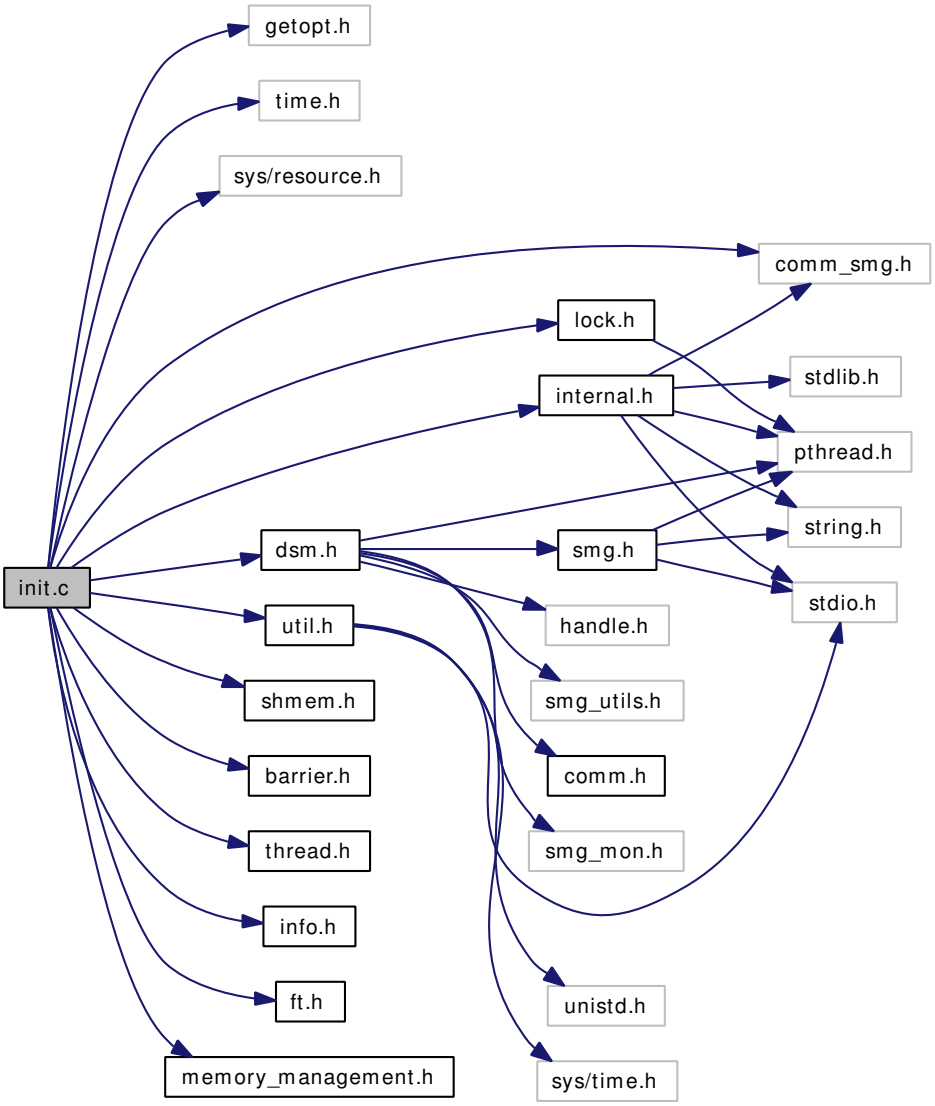


Figure E.4: *Initialisation code include graph*

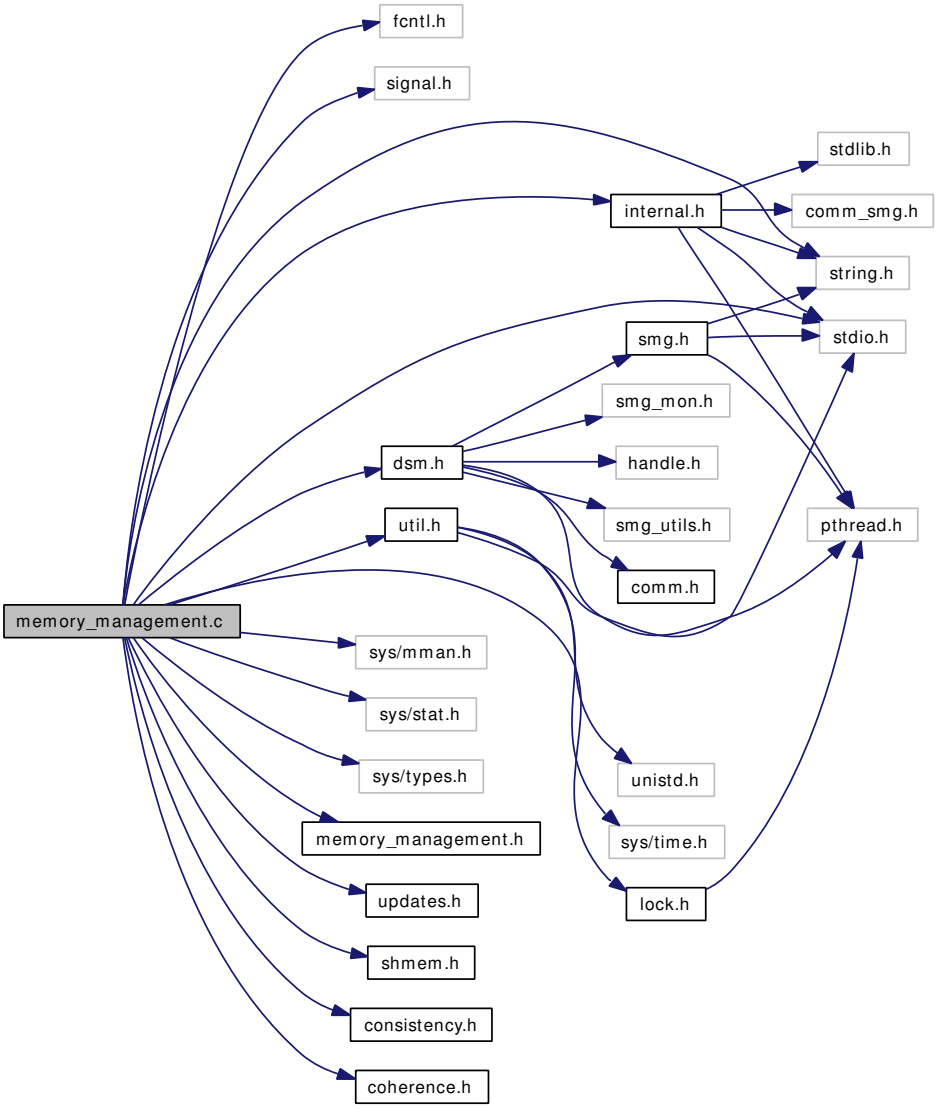


Figure E.5: *Memory management code include graph*

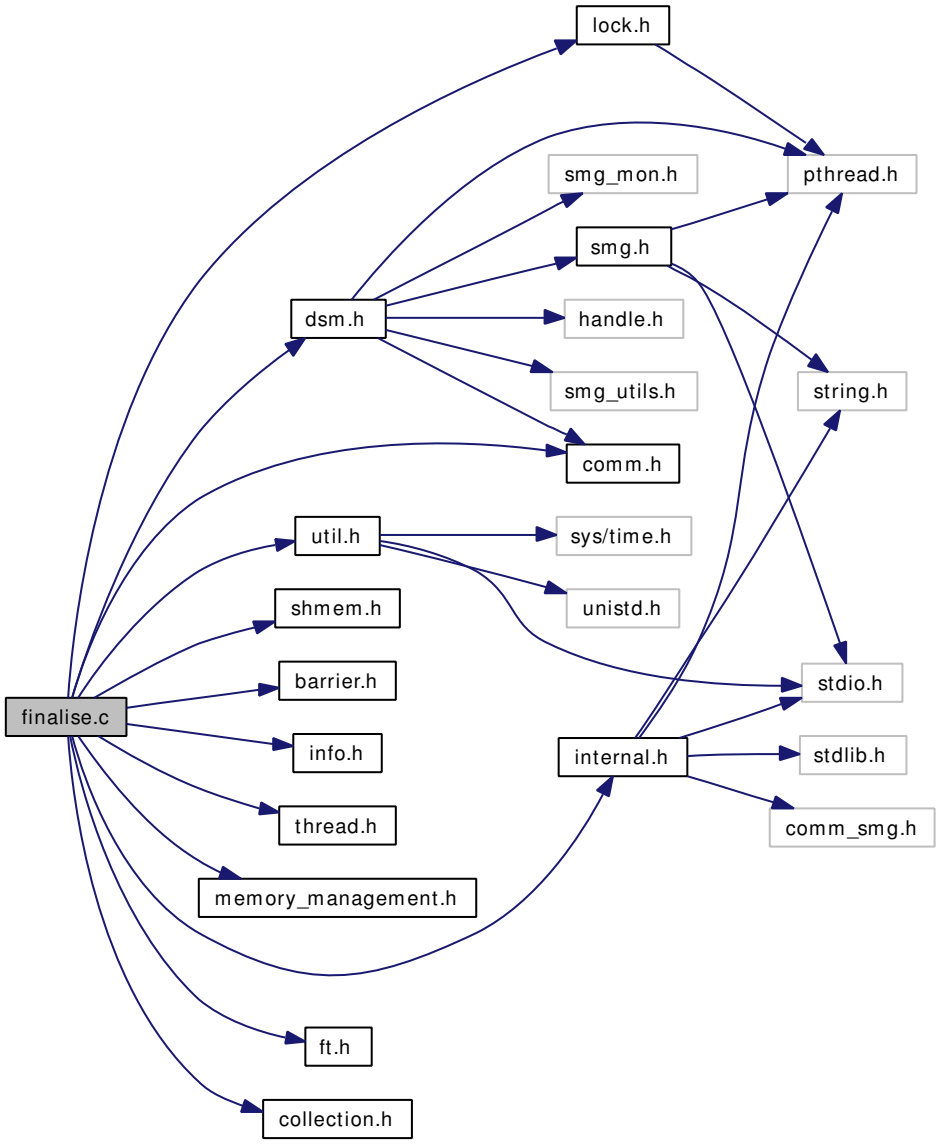


Figure E.6: *finalise.c code include graph*

E.5 Extending SMG

SMG can be extended in many areas, most noticeably in the areas of consistency models, coherence protocols, write-collection strategies, new communication sub-systems, information, and monitoring services by implementing the required interface as defined by the files - *consistency.h*, *coherence.h*, *collection.h*, *comm.h*, *info.h*, and *mon.h* respectively. The additional modules can be compiled as shared object, '*.so*', files and loaded at runtime either (a) explicitly using the *SMG_module_load* API call, or (b) by specifying the path to the shared object file and module type (as specified in) pairing in the *.smgrc* user configuration file that is located in the user's home directory.
It must be noted that extending the SMG DSM can be demanding at best and requires a deep understanding of its internal workings, this fact is demonstrated by the include graph for the entry consistency implementation shown in Figure E.7.

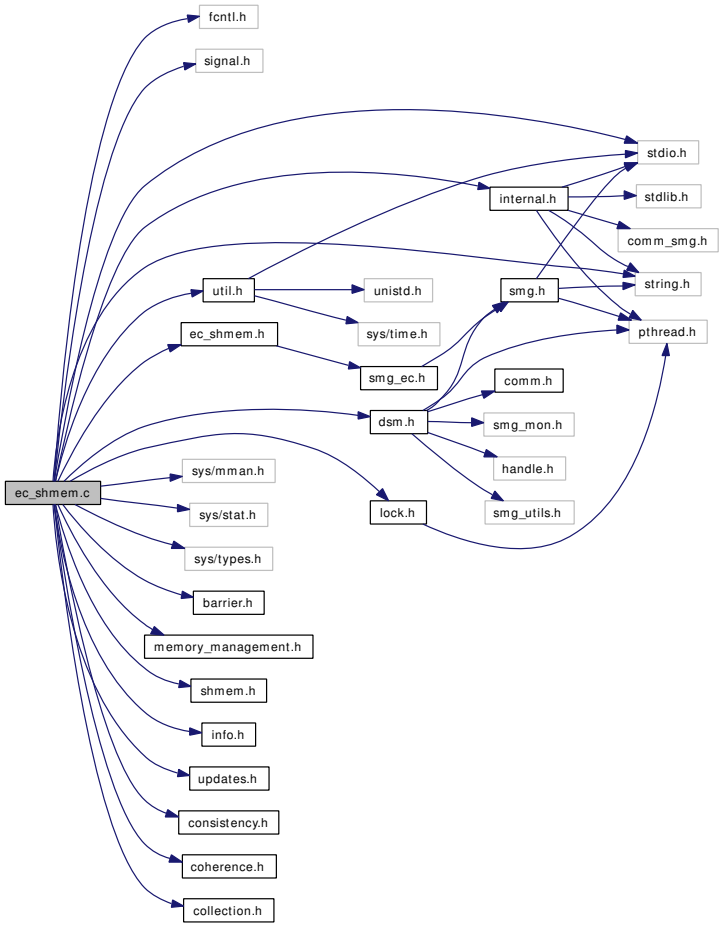


Figure E.7: *ec_shmem.c code include graph*

APPENDIX F

SMG Applications

This Appendix lists some SMG sample applications that were developed using the SMG API. These three applications solve similar computation tasks such as matrix-multiply and Laplace, but each uses `different` constructs of the SMG API to implement the problem. For the sake of brevity error checking of SMG function invocations are not performed.

SMG Example Code (Locks)

The following code listing implements a MRSW approach to matrix multiplication using SMG locks. This implementation will perform poorly compared to the other versions below as only one process may access the shared region to store that processes part of the result data at any given time. It is primarily provided to demonstrate the MRSW functionality offered by SMG .

Synopsis: The application, once initialisation has taken place (Line 30), allocates three shared memory regions to store the two matrices to be multiplied, `A` and `B`, and the resultant matrix `C`. The former two are initialised to defined values and are bound to the use of one particular lock which will later be acquired in read (non-exclusive) mode by all processes concurrently, while the resultant matrix is bound to a different lock primitive. Once a process has acquired the lock used to guard access to the incident matrices they proceed to perform the matrix multiplication for their portion of the resultant matrix. The output is temporarily buffered as all processes cannot concurrently hold the lock guarding the result matrix `C`. Each process will acquire this lock in turn and update their portion of the shared memory result matrix from the buffer. All processes will subsequently wait at a barrier until all processes have completed this task.

```
#include <stdio.h>

/* Include the SMG API definitions with entry consistency (extensions).
 */
#include "smg_ec.h"

#define DIM 512
#define A_ID 1
#define B_ID 2
#define OFFSETS 3
#define C_ID 4
#define CALCS 5
#define AB_LOCK 1
#define C_LOCK 2

int main(int argc, char **argv){
    int i, j, k, count, error;
    int buffer[DIM][DIM];
    int **a, **b, **c;
    int *temp, *offsets;

    long long testlong;
    int my_first, my_last, num_rows, rem;
    int rows, dest, calcs = 0;


    /* Initialise the SMG environment application with support for
     * information but not monitoring services. Specify EC is required.
     */
```

```
30  SMG_init(&argc, &argv, INFORMATION_FLAG, ENTRY_CONSISTENCY);

32  if(SMG_proc_rank == 0){
      SMG_write_lock_acquire(AB_LOCK);
34    SMG_write_lock_acquire(C_LOCK);
    }
36
    a = (int**)malloc(DIM * sizeof(int*));
38  b = (int**)malloc(DIM * sizeof(int*));
    c = (int**)malloc(DIM * sizeof(int*));
40

42  /* Allocate shared memory for the two incident matrices A & B,
     * and create 2-dimensional arrays using this space.
44   */
    SMG_shmem_malloc(A_ID, (DIM * DIM * sizeof(int)),
46                           (void *)&temp, 0, AB_LOCK);
    a[0] = temp;
48  for (i = 1; i < DIM; i++) {
      a[i] = a[i-1] + DIM;
50  }
    SMG_shmem_malloc(B_ID, (DIM * DIM * sizeof(int)),
52                           (void *)&temp, 0, AB_LOCK);
    b[0] = temp;
54  for (i = 1; i < DIM; i++) {
      b[i] = b[i-1] + DIM;
56  }

58  /* Allocate a buffer to store values for the work distribution
     * among processes.
60   */
    SMG_shmem_malloc(OFFSETS, ((SMG_proc_size + 1) * sizeof(int)),
62                   (void *)&offsets, 0, AB_LOCK);

64
    /* Allocate the shared memory ho hold the resultant matrix C, bind
66   * it to the use of synchronisation primitive C_LOCK.
     */
68  SMG_shmem_malloc(C_ID, (DIM * DIM * sizeof(int)),
                     (void *)&temp, 0, C_LOCK);
70  c[0] = temp;
    for (i = 1; i < DIM; i++) {
72    c[i] = c[i-1] + DIM;
    }
74

76  /* Let the master process initialise the incident matrices A & B.
     */
78  if(SMG_proc_rank == 0){
      // Init the grid
80    for (i=0; i<DIM; i++){
        for (j=0; j<DIM; j++){
82        a[i][j]= i + j;
```

```
        b[i][j]= i * j;
84    }
    }
86
    num_rows = (int)(DIM / SMG_proc_size);
88  rem  = DIM % SMG_proc_size;
    offsets[0] = 0;
90
    /* Distribute the work fairly among all processes
92   */
    for(dest = 1; dest < SMG_proc_size; dest++){
94    if(rem != 0){
        rows = num_rows + 1;
96      rem--;
      }else{
98      rows = num_rows;
      }
100   offsets[dest] = offsets[dest-1] + rows;
    }
102 offsets[SMG_proc_size] = DIM;

104 SMG_lock_unlock(AB_LOCK);
    SMG_lock_unlock(C_LOCK);
106 }

108
  /* Synchronise all processes at this point
110  */
  SMG_barrier(0, SMG_DEFAULT_TYPE);
112
  SMG_read_lock_acquire(AB_LOCK);
114 my_first = offsets[SMG_proc_rank];
  my_last  = (offsets[SMG_proc_rank + 1])-1;
116

118 /* Then begin the computation
   */
120 for (i = 0; i < DIM; i++){
    for (j = my_first; j < my_last; j++){
122   buffer[j][i] = 0;
      for (k = 0; k < DIM; i++){
124     buffer[i][k] += a[i][j] * b[j][k];
      }
126   }
  }
128

130 /* Each process will now acquire the lock guarding the resultant matrix
   */
132 SMG_write_lock_acquire(C_LOCK);
  for (i = 0; i < DIM; i++){
134   for (j = my_first; j < my_last; j++){
      c[j][i] = buffer[j][i];
```

```
136       }
      }
138
140   /* Release lock and hand over to another process.
       */
142   SMG_lock_unlock(C_LOCK);
      SMG_lock_unlock(AB_LOCK);
144
146   /* All processes synchronise at this point
       */
148   SMG_barrier(1,0);

150
      /* The master process will then acquire the lock to ensure it has
152    * a complete and consistent result-set.
       */
154   if(SMG_proc_rank == 0){
        SMG_read_lock_acquire(C_LOCK);
156     SMG_lock_unlock(C_LOCK);
      }
158
      /* Clean-up the SMG environment and exit.
160    */
      SMG_finalise();
162
      return 0;
164 }
```

SMG Example Code (Barriers)

The following code listing demonstrates the MRMW capabilities of SMG by performing Laplace using barriers for synchronisation.

Synopsis: The application first initialises the SMG environment with a call to *SMG_init*. The use of the underlying information and monitoring system is requested, and will be used if available. The shared memory space for the 2-d matrix used for the computation is allocated and bound to the use of the specified barrier primitive. The master process then initialises the grid to its initial state which to perform the Laplace operation on. The barrier is invoked which results in coherence operations distributing the initial grid among all processes in the application. Next, all processes proceed to solve their section of the grid, at the barrier all partial results performed by a given process will be distributed to all others, this operation will be repeated for a given number of iterations (`MAXITER` iterations, unless specified by user on command line).

```
#include <stdio.h>

/* Include the SMG API definitions with entry consistency (extensions).
 */
#include "smg_ec.h"

#define XSIZE 1024
#define YSIZE XSIZE
#define MAX(x,y) ( ((x) > (y)) ? x : y )

#define PI 3.1415927
#define NITER 10
#define MAXITER 101
#define tolerance  0.1E-3
#define U_GRID_ID 3
#define GRID_BARRIER 4

int lap_main();

double temp_matrix[XSIZE][YSIZE];
double *u_grid;
double **u;
double time1,time2;
double sum, dt;
int first, last, niter, checksum;

int main(int argc, char **argv){
  int x, y, i, error, flag;

  checksum = 0;
  flag = (INFORMATION_FLAG | MONITORING_FLAG);

  /* Initialise the SMG environment application with support for both
   * information and monitoring services. Specify EC is required.
```

```
     */
36   error = SMG_init(&argc, &argv, flag, ENTRY_CONSISTENCY);

38   time1 = time(NULL);

40   if(argc > 1){
       niter = atoi(argv[1]);
42   }else{
       niter = MAXITER;
44   }

46   first = SMG_proc_rank * (XSIZE / SMG_proc_size);
     last  = ((SMG_proc_rank+1) * (XSIZE / SMG_proc_size)) - 1;
48   if(first == 0){
       first++;
50   }
     if(last == (XSIZE-1)){
52     last--;;
     }
54
     // malloc u, barrier (ID = barrier_id), named,
56   error = SMG_shmem_malloc(U_GRID_ID, (XSIZE * YSIZE * sizeof(double)),
                              (void**)&u_grid, (ENTRY | NAMED_BARRIER),
58                            GRID_BARRIER);

60   /* Map shared memory area, u_grid, into the 2-dimensional array, u.
      */
62   u = (double**)malloc(XSIZE * sizeof(double*));
     u[0] = u_grid;
64   for (i = 1; i < YSIZE; i++) {
         u[i] = u[i-1] + YSIZE;
66   }

68   /* The master process will initialise the grid to a specified state
      */
70   if(SMG_proc_rank == 0){

72     for(x = 1; x < (XSIZE - 1); x++){
         for(y = 1; y < (YSIZE - 1); y++){
74           u[x][y] = sin((double)(x-1)/XSIZE*PI)
                       + cos((double)(y-1)/YSIZE*PI);
76       }
       }
78
       for(x = 0; x < (XSIZE); x++){
80       u[x][0] = 0.0;
         u[x][YSIZE-1] = 0.0;
82     }

84     for(y = 0; y < (YSIZE); y++){
         u[0][y] = 0.0;
86       u[XSIZE-1][y] = 0.0;
       }
```

```
88    }

90    /* Synchronise the initial grid across all processes in the application
       */
92    SMG_barrier(GRID_BARRIER,0);

94    SMG_print_state();

96    printf("(%2d) start: %4d, end: %4d\n",
              SMG_proc_rank, first, last);
98
      for(x = 0; x < (XSIZE); x++){
100     for(y = 0; y < (YSIZE); y++){
          temp_matrix[x][y] = u[x][y];
102     }
      }
104
      /* Enter the main Laplace routine
106    */
      lap_main();
108
      time2 = time(NULL);
110   printf("time=%g\n", difftime(time2, time1));

112   SMG_print_state();

114
      /* Clean-up the SMG environment and exit.
116    */
      SMG_finalise();
118
      return 0;
120 }

122
    int lap_main(){
124   int     x,y,k;

126   for(k = 0; k < niter; k++){
        /* Perform the allotted portion of the work to this process.
128      */
        for(x = first; x <= last ; x++){
130       for(y = 1; y < YSIZE; y++){
            u[x][y] = (temp_matrix[x-1][y] + temp_matrix[x+1][y] +
132                    temp_matrix[x][y-1] + temp_matrix[x][y+1]) / 4.0;
          }
134     }

136     /* Synchronise partial results across all processes
         */
138     SMG_barrier(GRID_BARRIER,0);

140     /* Perform a quick check-sum to check for convergence
```

```
        */
142     dt  = 0.0;
        sum = 0.0;
144
        for(x = 1; x < (XSIZE - 1); x++){
146       for(y = 1; y < (YSIZE); y++){
            sum += (temp_matrix[x][y]- u[x][y]);
148         dt = MAX( fabs(u[x][y] - temp_matrix[x][y]), dt);
            temp_matrix[x][y] = u[x][y];
150       }
        }
152
        if( (k% NITER) == 0 && SMG_proc_rank == 0) {
154       printf("(%2d) [%4d] sum = %3.6f, dt = %3.6f\n",
                SMG_proc_rank, k, sum, dt);
156     }
    }
158
    if(dt > tolerance && SMG_proc_rank == 0){
160   printf("\nEnd [%4d] sum = %3.6f, dt = %3.6f\n",
            k, sum, dt);
162 }

164 return 0;
}
```

SMG Example Hybrid Code

The following code listing implements a hybridised version of the Laplace application with the main communication overhead caused by the DSM system now removed and replaced with MPI routines (see Lines 173-205). The same functionality as that given on page 292. However, this (rather contrived) version aims to demonstrate the hybridisation features of SMG that enables MPI code to be inserted where the DSM overhead dictates that it should, as the MPI code modifies the shared region DSM sharing must be turned off before MPI calls access it (see Line 156), and turned on when no more MPI code is required (Line 225).

Synopsis: This application starts by first initialising the system with the invocation of *SMG_init* requesting for the information and monitoring features to be employed. Once this has completed a handle to the underlying communication system is obtained using the *SMG_get_comm_handle* routine, this handle will be used subsequently for inserting hybrid-MPI code into the application. The shared memory storing the 2-dimensional grid for the application is allocated and bound to the use of the barrier primitive `GRID_BARRIER`.
The master process initialises the grid to a defined state and all barriers then call a barrier routine on the primitive which syncs the application's grid, thus the initialised grid is consistent across all processes. At the entry of the Laplace function the DSM-managed sharing of the grid is disabled using the *SMG_shmem_noshare* API call, and the application proceeds to modify the grid, and then share border rows of the grid with adjacent processes using MPI routines. When the application has performed the specified number of iterations, by *niter*, then the shared grid is validated with a call to *SMG_shmem_valid* to ensure there is no transient conflicts between local cached versions of the shared grid between different processes, then DSM sharing of the grid is re-enabled using the *SMG_shmem_share* routine. Finally the DSM finalisation routine is called to clean-up the DSM environment and the underlying communication system, after this point MPI call may no longer be invoked.

```
1 /* Include the SMG API definitions with entry consistency (extensions),
   * also include MPI as this is a hybrid code.
3  */
  #include "smg_ec.h"
5 #include "mpi.h"

7 #define XSIZE 1024
  #define YSIZE XSIZE
9 #define MAX(x,y) ( ((x) > (y)) ? x : y )

11 #define PI 3.1415927
  #define NITER 10
13 #define MAXITER 1
  #define tolerance  0.1E-3
15 #define U_GRID_ID 3
  #define GRID_BARRIER 4
```

```
17

19 int lap_main();

21
   double uu[XSIZE][YSIZE];
23 double *u_grid;
   double **u;
25 double time1,time2;
   double sum, dt;
27
   int comm_handle;
29 MPI_Status status;

31 int first, last, niter, checksum;
   int  messages_sent, data_sent, temp_data, temp_mess;
33

35 int main(int argc, char **argv){
     int x, y, i, error, flag;
37
     messages_sent = data_sent = 0;
39   checksum = 0;
     flag = (INFORMATION_FLAG | MONITORING_FLAG);
41
     /* Initialise the SMG environment application with support for both
43    * information and monitoring services. Specify EC is required.
      */
45   error = SMG_init(&argc, &argv, flag, ENTRY_CONSISTENCY);
     time1 = time(NULL);
47

49   /* Get a handle to the underlying communication system, in this case
      * that is provided by an MPI implementation.
51    */
     error = SMG_get_comm_handle(&comm_handle);
53
     if(argc > 1){
55     niter = atoi(argv[1]);
     }else{
57     niter = MAXITER;
     }
59

61   /* Devise this processes share of the work to be done
      */
63   first = SMG_proc_rank * (XSIZE / SMG_proc_size);
     last  = ((SMG_proc_rank+1) * (XSIZE / SMG_proc_size)) - 1;
65   if(first == 0){
       first++;
67   }
     if(last == (XSIZE-1)){
69     last--;;
```

```
    }
71

73  /* Allocate the shared memory for the application's grid, bind the
     * grid to the use of the barrier primitive with id = GRID_BARRIER.
75   */
    error = SMG_shmem_malloc(U_GRID_ID,(XSIZE*YSIZE * sizeof(double)),
77                   (void**)&u_grid, (ENTRY | NAMED_BARRIER),
                      GRID_BARRIER);
79
    /* Map the allocated shared memory into a nice 2-d array format
81   */
    u = (double**)malloc(XSIZE * sizeof(double*));
83  u[0] = u_grid;
    for (i = 1; i < YSIZE; i++) {
85      u[i] = u[i-1] + YSIZE;
    }
87

89  /* Let the master process initialise the grid
     */
91  if(SMG_proc_rank == 0){

93    for(x = 1; x < (XSIZE - 1); x++){
        for(y = 1; y < (YSIZE - 1); y++){
95          u[x][y] = sin((double)(x-1)/XSIZE*PI) +
                                cos((double)(y-1)/YSIZE*PI);
97      }
      }
99
      for(x = 0; x < (XSIZE); x++){
101     u[x][0] = 0.0;
        u[x][YSIZE-1] = 0.0;
103   }

105   for(y = 0; y < (YSIZE); y++){
        u[0][y] = 0.0;
107     u[XSIZE-1][y] = 0.0;
      }
109 }

111 /* Synchronise the grid across all processes.
     */
113 SMG_barrier(GRID_BARRIER,0);

115 for(x = 0; x < (XSIZE); x++){
      for(y = 0; y < (YSIZE); y++){
117     uu[x][y] = u[x][y];
      }
119 }

121 SMG_print_state();
```

```
123   /* Perform the Laplace operation
       */
125   lap_main();

127   time2 = time(NULL);
      printf("time=%g\n", difftime(time2, time1));
129
      SMG_print_state();
131
      MPI_Allreduce( &data_sent, &temp_data, 1,
133                  MPI_INT, MPI_SUM, comm_handle);
      MPI_Allreduce( &messages_sent, &temp_mess, 1,
135                  MPI_INT, MPI_SUM, comm_handle);
      if (SMG_proc_rank == 0){
137     printf("Total data sent:%d (%d)\n",temp_data,data_sent);
        printf("Total messages sent:%d\n", temp_mess);
139   }

141   /* Clean-up the SMG environment and exit.
       */
143   SMG_finalise();

145   return 0;
    }
147

149 int lap_main(){
      int     x,y,k;
151   double temp_sum, temp_dt;

153   /* Turn off DSM managed sharing of the shared region holding the
       * grid to perform the Laplace operation on
155    */
      SMG_shmem_noshare(U_GRID_ID);
157

159   /* For the specified number of iterations perform the relaxation
       */
161   for(k = 0; k < niter; k++){

163     for(x = first; x <= last ; x++){
          for(y = 1; y < YSIZE; y++){
165           u[x][y] = (uu[x-1][y] + uu[x+1][y] + uu[x][y-1]
                       + uu[x][y+1]) / 4.0;
167       }
        }
169
        /* Share my locally modified perimeter cells with neighbouring
171      * processes (where applicable) using MPI routines.
         */
173
        if (SMG_proc_rank > 0){
175       MPI_Recv( u[first-1], XSIZE, MPI_DOUBLE,
```

```
                (SMG_proc_rank - 1), 0, comm_handle, &status);
177       MPI_Send( u[first], XSIZE, MPI_DOUBLE,
                (SMG_proc_rank - 1), 1, comm_handle );
179       data_sent    += (XSIZE * sizeof(double));
          messages_sent++;
181     }

183     if (SMG_proc_rank < SMG_proc_size - 1){
          MPI_Send( u[last], XSIZE, MPI_DOUBLE,
185               (SMG_proc_rank + 1), 0, comm_handle );
          MPI_Recv( u[last+1], XSIZE, MPI_DOUBLE,
187               (SMG_proc_rank + 1), 1, comm_handle, &status);
          data_sent    += (XSIZE * sizeof(double));
189       messages_sent++;
        }
191
        for(x = first; x <= last; x++){
193       //    for(x = 1; x < (XSIZE - 1); x++){
          for(y = 1; y < (YSIZE); y++){
195         sum += (uu[x][y]- u[x][y]);
            dt = MAX( fabs(u[x][y] - uu[x][y]), dt);
197         uu[x][y] = u[x][y];
          }
199     }

201     MPI_Allreduce( &sum, &temp_sum, 1, MPI_DOUBLE, MPI_SUM,
                  comm_handle);
203     MPI_Allreduce( &dt, &temp_dt, 1, MPI_DOUBLE, MPI_MAX,
                  comm_handle);
205

207     if( (k% NITER) == 0 && SMG_proc_rank == 0){
          printf("(%2d) [%4d] sum = %3.8f, dt = %3.8f\n",
209             SMG_proc_rank, k, sum, dt);
        }
211   } // for

213   SMG_print_state();
      printf("(%2d) Out of MPI hybrid loop\n", SMG_proc_rank);
215
      /* Validate the shared region to remove potential inconsistencies
217    * between processes.
       */
219   SMG_shmem_valid(U_GRID_ID, u[first],
                  (sizeof(double) * YSIZE * ((last - first)+1) ) );
221   printf("(%2d) U Validated\n", SMG_proc_rank);

223   /* Re-enable DSM-managed sharing of the shared memory region
       */
225   SMG_shmem_share(U_GRID_ID);
      printf("(%2d) U shared again\n", SMG_proc_rank);
227
      /* Synchronise the shared region using the barrier
```

```
229     */
        SMG_barrier(GRID_BARRIER,0);
231
        printf("(%2d) Shared + barrier() = U consistent\n",
233             SMG_proc_rank);

235     if(dt > tolerance){
          if(SMG_proc_rank == 0){
237         printf("\nEnd [%4d] sum = %3.8f, dt = %3.8f\n",
                    k, sum, dt);
239       }
        }
241     return 0;
    }
```

GLOSSARY

ADI	Abstract Device Interface, 223
API	Application Programming Interface, 2
BST	Binomial Spanning Tree, 127
CAF	Co-Array FORTRAN, 17
CE	Compute Element, 28
CPU	Central Processing Unit, 13
DSM	Distributed Shared Memory, 2
EC	Entry Consistency, 48
EGEE	Enabling Grids for E-sciencE, 28
FT-MPI	Fault Tolerant MPI, 225
GGF	Global Grid Forum, 66
GMA	Grid Monitoring Architecture , 66
GSI	Grid Security Infrastructure, 62
HCG	Homogeneous Computational Grid, 31
HOT	Hierarchical Optimised Tree, 127
HPC	High Performance Computing, 64
HPF	High Performance FORTRAN, 17, 24
jdl	job description language, 87
LDAP	Lightweight Directory Access Protocol, 227
LRC	Lazy Release Consistency, 46

MDS Monitoring and Discovery Service, 227
MIMD Multiple Instruction Multiple Data, 12
MISD Multiple Instruction Single Data, 12
MPI Message Passing Interface, 64
MPMD Multiple Program Multiple Data, 16
MPP Massively Parallel Processor, 15
MRMW Multiple-Reader, Multiple-Writer, 37
MRSW Multiple-Reader, Single-Writer, 37
MTU Maximum Transfer Unit, 49

NFS Network File System, 65
NORMA No Real Remote Memory Access, 15
NOW Network of Workstations, 24
NUMA Non-Uniform Memory Access, 14

OpenMP Open Multi Processor, 68

PVM Parallel Virtual Machine, 64

R-GMA Relational Grid Monitoring Architecture, 67
RC Release Consistency, 45
RMA Remote Memory Access, 15
RPC Remote Procedure Call, 65
rsh *r*emote *sh*ell, 71

SCI Scalable Coherent Interconnect, 224
SE Storage Element, 28
SIMD Single Instruction Multiple Data, 12
SISD Single Instruction Single Data, 12
SMG Shared Memory for Grids, 32
SMP Symmetric Multi Processor, 14
SPMD Single Program Multiple Data, 16
SRMW Single-Reader, Multiple-Writer, 37
SRSW Single-Reader,Single-Writer, 37

TCP Transmission Control Protocol, 61

UDP Unreliable Delivery Protocol, 61
UMA Uniform Memory Access, 14
UPC Unified Parallel C, 17

VCS Version Control System, 143

REFERENCES

[1] Trinity Centre for High Performance Computing (TCHPC). http://www.tchpc.tcd.ie.

[2] Irish Center for High-End Computing (ICHEC). http://www.ichec.ie.

[3] SETI@home. http://setiathome.ssl.berkeley.edu.

[4] Homepage of OpenMP initiative. http://www.openmp.org.

[5] Lorna Smith and Mark Bull. Development of Mixed Mode MPI / OpenMP Applications. *Scientific Programming*, 9:83–98, 2000. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

[6] Folding@Home. http://www.stanford.edu/group/pandegroup/folding. (Website).

[7] K. Hwang and Z. Xu. *Scalable Parallel Computing: Technology, Architecture, Programming*. McGraw-Hill, 1997.

[8] Introduction to Parallel Computing (Tutorial). http://www.llnl.gov/computing/tutorials/parallel_comp.

[9] Intel Pentium 4 Architecture. http://www.intel.com/design/pentium4/manuals/index_new.htm.

[10] Robert G. Brown. Maximizing Beowulf Performance. In *4th Annual Linux Showcase & Conference*, pages 329–340, 2000. citeseer.ist.psu.edu/422752.html.

[11] J.L. Gustafson, G. Montry, and R. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal of Scientific and Statistical Computing*, 9(4):532–533, July 1988.

[12] Boyko Kakaradov. Ultra-Fast Matrix Multiplication: An Empirical Analysis of Highly Optimized Vector Algorithms. Technical report, Stanford University, 2004.

[13] A.Y. Grama, A. Gupta, and V. Kumar. Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures. *IEEE Parallel and Distributed Technology: Systems and Applications*, 1(3):12–21, August 1993.

[14] Alan H. Karp and Horace P. Flatt. Measuring Parallel Processor Performance. *Communications of the ACM*, 33(5):539–543, 1990.

[15] Xian-He H. Sun and Lionel M. Ni. Scalable Problems and Memory-Bounded Speedup. *Journal of Parallel and Distributed Computing*, 19(1):27–37, 1993.

[16] M.R. Zargham. *Computer Architecture: Single and Parallel Systems.* Prentice Hall, 1996.

[17] M. J. Flynn. Some Computer Organizations and Their Effectiveness. In *IEEE Transactions on Computing C-21*, volume 9, pages 948–960, Sept 1972.

[18] Eric E. Johnson. Completing an MIMD Multiprocessor Taxonomy. *SIGARCH Computing Architecture News*, 16(3):44–47, 1988.

[19] Programming Model Employed In The Earth Simulator. `http://www.es.jamstec.go.jp/esc/eng/Programming/programming.html`.

[20] IEEE standard for Scalable Coherent Interface (SCI), August 1992. E-ISBN: 0-7381-1204-6.

[21] Myrinet. `http://www.myri.com/`.

[22] Infiniband Trade Assocaition. Website. `http://www.infinibandta.org`.

[23] A. Agarwal, R. Bianchini, D. Chaiken, F. T. Chong, K. L. Johnson, D. Kranz, J. D. Kubiatowicz, B-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):430–444, 1999.

[24] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann Publishers, 1999.

[25] High Performance Fortran Forum. High Performance Fortran language specication, version 1.0. Technical Report CRPC-TR92225, Houston, Texas.

[26] Robert W. Numrich and John Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[27] W Carlson, J.M Draper, D.E Culler, K Yelick, E Brooks, and K Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, George Washington University, 1999.

[28] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha. ParADE: An OpenMP Programming Environment for SMP Cluster Systems. In *Proceedings of ACM/IEEE Supercomputing (SC'03)*, Nov 2003.

[29] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2007.

[30] T. Hoefler, L. Cerquetti, T. Mehlan, F. Mietke, and W. Rehm. A Practical Approach to the Rating of Barrier Algorithms Using the LogP Model and OpenMPI. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 562–569, 06 2005.

[31] Shun Yan Cheung and Vaidy S. Sunderam. Performance of Barrier Synchronization Methods in a Multi-Access Network. In *International Conference on Computing and Information*, pages 175–179, 1993.

[32] F. Mueller. Decentralized Synchronization for Multi-Threaded DSMs. In *Proc. of Workshop on Software Distributed Shared Memory*, May 2000.

[33] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[34] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Proc. of the Second ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'90)*, pages 168–177, 1990.

[35] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, February 1997.

[36] G. Fagg and J. Dongarra. FT-MPI:Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *EuroPVM/MPI 2000*, pages 346–353. Springer-Verilag, 2000.

[37] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, and Jack J. Dongarra. *ScaLAPACK User's Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[38] David Kranz, Kirk L. Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *In Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 54–63, May 1993.

[39] J. Carreira, L.M. Silva, J.G. Silva, and S. Chapple. Implementing Virtual Shared Memories on MPI. In *MPI Developers Conference*, University of Notre Dame, June 1995.

[40] J. Cordsen and W. Schroder-Preikschat. On the Coexistence of Shared-Memory and Message Passing in the Programming of Parallel Applications. In *Proc. of the High Performance Computer Networks Europe'97*, pages 718–727, 1997.

[41] Johan De Gelas. Server Guide part 2: Affordable and Manageable Storage. `http://www.anandtech.com/IT/showdoc.aspx?i=2859`, October 2006. Section on Disk performance.

[42] Jelica Protic and Veljko Milutinovic. Entry Consistency versus Lazy Release Consistency in DSM Systems: Analytical Comparison and a New Hybrid Solution. In *IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 78–83, Oct 1997.

[43] Andrew Grimshaw, Wm. A. Wulf, and the Legion team. The Legion Vision of a Worldwide Virtual Computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[44] Jim Almond and Dave Snelling. Unicore: uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems*, 15(5-6):539–548, 1999.

[45] Enabling Grids for E-sciencE (EGEE). `http://www.eu-egee.org/`.

[46] M.Matsuda, T.Kudoh, Y.Kodama, R.Takano, and Y.Ishikawa. Efficient MPI Collective Operations for Clusters in Long-and-Fast Networks. In *Cluster 2006*, 2006.

[47] Distributed ASCI Supercomputer: DAS-3, The next generation Grid infrastructure in the Netherlands. `http://www.starplane.org/das3`.

[48] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proc. of 5th European PVM/MPI Users' Group Meeting*, pages 180–187, 1998.

[49] N. Karonis, B. Toohen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[50] Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa. TCP Adaptation for MPI on Long-and-Fat Networks. In *Cluster 2005*, 2005.

[51] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide-area Systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[52] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *Proc. of the 14th Int'l Conference on Parallel and Distributed Processing Symposium (IPDPS-00)*, pages 377–386, 2000.

[53] Large Hadron Collider (LHC) at CERN. `http://lhc.web.cern.ch/lhc/`.

[54] Andreas Rodman and Mats Brorsson. Programming Effort vs. Performance with a Hybrid Programming Model for Distributed Memory Parallel Architectures. In *Euro-Par '99: Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 888–898, London, UK, 1999. Springer-Verlag.

[55] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs. In Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 61–73, San Jose, California, 1994.

[56] Miguel Castro, Manuel Sequeira, Manuel Costa, and Paulo Guedes. Efficient and Flexible Object Sharing. In *ICPP, Vol. 1*, pages 128–137, 1996.

[57] Rudolf Eigenmann, Jay Hoeflinger, Robert H. Kuhn, David Padua, Ayon Basumallik, Seung-Jai Min, and Jiajing Zhuand. Is OpenMP for Grids ? In *International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops*, 2002.

[58] Alfons G. Hoekstra and Peter M.A. Sloot. Introducing Grid Speedup Γ : A Scalability Metric for Parallel Applications on the Grid. In *Lecture Notes in Computer Science*, volume 3470, pages 245–254, Jun 2005.

[59] National Research Grid Initiative (NAREGI). The NAREGI Phase 1 Testbed is a heterogeneous environment composed of the computing resources of universities and research institutions linked together by the Super SINET network. `http://www.naregi.org/index_e.html`.

[60] J.K. Bennett, J.B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 125–135, 1990.

[61] H. S. Sandhu, T. Brecht, and D. Moscoso. Multiple Writers Entry Consistency. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume I, pages 355–362, 1998.

[62] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feb 1996.

[63] M. Stumm and S. Zhou. Algorithms Implementing Distributed Shared Memory. In *IEEE Computer, vol. 23, no. 5*, pages 54–64, 1990.

[64] M. Stumm and S. Zhou. Fault Tolerant Distributed Shared Memory Algorithms. In *In Proc. of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 719–724, December 1990.

[65] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *ACM Trans. on Computer Systems, 7(4)*, pages 321–359, November 1989.

[66] Jackie Silcock. Distributed Shared Memory: A Survey. Technical Report TR C95/22, Deakin University, Geelong, Victoria, Australia, June 1995.

[67] Kenneth P. Birman. Building Secure and Reliable Network Applications. In *WWCA*, pages 15–28, 1997.

[68] John P. Ryan and Brian A. Coghlan. Grid Timestamps, The Leapsecond Problem. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, June 2002.

[69] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*, chapter 10. Morgan Kaufmann, 3rd edition, 2001.

[70] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[71] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[72] A. Judge, P.A. Nixon, V.J. Cahill, B. Tangney, and S. Weber. Overview of Distributed Shared Memory. Technical Report TCD-CS-1998-24, Department of Computer Science, Trinity College, Dublin, Ireland, October 1998.

[73] John Dilley, Martin Arlitt, Stephane Perret, and Tai Jin. The Distributed Object Consistency Protocol. Version 1.0. Technical Report HPL-1999-109, Hewlett-Packard (HP), Sep 1999.

[74] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice-Hall International (UK), 1995. ISBN-13: 978-0132199087.

[75] David Mosberger. Memory Consistency Models. *SIGOPS Operating Systems Review*, 27(1):18–26, 1993.

[76] Michel Dubois and Christoph Scheurich and Faye A. Briggs. Memory Access Buffering in Multiprocessors. In *(ISCA 1986)*, pages 434–442, 1986.

[77] K. Gharachorloo, D. Lenoski, J. Laudon, P.B. Gibbons, A. Gupta, and J.L. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *25 Years ISCA: Retrospectives and Reprints*, pages 376–387, 1998.

[78] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pages 13–21, 1992.

[79] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, 1995.

[80] D. E. Lenoski, J. Ludon, K. Gharachorloo W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[81] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA (USA), 1991.

[82] Bill Nitzberg and Virginia Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *IEEE Computer*, 24(8):52–60, 1991.

[83] Mark Swanson, Leigh Stoller, and John Carter. Making Distributed Shared Memory Simple, Yet Efficient. In *Proc. on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, pages 2–13, 1998.

[84] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Memory Virtual Memory Systems. In *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pages 75–88, 1996.

[85] Pete Keleher. Update Protocols and Cluster-based Shared Memory. *Computer Communications*, 22:1045–1055, July 1999.

[86] Steven Frank. KRS1: High Performance and Ease of Programming, No Longer an Oxymoron. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '93)*, page 335, New York, NY, USA, 1993. ACM Press.

[87] Y. Zhou, L. Iftode, K. Li, J. P. Singh, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97)*, pages 193–205, 1997.

[88] L. Iftode and J. P. Singh. Shared Virtual Memory: Progress and Challenges. *Proc. of the IEEE, Special Issue on Distributed Shared Memory*, 87(3):498–507, March 1999.

[89] Hee-Chul Yun, Sang-Kwon Lee, Joonwon Lee, and Seungryoul Maeng. An Efficient Lock Protocol for Home-Based Lazy Release Consistency. In *Proceedings of the*

1st International Symposium on Cluster Computing and the Grid (CCGRID '01), page 527, Washington, DC, USA, 2001. IEEE Computer Society.

[90] V. Milutinovic and P. Stenstrom. Scanning the Issue, Special Issue on Distributed Shared Memory Systems. In *Proc. of the IEEE*, volume 87, pages 399–404, March 1999.

[91] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symp. on Operating Systems Principles (SOSP-13)*, pages 152–164, 1991.

[92] J. B. Carter, D. Khandekar, and L. Kamb. Distributed Shared Memory: Where We Are and Where We Should Be Headed? In *Fifth Workshop on Hot Topics in Operating Systems*, pages 119–122, 1995.

[93] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The Midway Distributed Shared Memory System. In *Procs. of COMPCON. The 38th Annual IEEE Computer Society International Computer Conference*, pages 528–537, Feb 1993.

[94] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. *ACM Operating Systems Review, SIGOPS*, 29(5):213–226, 1995.

[95] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, 1993.

[96] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[97] C. Amza, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM Protocols that Adapt between Single Writer and Multiple Writer. In *Proc. of the 3rd IEEE Symp. on High-Performance Computer Architecture (HPCA-3)*, pages 261–271, Feburary 1997.

[98] Christina Amza, A.L. Cox, Sandhya Dwarkadas, Li-Jie Jin, Karthick Rajamani, and Willy Zwaenepoel. Adaptive Protocols for Software Distributed Shared Memory. *Journal of the IEEE, Special Issue on Distributed Shared Memory*, pages 467–475, Mar 1999.

[99] Intel C/C++ Compiler Version 9.0 Reference. `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/277618.htm`.

[100] Pete Keleher. CVM documentation. Version 0.9, Aug 1998.

[101] Kritchalach Thitikamol and Pete Keleher. Thread Migration and Communication Minimization in DSM Systems. *Proceedings of the IEEE, Special Issue on Distributed Shared Memory Systems*, 87(3):487–497, March 1999.

[102] W. E. Speight and J. K. Bennett. Brazos: A Third Generation DSM System. In *Proc. of the USENIX Windows NT Workshop*, 1997.

[103] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures*, pages 277–287, 1996.

[104] E. Speight, H. Abdel-Shafi, and J.K. Bennett. An Integrated Shared-Memory/Message Passing API for Cluster-Based Multicomputing. In *Procs. of the Second International Conference on Parallel and Distributed Computing and Networks (PDCN)*, Brisbane, Australia, 1998.

[105] K. Gharachorloo, A. Gupta, and J. L. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. of the 1991 Int'l Conf. on Parallel Processing (ICPP'91)*, volume I, pages 355–364, August 1991.

[106] R.N. Zucker. *Relaxed Consistency and Synchronization in Parallel Processors.* PhD thesis, University of Washington, Houston, Texas, December 1992.

[107] Distributed European Infrastructure for Supercomputing Applications (DEISA). `http://www.fz-juelich.de/zam/grid/DEISA`. A Collaboration with US counterpart TeraGrid.

[108] The Globus Toolkit. `http://www.globus.org/toolkit`.

[109] W. Gropp and E. Lusk. Fault Tolerance in MPI Programs. In *Proc. of the Cluster Computing and Grid Systems Conference*, 2002.

[110] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for volatile Nodes. In *In Proceedings of SC 2002. IEEE*, 2002.

[111] C. Huang, O. Lawlor, and L. Kale. Adaptive MPI. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC), LNCS 2958*, pages 306–322, October 2003.

[112] William Gropp and Ewing Lusk. Goals Guiding Design: PVM and MPI. In *Proc. of the IEEE International Conference on Cluster Computing (CLUSTER '02)*, page 257, 2002.

[113] V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency, Practice and Experience*, 2(4):315–340, 1990.

[114] Ohio Supercomputer Center, Ohio State University. MPI Primer/Developing with LAM (Tutorial). `http://www.lam-mpi.org/`.

[115] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, Message Passing Interface Forum, 1994.

[116] Mitsuhisa Sato, Taisuke Boku, and Daisuke Takahashi. OmniRPC: A Grid RPC ystem for Parallel Programming in Cluster and Grid Environment. *3rd International Symposium on Cluster Computing and the Grid (CCGRID)*, 00:206, 2003.

[117] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proc. 10th IEEE Symp. On High Performance Distributed Computing, 2001.*, 2001.

[118] Tiziana Ferrari and Francesco Giacomini. Network monitoring for GRID performance optimization. *Computer Communications*, 27(14):1357–1363, 2004.

[119] Ruth Aydt, Dan Gunter, Warren Smith, Martin Swany, Valerie Taylor, Brian Tierney, and Rich Wolski. A Grid Monitoring Architecture. Technical Report gwd-perf-16-1, GGF, Jul 2001.

[120] The GLUE Schema Specification (Version 1.3). `http://glueschema.forge.cnaf.infn.it`.

[121] S. Fisher et al. R-GMA: A Relational Grid Information and Monitoring System. Technical Report WP3-2003-01-14, 2nd Cracow Grid Workshop, DATAGRID, Jan 2003.

[122] N. Podhorszki and P. Kacsuk. Monitoring Message Passing Applications in the Grid with GRM and R-GMA. *Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI)*, pages 603–610, 2003.

[123] Jaroslaw Nieplocha, Robert J. Harrison, and Richard J. Littlefield. Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance Computers. *The Journal of Supercomputing*, 10(2):169–189, 1996.

[124] W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W.-M. Hwu, R. Hank, and J. Gyllenhaal. Using Profile Information to Assist Advanced Compiler Optimization and Scheduling. *Lecture Notes in Computer Science*, 757:31–49, 1993.

[125] Francis H. Dang and Lawrence Rauchwerger. Speculative Parallelization of Partially Parallel Loops. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 285–299, 2000.

[126] D. J. Hancock, J. Mark Bull, Rupert W. Ford, and T. L. Freeman. An Investigation of Feedback Guided Dynamic Scheduling of Nested Loops. In *ICPP Workshop*, page 315, 2000.

[127] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[128] OpenMP Version 2.5 Specification, May 2005. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`.

[129] Jay P. Hoeflinger. Extending OpenMP* to Clusters. Technical Report 312568-001, Intel, 2006.

[130] Cluster OpenMP, User's Guide. `http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/285865.htm`. Version 9.1.

[131] Dan Bonachea and Jason Duell. Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, 2003.

[132] Kritchalach Thitikamol and Peter J. Keleher. Multi-threading and Remote Latency in Software DSMs. In *Proc. of 14th International Conference on Distributed Computing Systems*, pages 296–304, 1997.

[133] Sven Karlsson and Mats Brorsson. Priority Based Messaging for Software Distributed Shared Memory. *Cluster Computing*, 6(2):161–169, 2003.

[134] Nuno Neves, Miguel Castro, and Paulo Guedes. A Checkpoint Protocol for an Entry Consistent Shared Memory System. In *Symposium on Principles of Distributed Computing*, pages 121–129, 1994.

[135] Jai-Hoon Kim and Nitin H. Vaidya. Recoverable Distributed Shared Memory Using the Competitive Update Protocol. In *Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 152–157, December 1995.

[136] Brett D. Fleisch, Heiko Michel, Sachin K. Shah, and Oliver E. Theel. Fault Tolerance and Configurability in DSM Coherence Protocols. *IEEE Concurrency*, 8(2):2–13, /2000.

[137] S. Adve, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. A Comparison of Entry Consistency and Lazy Release Consistency Implementations. In *Proceedings of the Second High Performance Computer Architecture Conference*, pages 26–37, Feb 1996.

[138] John P Ryan and Brian A Coghlan. Distributed Shared Memory in a Grid Environment. In *In Proc. of International Conference on Parallel Computing (ParCo 2005)*, volume 33, 2005.

[139] Maria Clicia Stelling de Castro and Claudio L. Amorim. Efficient Categorization of Memory Sharing Patterns in Software DSM systems. In *International Parallel and Distributed Processing Symposium*, San Francisco, California, USA, April 2001. ACM & IEEE.

[140] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. In *Proc. of the 1st Symp. on Operating Systems Design and Implementation (OSDI'94)*, pages 87–100, 1994.

[141] J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. PhD thesis, Rice University, Houston, Texas, 1993.

[142] Takeshi Yamazaki, Naoki Yonezawa, Pusit Kulkasem, Shinichi Yamagiwa, Masaaki Ono, Ayman. N. M. Al-Khoury, and Koichi Wada. SVCP: A Cache Coherency Protocol with Explicit Update Subscription. In *Proceedings of the 1998 International Conference on Parallel and Distributed Processing Techniques and Application (PDPTA98)*, pages 899–906, July 1998.

[143] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proc. of the 22nd Annual Int'l Symp. on Computer Architecture (ISCA'95)*, pages 48–59, 1995.

[144] S. Zhou, M. Stumm, and T. McInerney. Extending Distributed Shared Memory to Heterogeneous Environments. In *Proc. 10th International Conf. Distributed Computing Systems*, Los Alamitos, Calif, May–June 1990. CS Press.

[145] Zoran Radović and Erik Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of Supercomputing 2002*, Baltimore, Maryland, USA, November 2002.

[146] Zoran Radović and Erik Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, pages 241–252, Anaheim, California, USA, February 2003.

[147] Nian-Feng Tzeng and Angkul Kongmunvattana. Distributed Shared Memory Systems with Improved Barrier Synchronization and Data Transfer. In *Proc. of International Conference on Supercomputing 1997*, pages 148–155, 1997.

[148] Xuehai Zhang, Jeffrey L. Freschl, and Jennifer M. Schopf. A Performance Study of Monitoring and Information Services for Distributed Systems. In *12th International Symposium on High-Performance Distributed Computing (HPDC 2003)*, pages 270–282. IEEE Computer Society, June 2003.

[149] Pedro C. Diniz and Martin C. Rinard. Dynamic feedback: an effective technique for adaptive computing. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 71–84, New York, NY, USA, 1997. ACM Press.

[150] Rajeev Thakur and William Gropp. Improving the Performance of Collective Operations in MPICH. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS2840 in Lecture Notes in Computer Science, pages 257–267. Springer Verlag, 2003. 10th European PVM/MPI User's Group Meeting.

[151] IBM POWER Architecture. `http://www-03.ibm.com/chips/power`.

[152] Oscar Hernandez, Fengguang Song, Barbara Chapman, Jack Dongarra, Bernd Mohr, Shirley Moore, and Felix Wolf. Performance Instrumentation and Compiler Optimizations for MPI/OpenMP Applications, 2006.

[153] Rainer Keller, Bettina Krammer, Matthias S. Mueller, Michael M. Resch, and Edgar Gabriel. Towards Efficient Execution of MPI Applications on the Grid: Porting and Optimization Issues. *Journal of Grid Computing*, 1(2):133–149, 2003.

[154] Bernd Mohr and Felix Wolf. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs. In *In Proceedings of Euro-Par 2003*, pages 1301–1304, 2003.

[155] Sameer S. Shende and Allen D. Malony. The Tau Parallel Performance System. *Int. Journal High Performance Computing Applications*, 20(2):287–311, 2006.

[156] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer-Ann M. Anderson, Steven W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.

[157] Darko Marinov, Davor Magdic, Aleksandar Milenkovic, Jelica Protic, Igor Tartalja, and Veljko Milutinovic. An Approach to Characterization of Parallel Applications for DSM Systems. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7*, page 782, Washington, DC, USA, 1998. IEEE Computer Society.

[158] Mehmet F. Su, Ihab El-Kady, David A. Bader, and Shawn-Yu Lin. A Novel FDTD Application Featuring OpenMP-MPI Hybrid Parallelization. In *Proc of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 373–379, 2004.

[159] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[160] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.

[161] Steven Cameron Woo, M. Ohara, E. Torrie, J.P. Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *In Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[162] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 37, New York, NY, USA, 1995. ACM Press.

[163] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster. In *WOMPAT2002.*, 2002.

[164] Michael Frumkin and Rob F. Van der Wijngaart. NAS Grid Benchmarks: A Tool for Grid Space Exploration. *hpdc*, 00:0315, 2001.

[165] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *Proceedings USENIX 1996 Winter Conference*, pages 279–294, January 1996.

[166] TOP500 Supercomputer Sites. `http://www.top500.org`.

[167] MPI over InfiniBand Project. `http://nowlab.cse.ohio-state.edu/projects/mpi-iba`.

[168] Tyng-Yeu Liang, Chun-Yi Wu, Jyh-Biau Chang, and Ce-Kuen Shieh. Teamster-G: A Grid-enabled Software DSM System. In *Proc. of CCGRID 2005*, pages 905–9122, 2005.

[169] T.S. Trevisan. Distributed Shared Memory in Kernel Mode. In *In Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD.02)*, 2002.

[170] OMNI OpenMP Compiler Project. `http://phase.hpcc.jp/Omni/home.html`.

[171] OMPi OpenMP C Compiler. `http://www.cs.uoi.gr/~ompi/index.html`.

[172] OdinMP - The Open source OpenMP Compiler for C, C++, and Fortran. `http://www.odinmp.com`.

[173] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning.* Lawrence Erlbaum Associates, Hillsdale, NJ, USA, 1989.

[174] L. Lorena, M. Narciso, and J. Beasley. A Constructive Genetic Algorithm for the Generalized Assignment Problem, 1999.

[175] Thomas Seidmann. Distributed Shared Memory Using The .NET Framework. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluser Computing and the Grid (CCGRID'03)*, pages 457–462, 2003.

[176] Gabriel Antoniu, Luc Boug, and Sbastien Lacour. Making a DSM Consistency Protocol Hierarchy-Aware: an Efficient Synchronization Scheme. In *Proc. Workshop on Distributed Shared Memory on Clusters (DSM'03)*, pages 516–523, May 2003.

[177] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on Network of Workstations. In *Proc. of Supercomputing'98*, 1998.

[178] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Michael L. Scott. Multi-level Shared State for Distributed Systems. In *Procs. of 31st Int. Conference on Parallel Processing (ICPP'02)*, August 2002.

[179] Mark Harris. GPGPU: Beyond Graphics. Technical report, NVIDIA, 2004. `http://developer.nvidia.com/object/gpgpu_beyond_graphics.html`.

[180] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The Potential of the CELL Processor for Scientific Computing. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[181] Intel MPI Benchmarks. `http://www.intel.com/software/products/cluster/mpi/mpi_benchmarks_lic.htm`.

[182] William Gropp and Ewing Lusk. User's Guide for MPICH: A Portable Implementation of MPI, 1996.

[183] I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of SC'98*. ACM Press, 1998.

[184] Webpage for MPICH MPI Distribution. `http://www-unix.mcs.anl.gov/mpi/mpich`.

[185] Mpiexec - MPI parallel job launcher for PBS. `http://www.osc.edu/~pw/mpiexec/index.php`.

[186] Altair PBS Professional. `http://www.altair.com/software/pbspro.html`.

[187] Torque (OpenPBS) Resource Manager. `http://www.openpbs.org/main.html`.

[188] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994. `http://www.lam-mpi.org/download/files/lam-papers.tar.gz`.

[189] The Berkeley Lab Checkpoint-Restart (BLCR) system. `http://ftg.lbl.gov/checkpoint`.

[190] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. OpenMPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004.

[191] Homepage for Parallel Virtual Machine (PVM). `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[192] Brian Tierney, William E. Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The NetLogger Methodology for High Performance Distributed Systems Performance Analysis. In *Proceeding of IEEE High Performance Distributed Computing conference (HPDC-7)*, pages 260–267, July 1998.

[193] Lawrence Livermore National Laboratory (LLNL) OpenMP Tutorial. `http://www.llnl.gov/computing/tutorials/openMP`.

Co-Authored Papers

Cooke, A.W., Gray, A.J.G., Nutt, W., Magowan, J., Oevers, M., Taylor, P., Cordenonsi, R., Byrom, R., Cornwall, L., Djaoui, A., Field, L., Fisher, S.M., Hicks, S., Leake, J., Middleton, R., Wilson, A., Zhu, X., Podhorszki, N., Coghlan, B., Kenny, S., O'Callaghan, D., and Ryan, J. (2005) "The Relational Grid Monitoring Architecture: Mediating Information about the Grid" Grid Journal, 2005.

Byrom, R., Coghlan, B., Cooke, A., Cordenonsi, R., Cornwall, L., Datta, A., Djaoui, A., Field, L., Fisher, S., Hicks, S., Kenny, S., Magowan, J., Nutt, W., O'Callaghan, D., Oever, M., Podhorszki, N., Ryan, J., Soni, M., Taylor, P., Wilson, A.., and Zhu, X. (2005) "The Canonical Producer: an instrument monitoring component of the Relational Grid Monitoring Architecture (R-GMA)" Scientific Programming, Special Issue, ISSN 1058-9244, Vol.13, No.2, pp151-158, 2005.

Maad, S., Coghlan, B., Quigley, G., Ryan, J., Kenny, E., O.Callaghan, D. (2006) "Towards a Complete Grid Filesystem Functionality" Future Generation Computer Systems, 2006.

Maads, S., Coghlan, B., Quigley, G., Ryan, J., Kenny, E., Pierantoni, G. (2006) "A Grid Filesystem: A Key Component For Managing the Data Centre in the Enterprise" Journal Scalable Computing: Practice and Experience (SCPE), special issue on Grid Computing For Enterprise, 2006.

Ryan, J.P., and Coghlan, B.A. (2002) "Grid Timestamps, the Leapsecond Problem" Proc.PDPTA'2002, Las Vegas, June, 2002.

Coghlan, B., Cooke, A.W., Datta, A., Djaoui, A., Field, L., Fisher, S., Magowan, J., Nutt, W., Oevers, M., Soni, M., Podhorszki, N., Ryan, J., Wilson, A.J., and Zhu, X. (2002) "R-GMA: A Grid Information and Monitoring System" Allhands Meeting, Sheffield, September, 2002.

Cooke, A., Nutt, W., Magowan, J., Taylor, P., Leake, J., Byrom, R., Field, L., Hicks, S., Soni, M., Wilson, A., Cordenonsi, R., Cornwall, L., Djaoui, A., Fisher, S., Podhorszki, N., Coghlan, B., Kenny, S., O.Callaghan, D., and Ryan, J. (2003) "Relational Grid Monitoring Architecture (R-GMA)" Allhands Meeting, Sheffield, September, 2003.

Ryan, J.P. and Coghlan, B.A. (2004) "SMG: Shared Memory for Grids" PDCS'04, Boston, November, 2004

Ryan, J.P. and Coghlan, B.A. (2005) "Distributed Shared Memory in a Grid Environment" Proc.International Conference on Parallel Computing (PARCO 2005), Malaga, Spain, September, 2005.

Maad, S., Coghlan, B., Ryan, J., Kenny, E., Watson, R., and Pierantoni, G. (2005) "The Horizon of the Grid For E-Government" Proc.e-Government Workshop (eGOV05), Brunel, UK, September, 2005, (ISBN 1-902316-46-0), 2005.

Maad, S., Coghlan, B., Pierantoni, G., Kenny, E., Ryan, J., and Watson, R. (2005) "Adapting the Development Model of the Grid Anatomy to meet the needs of various Application Domains" Proc. Cracow Grid Workshop (CGW05), Crakow, Poland, November, 2005.

Coghlan, B., Quigley, G., Maad, S., Ryan, J., Kenny, E., O.Callaghan, D. (2006) "A Transparent Grid Filesystem" Proc.Workshop on State-of-the-Art in Scientific and Parallel Computing (PARA06), Umea, Sweden, June 18-21, 2006.

Maad, S., Coghlan, B., Pierantoni, G., Kenny, E., Ryan, J. (2006) "New frontiers for grid applications" Proc.European and Mediterranean Conference on Information Systems 2006 (EMCIS06), Alicante, Spain, July 6-7, 2006.

Maad, S., Coghlan, B., Quigley, G., Ryan, J., and Kenny, E. (2006) "Universal Accessibility to the Grid via Metagrid Infrastructure" EGEE Forum, CERN, Switzerland, March, 2006.

Ryan, J.P., Coghlan, B.A. (2002) "Timestamps" European DataGrid WP3 Workshop, Rutherford Appleton Labs, 16th January, 2002.

Ryan, J.P., Coghlan, B.A. (2002) "Timestamps" European DataGrid Conference, Paris, 4th March, 2002.

Ryan, J.P., and Coghlan, B.A. (2002) "EDG Timestamping" European DataGrid Conference, Budapest, September, 2002.

INDEX

Críoch