Integrating Software Distributed Shared Memory and Message Passing Programming

H'sien Jin Wong

April 2010

A thesis submitted for the degree of Doctor of Philosophy
of the Australian National University

Declaration

The fact is this: I am my own man, and I am not a slave.

*For my loving parents...
...and my younger brother who wants a "doctor brother!"*

Declaration

The work in this thesis is my own except where otherwise stated.

H'sien Jin Wong

Acknowledgements

Supervisors

Alistair for keeping my spirits up throughout the many years; Peter for keeping my spirits topped up throughout “beer-o-clock;” and Sato for being a gracious host in Japan.

Readers

Alistair, Warren, Josh, Pete, Jie, Joseph, and Ganesh.

Housemates

Dogol and Brandall for sharing your home.

Last but definitely not least

For being a true friend, thank you very much Roy.

Abstract

Software Distributed Shared Memory (SDSM) systems provide programmers with a shared memory programming environment across distributed memory architectures. In contrast to the message passing programming environment, an SDSM can resolve data dependencies within the application without the programmer having to explicitly specify the required communications. Such ease-of-use is, however, provided at a cost to performance. This thesis considers how the SDSM programming model can be combined with the message passing programming model with the goal of avoiding these additional costs when a message passing solution is straightforward.

To pursue the above goal a new SDSM library, named Danui, has been developed. The SDSM manages a shared address space in units of pages. Consistency for these pages is maintained using a variety of home-based protocols. In total, the SDSM includes seven different barrier implementations, five of which support home-migration.

Danui is designed with portability and modularity in mind. It is written using the standard Message Passing Interface (MPI) to perform all underlying SDSM related communications. MPI was used both to permit the subsequent exploration of user-level MPI/SDSM programming, and also to enable the SDSM library to exploit optimised MPI implementations on a variety of cluster interconnects.

A detailed analysis of the costs associated with the various SDSM operations is given. This analysis characterizes each SDSM operation based upon a number of factors. For example, the size of an SDSM page, the number of modifications made to shared memory, the choice of barrier used etc. This information is then used as a guide to determine which parts of an SDSM program would benefit most if replaced by direct calls to MPI.

The integration of the shared memory and message passing (SMMP) programming models within Danui is discussed in detail. It is shown that a naïve integration of the SDSM and MPI programming models leads to memory consistency problems when MPI transfers occur to or from memory that is part of the software managed shared address space. These problems can, however, be overcome by noting the semantics of the shared memory and message passing paradigms, and introducing a few extensions to the MPI interface.

The performance of the Danui SMMP environment is evaluated for a variety of benchmarks. Results show that the SMMP implementation lowers the latency of data transfer in SDSM programs, and in some cases even matches the latency of MPI. The possibility of adapting compilers, that currently map high-level shared memory models such as OpenMP onto SDSM systems, to use both SDSM and message passing constructs is considered. The applicability of this work to other SDSM models, such as the home-less model, is also discussed.

Contents

Acknowledgements	vii
Abstract	ix
1 Introduction	1
1.1 Contributions	2
1.1.1 The Danui Software Distributed Shared Memory Systems	3
1.1.2 Performance Modeling and Evaluation	4
1.1.3 Extensions for Supporting Integrated Shared Memory and Mes- sage Passing Programming	5
1.2 Thesis Structure	6
2 Background	7
2.1 Memory Consistency	7
2.1.1 Program Order	8
2.1.2 Sequential Consistency	9
2.1.3 Relaxed Memory Consistency	11
2.1.4 Other Consistency Models	15
2.2 Software Distributed Shared Memory	15
2.2.1 IVY	16
2.2.2 Munin	18
2.2.3 TreadMarks	19
2.2.4 Home-based Lazy Release Consistency SDSMs	21
2.3 Message Passing Interface	23
2.3.1 MPI Communicators	24
2.3.2 MPI Point-to-point Communication	24
2.3.3 Collective Communication	26
2.3.4 Remote Memory Access	28
2.3.5 MPI and Threading	31
2.4 Related Work	31

2.4.1	MIT Alewife Machine	31
2.4.2	Stanford FLASH	32
2.4.3	TreadMarks Extension	32
2.4.4	Brazos	32
2.5	Alternative Programming Models	32
2.5.1	Linda	33
2.5.2	Global Arrays	34
2.5.3	Partitioned Global Address Space Languages	35
2.6	Summary	35
3	Design of the Danui SDSM Library	37
3.1	The Event Module	38
3.1.1	Defining Event Handlers	39
3.1.2	Associating Event IDs to Handlers	39
3.1.3	Sending Events	40
3.1.4	Event Daemon Thread	41
3.2	The Lock Module	43
3.2.1	The Master Queue	44
3.2.2	The Distributed Queue	45
3.2.3	Nested Lock Usage	55
3.3	The Heap Module	55
3.3.1	The Home-based MCM Class	58
3.3.2	The Invalidate All MCM Class	62
3.3.3	The Selective Invalidation MCM Class	63
3.3.4	The Write-set MCM Class	67
3.3.5	The Scalable MCM Class	69
3.4	Danui Application Programming Interface	72
3.4.1	Module Selection	73
3.4.2	Example Code: Finite-difference Approximation of the 2D Laplace Equation	74
3.5	Summary	76
4	Performance Characterization of the Danui SDSM	79
4.1	Latency and Bandwidth of MPI on the Evaluation System	80
4.1.1	MPI Latency Benchmark	81
4.1.2	MPI Bandwidth Benchmark	82
4.1.3	Measured Latency and Bandwidth	82
4.2	Event Module Performance Model	83
4.3	Lock Module Performance Model	85

4.3.1	Lock Event Count Analysis	86
4.3.2	Lock Cost Measurement	89
4.4	Heap Module Performance Model	92
4.4.1	Fault Cost Model	92
4.4.2	Flush Cost Model	93
4.4.3	Barrier Cost Model	97
4.5	Comparison of Barrier Costs	108
4.5.1	Barrier Cost versus Number of Processes	108
4.5.2	Barrier Cost versus Number of Shared Pages	109
4.5.3	Barrier Cost versus Number of Diffs	110
4.6	MCBENCH: A Memory Consistency Benchmark	111
4.6.1	MCBENCH Results: 4-byte Chunk Size	112
4.6.2	MCBENCH Results: 2048-byte Chunk Size	114
4.6.3	MCBENCH Results: 4096-byte Chunk Size	115
4.6.4	Conclusions for MCBENCH	116
4.7	Critical Path Performance Model	116
4.7.1	Performance Model Accuracy: MCBENCH Case	119
4.7.2	Performance Model Accuracy: 2D Fast Fourier Transform Case .	120
4.7.3	Conclusions for Critical Path Performance Model	124
4.8	Summary	125
5	Hybrid Programming Model	127
5.1	Analysis for SMMP Programming	128
5.1.1	Memory Access Side-effects	129
5.1.2	Classification of Point-to-point Communication	131
5.1.3	Issues Arising from Symmetric Shared-to-shared Communication	132
5.1.4	Leveraging the Semantics of Message Passing Send and SDSM Operations	135
5.2	The Danui SMMP Extension	136
5.2.1	Infrastructure for Integrating MPI Calls	137
5.2.2	Taxonomy of Bookkeeping Tasks	139
5.2.3	One-sided Home-migration	143
5.3	Evaluation of the Danui SMMP Extension	146
5.3.1	Bulk Data Transfer SMMP Benchmark	146
5.3.2	Parallel Dense Blocked LU Factorization	152
5.4	Summary	160

6	Conclusions	163
6.1	Future Work for the Danui Software Distributed Shared Memory Library	164
6.1.1	Page Pre-fetching	164
6.1.2	Multiple Computation Threads per SDSM Process	164
6.1.3	Critical Path Performance Model and Locks	165
6.1.4	Use of MPI_THREAD_MULTIPLE	165
6.1.5	Cluster of Sub-clusters	165
6.2	Future Work for the Danui SMMP Implementation	165
6.2.1	Migrate Page to Receiver	166
6.2.2	MPI-2 Remote Memory Access	166
6.2.3	Collective Operations	166
6.2.4	Homeless Memory Consistency Manager and SMMP Extension .	167
6.2.5	SMMP Application Examples	167
	Bibliography	169

Chapter 1

Introduction

Software Distributed Shared Memory (SDSM) systems [47, 13, 33, 43, 63, 48, 65, 35, 39] enable the use of shared memory programming models on distributed memory architectures. Shared memory programming provides some conveniences over message passing programming. In message passing programming [51, 29], data dependencies between processes are resolved by transferring data in messages; thereby giving programmers explicit control over inter-process communication. While this control can lead to good scalability and performance, having to explicitly specify all inter-process communication can be tedious and potentially error prone. In shared memory programming, attention is shifted from the specifying of data dependencies to the ordering of the shared memory accesses; leaving the shared memory system to determine potential data dependencies. It is widely accepted that the shared memory programming model simplifies the development of applications that exhibit irregular and/or dynamic behaviour [36].

In an SDSM system identifying possible data dependencies, and then generating the inter-process communications required to resolve them, can represent a considerable cost overhead. Also, the generated inter-process communications may be suboptimal, for example, servicing memory accesses to a large region of shared memory may result in the SDSM issuing multiple requests for small pieces of memory as opposed to transferring the whole region at one time [44]. Given these overheads, it is reasonable to argue that message passing should be used when an application exhibits regular behaviour, while an SDSM should be used when an application exhibits irregular behaviour.

In some cases, however, it is not clear cut whether an application should use message passing or an SDSM. Zhu *et al.* [79, 78] observed that access patterns to arrays in an application may differ such that message passing is suited for some, while an SDSM would be beneficial for the rest. Zhu *et al.* also note in [79] that access patterns for a given array may be regular in some parts of an application but be irregular in other parts of the same application. In these circumstances it would be beneficial to

provide the programmer with an environment that supports the combined use of both the message passing and the SDSM paradigms.

Kranz *et al.* [44] list the following cases where the use of message passing would be beneficial:

- “*Coarse granularity*”: When data dependencies between processes involve a large contiguous region of memory, the communication cost for transferring this data must be paid regardless of programming model. Relying on shared memory coherency mechanisms to transfer data in piecemeal chunks is less efficient than a single bulk data transfer via message passing.¹
- “*Known communication patterns*”: When communications between processes are regular it is more efficient to transfer data directly between the processes as relying on shared memory coherency mechanisms may involve intermediaries.
- “*Combining synchronization with data transfer*”: Synchronization in shared memory programming models is used to order memory accesses between processes that create data dependencies. Synchronization and data transfers resulting from resolution of data dependencies tend to be separate from one another in shared memory programming. In contrast, synchronization and data transfer are combined by using message passing (it is impossible to receive data that has not been sent).

Of the above, combining synchronization with data transfer is arguably the most significant difference between the shared memory and the message passing programming paradigms; in the sense that message passing operations are explicit expressions of inter-process data dependencies.

The goal of the work outlined in this thesis was to develop an SDSM system that, for the sort of scenarios given by Kranz *et al.* [44], was able to exploit the direct use of message passing constructs. In other words, the objective was to develop an SDSM system that supports hybrid shared memory and message passing programming.

Below we outline the major contributions presented in this thesis, before providing a brief overview of the structure of the thesis itself.

1.1 Contributions

This thesis describes a new SDSM library, called Danui, that is designed with portability and flexibility in mind. It presents detailed performance models for the core

¹Kranz *et al.* [44] note an exception (to the “coarse granularity” case above) where a region consists of independent data elements that can be processed immediately upon receiving without having to receive all other data elements. In these cases, a shared memory system with pre-fetch capability would be able to overlap data transfer with processing of received data; therefore, potentially achieving better performance.

operations in Danui, that are then verified experimentally. It also outlines extensions to Danui that enable Danui to support an integrated shared memory and message passing programming environment. The specific contributions in each of these three areas are summarized below.

1.1.1 The Danui Software Distributed Shared Memory Systems

A new SDSM system, called Danui, was developed from scratch as part of the work reported in this thesis. To permit portability between platforms, and mindful of the overall objective to develop integrated SDSM and message passing programming models, Danui uses the standard Message Passing Interface (MPI) [51] to perform all inter-process communications. In this respect Danui is similar to the SDSM system developed by Ojima *et al.* [58]. It differs, however, from the work of Ojima *et al.* in that it assumes the use of a thread-safe version of MPI, and this has allowed much greater use of MPI from within the SDSM.

Danui is also highly modular, and this has allowed for a variety of optimization strategies to be tested. It has the following features:

- An Event module that implements a lightweight Remote Procedure Call (RPC) facility with minimal latency overheads.
- A Lock module with two alternative implementations for managing processes waiting for a lock: Master Queue (central management), and Distributed Queue (distributed management).
- A Heap module with seven alternative Memory Consistency Manager (MCM) implementations: 1) Invalidate All (basic implementation), 2) Selective Invalidation, 3) Write-set Single-writer (home-migration), 4) Write-set Lowest ID (multiple-writers home-migration), 5) Write-set N-Homes (multiple-writers home-migration), 6) Scalable Single-writer, and 7) Scalable Multiple-writers.
- The ability to be initialized with any given MPI communicator object. This allows multiple SDSM instances to be initialized for different groups of MPI processes on a large cluster.

Partial details of the Danui system have been published in the following articles:

- H'sien Jin Wong and Alistair P. Rendell, "The Design of MPI based Distributed Shared Memory Systems to Support OpenMP on Clusters." In *IEEE Cluster*, pp.231–240, 2007.
- H'sien Jin Wong and Alistair P. Rendell, "Integrating Software Distributed Shared Memory and Message Passing Programming." In *IEEE Cluster*, 2009.

1.1.2 Performance Modeling and Evaluation

Cost models and benchmarks are presented for key SDSM operations including lock synchronization and shared memory coherency and consistency operations. The following contributions were made in this area:

- Lock usage is characterized by the degree of contention on a lock, which can be either low or high. This characterization is used to develop cost models for each lock operation under a specific contention level. In particular, models for lock acquisition during high contention usage differentiate between time spent waiting for a lock to be available and the time spent waiting to complete the operation when the lock becomes available. Benchmarks were designed to measure each lock operation under these contention levels. These benchmarks provide more detail than existing lock benchmarks, such as in the EPCC micro-benchmark suite [18, 19] for OpenMP.
- A benchmark called MCBENCH was developed for measuring memory consistency overheads. The benchmark can be used to compare the performance of different shared memory systems (whether they are implemented in hardware or software). In this thesis, MCBENCH is used to compare the performance of the seven Memory Consistency Managers (MCMs) provided by the Danui SDSM.
- A critical path [38, 62] model is developed that extends from the models described by Cai *et al.* [20]. The extension differs from that presented in Cai *et al.* in two ways: it does not require programs to be load balanced; and cost models for more coherency operations are included (Cai *et al.* included only page faults; the new cost models are also more detailed).
- A novel approach for applying the model called Modeled Time is presented. Modeled Time builds the extended critical path model mentioned above into the SDSM. A special timing call is provided to obtain wall-times that have been adjusted by the cost models. Thus, by comparing modeled time with actual times, it is possible to check the accuracy of the cost models.

Partial details of the benchmarks and performance models have been published in the following articles:

- H'sien Jin Wong, Jie Cai, Alistair P. Rendell, and Peter E. Strazdins, "Micro-benchmarks for Cluster OpenMP Implementations: Memory Consistency Costs." International Workshop on OpenMP (IWOMP), LNCS 60-70, 5004, 2008.
- Jie Cai, Alistair P. Rendell, Peter E. Strazdins, H'sien Jin Wong, "Performance Models for Cluster-Enabled OpenMP Implementations." In proceedings of 13th

IEEE Asia-Pacific Computer Systems Architecture Conference, pp.1–8, 4–6 Aug, 2008.

1.1.3 Extensions for Supporting Integrated Shared Memory and Message Passing Programming

Danui is extended to allow message passing operations to be used with shared memory regions in what is referred to as integrated shared memory and message passing (SMMP). Message passing is supported using MPI in a coherent fashion that allows MPI and SDSM operations to be used together seamlessly. This differs from the hybrid programming model described by Zhu *et al.* [79, 78] in that they seek to apply different programming models (shared memory or message passing) to different arrays in a program; but the integrated SMMP programming model seeks to support combined application of both programming models to arrays in a program.

Brazos [65] is an SDSM that provides an integrated SMMP programming environment [64] by supporting a “large subset” of MPI operations². Coherent message passing is supported by translating message passing semantics into shared memory coherency operations. In this work, an integrated SMMP implementation is detailed that improves upon the Brazos implementation by leveraging key observations about message passing and shared memory semantics to avoid a substantial number of shared memory coherency tasks. This lowers overheads for coherent message passing.

The contributions made in this area can be summarized as follows:

- An MPI-based SDSM was extended to support integrated use of MPI functions from within shared memory programs that are using the SDSM.
- Key observations were made that greatly reduced the need for coherency action when supporting coherent message passing involving shared memory.
- Latency benchmarks indicate that, in the best case, the latency of coherent message passing using the integrated SMMP implementation is able to match that of pure MPI.

Partial details of the extended Danui system have been published in the following article:

- H’sien Jin Wong and Alistair P. Rendell, “Integrating Software Distributed Shared Memory and Message Passing Programming.” In IEEE Cluster, 2009.

²Brazos is, however, not built on MPI.

1.2 Thesis Structure

Chapter 2 presents background material on memory consistency, SDSM implementation, MPI, and integrated SMMP implementations. Chapter 3 describes implementation of the Danui SDSM library. Chapter 4 presents cost models that characterize the performance of the Danui SDSM. Chapter 5 details extensions to the Danui SDSM for supporting an integrated SMMP programming environment. Lastly, chapter 6 presents conclusions and future work.

Chapter 2

Background

This thesis is about integrating Software Distributed Shared Memory (SDSM) and message passing programming. This chapter provides background information for the SDSM library implementation developed as part of this work and the widely used Message Passing Interface (MPI) [51, 52] standard on which it is based.

SDSM systems [47, 13, 33, 43, 63, 48, 65, 35, 39, 41] provide programmers with a virtual shared memory programming environment that hides the distributed nature of the underlying memory. This is achieved by converting shared memory accesses into inter-process communications. Memory consistency models are used to determine when such inter-process communications are required [4]. An overview of memory consistency models is given as a precursor to understanding how SDSM systems can be implemented. In particular, relaxed memory consistency models [25, 30] enable SDSM systems to reduce the amount of communication required [43]. This is especially important on computer clusters [66] where communication over the network interconnects is costly.

Section 2.1 provides an introduction to memory consistency models. Section 2.2 presents important milestones in SDSM implementation and discusses important SDSM issues and solutions. Section 2.3 gives details about the MPI programming model used in this work. Section 2.4 concerns related work. Section 2.5 gives a review of alternative programming models/environments. Finally, section 2.6 provides a summary of this chapter.

2.1 Memory Consistency

Memory consistency models [4, 25, 30, 14, 40] describe when the effect of a store operation has to be observable to load operations and what observing, or not observing, the effect of the store operation implies about other load and store operations. In short, a memory consistency model describes how a programmer can expect a memory system

to behave and is therefore important for reasoning about the correctness of a program.

This section begins with a description of *program order*, a consistency model for sequential programs. This is then built upon to describe *sequential consistency*, a consistency model for parallel programs. Following which, relaxed consistency models that help to reduce the amount of communication required between processes are described. Finally, other consistency models are mentioned.

2.1.1 Program Order

Program order [4] is a consistency model for sequential programs. It is satisfied when memory operations *appear* to happen in the order in which they occur in the actual program.¹ It is important to note that this does not restrict execution of program instructions to the order in which operations occur in the program. Rather, program order is satisfied so long as the results of an executed program are observed to be consistent with the result of executing operations in the order in which they occur in the program. The sequential program listed below illustrates this distinction:

1. $A \leftarrow 1$.
2. $B \leftarrow A$.
3. $C \leftarrow A$.
4. $A \leftarrow 2$.
5. Read values of A , B , and C .

An analysis of dependencies between the statements listed above reveals that statements 2 and 3 are dependent on statement 1 because of variable A . In turn, statement 4 cannot execute until both statements 2 and 3 have read A . Thus, swapping or concurrently executing statements 2 and 3 would result in a reordering of instructions that would still produce the same result as if the instructions were executed in program order. On the other hand, swapping statements 3 and 4 would cause variable C to be incorrectly initialized (to 2 instead of 1), thereby violating program order. Maintaining the appearance of program order is fundamentally important because it is what the programmer expected when they wrote the original code.

Program order has implications for the correct treatment of store operations in a memory system. Store queues [60] are mechanisms for offloading the cost of store operations to a separate system so that subsequent operations in a program may be executed without having to wait for the store operation to complete. If naïvely implemented, statement 2 in the previous example may load the value of A from memory while the

¹Lamport [46] describes this as “sequential.”

operation to store A to memory given in statement 1 is still in a store queue; thereby resulting in a violation of program order. Two approaches that can be used to address this coherency issue are: 1) wait for the store queue to be empty before issuing load operations, or 2) inspect the store queue for stores to the memory location of interest (e.g. A) and if so use the value of A present in the store queue.

2.1.2 Sequential Consistency

Sequential consistency is a memory consistency model for parallel programs. It is described by Lamport [46] as follows:

“[The] result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.”

The definition has two parts. Firstly, the results of any execution must agree with the output of some *strict total ordering* of the memory operations of all processes²; and secondly, program order must be maintained within each process.

For example, assume that a parallel program has two shared integers, X and Y , which are initially 0. Program operations for this parallel program are described for two processes as follows:

1. At process P0:

- (a) Set $X \leftarrow 1$.
- (b) Set $Y \leftarrow 2$.

2. At process P1:

- (a) Read Y .
- (b) Read X .

Program order requirement: In order to satisfy program order, statement 1a must appear to have happened before statement 1b; and statement 2a must appear to have happened before statement 2b. Let these constraints be denoted as $1a < 1b$ and $2a < 2b$; where the notation “ $<$ ” is read as “happens before.”

Sequential consistency requirement: Sequential consistency is achieved when it appears that operations have been executed in some interleaved fashion that agrees with the constraints defined by program order. Thus, at statements 2a and 2b, process P1 must observe one of the following combinations for values of Y and X :

²This thesis refers to a *process* as the entity that executes a program. A process may be a hardware processor, a software process, a thread, etc. Exceptions are made when quoting references, or when “processor” can only mean a hardware processor in the context in which it is used (e.g. multiprocessor).

- ($Y = 0, X = 0$.) This is the case when statements 2a and 2b both appear to have happened before 1a and 1b. Therefore, the ordering can be listed as:

$$2a < 2b < 1a < 1b.$$

- ($Y = 2, X = 1$.) This is the case when statements 2a and 2b both appear to have happened after 1a and 1b. The ordering for this case would be:

$$1a < 1b < 2a < 2b.$$

- ($Y = 0, X = 1$.) In this case, the observation of $X = 1$ indicates that X has been written by P0 before it was read by P1, thus $1a < 2b$; and the observation of $Y = 0$ indicates that P1 has read Y before P0 modifies Y , thus $2a < 1b$. Given these observations, four ordering of events are possible that would satisfy sequential consistency:

$$2a < 1a < 2b < 1b,$$

$$2a < 1a < 1b < 2b,$$

$$1a < 2a < 1b < 2b, \text{ or}$$

$$1a < 2a < 2b < 1b.$$

Any other observation of Y and X values would not satisfy sequential consistency. For example, assume that P1 observes $Y = 2$ and $X = 0$. Observing $Y = 2$ would indicate that $1b < 2a$ as it appears that Y was modified before being read. On the other hand, observing $X = 0$ indicates that $2b < 1a$ as it appears that X was read before being modified. When these observations are considered together with those required for program order at each process, a cycle is formed where: $1a < 1b < 2a < 2b < 1a$ (statement 1a is repeated to indicate a cycle). Thus, no sequentially consistent ordering of the statements would be possible.

The parallel program shown in the example above is non-deterministic in the sense that any one of the sequentially consistent outcomes can be expected. To have some degree of determinism, synchronization of processes is introduced so that statements or groups of statements are executed in some prescribed order. This is illustrated below.

2.1.2.1 Synchronization

Shared variables may be used for synchronization. In a producer-consumer example, the consumer process may repeatedly check the value of a shared variable f until it is changed to some expected value by a producer process. The expectation of the consumer process is that all memory operations in the producer's program order before the write to f would have been completed. For instance, if the statements executed by the producer process were as follows:

1. $X \leftarrow 100$.
2. $Y \leftarrow 200$.
3. $f \leftarrow 1$.

Then the consumer process would expect to see values 100 and 200 if it read the values of variables X and Y respectively.

In the example above, our expectation is that a shared memory system will ensure that writes to X and Y complete before the write to f ; in other words, we treat f as a variable of special significance — one that synchronizes the processes. The shared memory system, however, holds no such view and would treat all three write operations equally. To be sequentially consistent it must ensure that the write to X completes before the write to Y , and that the write to Y completes before the write to f ; since any write operation may potentially be used for synchronization. That is it must ensure that *all* write operations complete in the order in which they occur in the source code to satisfy sequential consistency.

Relaxed consistency models, introduced next, relax this constraint by associating memory operation with labels that give special significance to memory operations used for synchronization.

2.1.3 Relaxed Memory Consistency

Relaxed memory consistency models provide weaker guarantees to the programmer but, in exchange, offer the potential of greater performance. This is achieved by adding synchronization semantics to the shared memory programming paradigm that allow the shared memory system to recognize situations when it can relax the requirement for execution of memory operations in strict program order. We consider here two relaxed memory consistency models, weak consistency and release consistency.

2.1.3.1 Weak Ordering

Dubois *et al.* [25] introduce a consistency model called *weak ordering* where shared variables are classified into those that control the concurrent execution — synchronizing, and those that do not — non-synchronizing. Accesses of these variables are accordingly classified as synchronizing or non-synchronizing accesses.

Synchronizing accesses are strongly ordered in the sense that a program would be sequentially consistent if it only used synchronizing accesses. On the other hand, *non-synchronizing accesses* only need to be ordered with respect to synchronizing accesses; as long as program order is maintained. Thus, for any sequence of non-synchronizing accesses $N = \langle n_1, n_2, \dots, n_k \rangle$ that occur between synchronizing accesses s_1 and s_2 (in some process):

- Access s_1 must appear to have completed before any of the accesses in N .
- All accesses in N must appear to have completed before access s_2 .
- Accesses in N can complete in any order.
- Program order is maintained for all accesses: $s_1, n_1, n_2, \dots, n_k, s_2$.

To illustrate weak ordering, the producer-consumer example presented earlier is revisited. Variables X and Y are labeled as non-synchronizing, while f is labeled as synchronizing such that:

1. At the producer:

- (a) Set $X \leftarrow 100$. (Non-synchronizing.)
- (b) Set $Y \leftarrow 200$. (Non-synchronizing.)
- (c) Set $f \leftarrow 1$. (Synchronizing.)

2. At the consumer:

- (a) While $f \neq 1$, repeat this statement. (Synchronizing.)
- (b) Read X . (Non-synchronizing.)
- (c) Read Y . (Non-synchronizing.)

Given these labellings, weak ordering can be satisfied as long as: 1) statements 1a and 1b complete before 1c, 2) statement 1c completes before the last iteration of 2a (statement 2a-last), and 3) statement 2a-last completes before statements 2b and 2c. The advantage of weak ordering is that a distributed memory system may begin executing statement 1b before statement 1a has completed; thereby avoiding the added overheads associated with enforcing an ordering between statements 1a and 1b.

Consider an extended variation of the producer-consumer example where, at the producer, shared variables X and Y are read by the producer after synchronizing statement 1c, such that:

1. At the producer:

- (a) Set $X \leftarrow 100$. (Non-synchronizing.)
- (b) Set $Y \leftarrow 200$. (Non-synchronizing.)
- (c) Set $f \leftarrow 1$. (Synchronizing.)
- (d) Read X . (Non-synchronizing.)
- (e) Read Y . (Non-synchronizing.)

2. At the consumer: (same as before)

To satisfy weak ordering, new statements 1d and 1e cannot be executed until synchronizing statement 1c has completed. However, this is not necessary for correctness as the purpose of statement 1c is to indicate to the consumer that statements 1a and 1b have completed. Therefore, weak ordering is enforcing an ordering of events at the producer that is unnecessary and a cause for added overheads. This issue is addressed by release consistency.

2.1.3.2 Release Consistency

Release consistency (RC) [30] is derived by extending the memory access classification used in weak ordering. These extensions are illustrated in figure 2.1. The hierarchy for weak ordering is shown on the left, while the hierarchy for release consistency is shown on the right. As described earlier, weak ordering classifies shared memory variables into synchronizing and non-synchronizing types. This is extended for release consistency in two ways:

- First, shared memory accesses are labeled as *competing* or *non-competing*. Accesses are labeled as competing if they form a conflicting pair — which is a pair of accesses for which the ordering affects the final result. For example, accesses $X \leftarrow 1$ and $Y \leftarrow X$ would form a conflicting pair as the order in which they are executed affects the final outcome; whereas $X \leftarrow 1$ and $Y \leftarrow 2$ would not form a conflicting pair. Accesses that are not part of any conflicting pairs are labeled as non-competing. In release consistency, only competing accesses are classified as synchronizing and non-synchronizing accesses. (Weak ordering, in contrast, does not make this distinction and assumes that all accesses are competing.)
- Second, accesses that have been labeled as competing and also synchronizing are further classified as *acquire* or *release* accesses. This classification allows for consistency rules on the ordering of accesses to be further relaxed as follows:
 - Whereas all preceding non-synchronizing accesses have to complete before a synchronizing access in weak ordering, release consistency only requires that preceding non-synchronizing accesses complete before a release access.
 - Whereas all preceding synchronizing accesses have to complete before a non-synchronizing access, release consistency only requires that preceding acquire accesses have to complete before a non-synchronizing access.

Considering the extended producer-consumer example from earlier, using acquire and release classification gives:

1. At the producer:

Figure 2.1: Categorization of memory accesses for weak ordering and release consistency. Release consistency extends weak ordering by 1) classifying accesses as competing or non-competing, and 2) further classifying synchronizing accesses as acquire or release types.

- (a) Set $X \leftarrow 100$. (Non-synchronizing.)
- (b) Set $Y \leftarrow 200$. (Non-synchronizing.)
- (c) Set $f \leftarrow 1$. (Release, synchronizing.)
- (d) Read X . (Non-synchronizing.)
- (e) Read Y . (Non-synchronizing.)

2. At the consumer:

- (a) While $f \neq 1$, repeat this statement. (Acquire, synchronizing.)
- (b) Read X . (Non-synchronizing.)
- (c) Read Y . (Non-synchronizing.)

To satisfy release consistency, at the producer the non-synchronizing statements 1a and 1b must complete before the release at 1c, while at the consumer the final iteration of the acquire statement 2a-last must complete before executing non-synchronizing statements 2b and 2c. So far, this is similar to weak ordering. What sets release consistency apart from weak ordering is whether statement 1c is synchronized with respect to statements 1d and 1e. Whereas in weak ordering these non-synchronizing statements are allowed to execute only after synchronizing statement 1c has completed, in release consistency they may be performed before 1c has completed. This is because release consistency does not require the completion of release accesses before the non-synchronizing accesses that follow are executed.

Release consistency can further be categorized as *eager release consistency* (ERC) or *lazy release consistency* (LRC) [43]. The differentiating characteristic is the point at which non-synchronizing accesses are completed.

In ERC, non-synchronizing accesses are completed before release accesses. LRC, on the other hand, delays completion until another process performs an acquire access. At this point all previous release accesses, and non-synchronizing accesses before these release accesses, must be complete before the acquire access can proceed.

The difference between ERC and LRC can be loosely described as “push” (eager) versus “pull” (lazy). Intuitively, the pull approach will result in fewer data transfers because accesses only need to be complete with respect to the processes that require them to be complete; whereas the push approach would complete all non-synchronizing accesses with respect to all processes regardless of whether those processes require them to be complete or not.

2.1.4 Other Consistency Models

Entry consistency (EC) was introduced by Bershad and Zekauskas [14] as a consistency model where shared variables are associated with synchronization variables that protect critical regions. The shared variable becomes consistent when its associated synchronization variable has been acquired. Synchronization variables can be acquired exclusively or non-exclusively. Multiple non-exclusive acquires to a synchronization variable are allowed to proceed simultaneously, while exclusive acquires are allowed to proceed only when there exists no other non-exclusive and/or exclusive accesses in progress. This allows for multiple concurrent readers that use the non-exclusive mode. Exclusive mode must be used if the associated shared variables are to be modified.

Scope Consistency (ScC) was introduced by Iftode *et al.* [40] as a consistency model that defines a session to be the interval between when a scope opens and when it closes (e.g., a critical section protected by a lock). Write operations during a session must be completed with respect to its scope when the session ends. Later, before a new session of that scope is allowed to open at a process p , any write that have previously been performed with respect to that scope must be performed at process p . Program order must still be maintained among operations at p . ScC is similar to EC but differs in that shared variables are implicitly associated with synchronization events.

2.2 Software Distributed Shared Memory

As mentioned previously Software Distributed Shared Memory (SDSM) systems provide programmers with a shared memory programming environment across distributed memory architectures. SDSM systems may be categorized broadly as page-based or object-based:

- *Page-based*: Shared memory is viewed as a shared address space that is partitioned into equally sized pages. Coherency is managed in units of pages. These

systems typically make use of the underlying virtual memory system.

- *Object-based:* Shared memory is an abstract storage space for storing shared objects of variable sizes. Coherency is managed in application specific granularity that is defined by objects. These systems are typically associated with object-oriented languages.

This thesis focuses on page-based approaches.

This section reviews some of the techniques that are used for implementing Software Distributed Shared Memory (SDSM) systems. The discussion is framed by describing aspects of some well known SDSM implementations.

2.2.1 IVY

IVY [47] is an SDSM that represented a “proof of concept,” i.e. that it was indeed possible to implement an SDSM system. Its main design objective was to produce a tractable system that would be easy to understand. This section describes the key elements of the IVY SDSM, many of which were used in later generations of SDSM systems.

IVY is built into an operating system and memory coherency is managed in units of pages, where the page size can be any multiple of that used by the underlying Memory Management Unit (MMU). Thus IVY is a page-based SDSM system.

Pages in IVY are marked as read-only, or read-writable. Read-only pages may have multiple copies residing in the physical memories of multiple processes. Read-writable pages may only have one copy that resides at a process that is referred to as the “owner.” Before a page can become read-writable, all copies of the page at other processes must be invalidated.

Invalidated pages are only validated again when they are next accessed by the process. To validate a page, an update request is delivered to the owner of the page. Upon receiving the update request, the owner process will send its view of the page data to the requesting process and, if the page is read-writable, also revert the state of the page in question to read-only (as there will then be multiple copies of the same page). Once the update has been received at the requesting process, any modifications that were made by the owner process (and any previous owner processes) will now be visible.

The protocol described above depends on the ability to send a request to the owner process, however owner processes change all the time and so it is necessary to keep track of where the owner process is. Li [47] describes three algorithms for keeping track of the owner process’ location:

- *Centralized Manager*: A single process serves as a directory for all pages. Information about all the pages, including the location of the owner, may be obtained from this process.
- *Fixed Distributed*: Each page is assigned a process at which information for that page is kept. A mapping function, $H(p)$, points to the process that holds page p .
- *Dynamic Distributed*: Information about the page is maintained at the page owner. A mapping function $probOwner(p)$ is used to indicate where the owner of the page might be. At initialization, the $probOwner$ for each page is set according to a fixed distribution. This is updated when the process relinquishes ownership of a page to a new owner, or when the process becomes the new owner of the page. Hence, the present owner of a page can always be found by following the $probOwner$ pointers at each process.

This coherency protocol guarantees that memory reads will always observe the most recent write. The IVY SDSM is therefore sequentially consistent. The coherency protocol, however, makes IVY susceptible to a problem called “false sharing.”

2.2.1.1 The False Sharing Problem

Assuming that memory is managed as described earlier for IVY, then false sharing arises when two or more writers wish to modify *different* regions of shared memory that are located within the *same* page. As read-writable pages are only permitted to exist at one process, this use case will generate competing invalidation and update request messages from the processes involved.

For example, consider a program that sums the contents of an array in parallel. The work load is divided by partitioning the array's index range evenly between the available processes. Each process writes its partial sum into an array that is indexed by process IDs.

1. (Get the input array elements.) $A \leftarrow$ array of N elements.
2. (Get partial sum array.) $S \leftarrow$ allocate an array of P elements; and also initialize the partial sums to zero.
3. (Calculate partial sum.) At process u , $S[u] \leftarrow \sum_i A[i]$, for each element i that has been assigned to process u .
4. (Sum up the partial sums to get the global sum.) This is executed at one of the processes.

Using a coherency protocol like IVY, the page containing the partial sum array S will be a source of contention between processes whose partial sums reside in the

same page. The effect of false sharing in this case would be to serialize the affected processes, since they would all have to take turns to write into S . Li [47] notes that this situation can be avoided by ensuring that unrelated shared regions of memory be allocated in different pages. For the summation example above, this can be achieved by placing padding (unused memory) between the partial sums, such that partial sum for each process is guaranteed to reside in its own page. However, this results in page fragmentation as only a small portion of each page would contain data. Moreover this strategy requires a programmer to know details about the underlying shared memory system. A better solution for this problem is introduced in the Munin SDSM.

2.2.2 Munin

Munin [13, 22] is a page-based SDSM that solves the false sharing problem. To enable this Munin uses eager release consistency (ERC). The advantage of using this consistency model is that propagation of modifications can be delayed until a release access. Until such an access is encountered, Munin tracks modifications to shared memory using a protocol called “twinning-and-diffing.” The protocol consists of (and is named after) two operations:

- *Twinning*: This involves making a twin, which is a copy of the page. Twinning is performed before modifications are made to the page.
- *Diffing*: This compares the contents of a potentially modified page with the twin and encodes any differences into a diff. Changes to the page can thus be propagated by sending the diffs to other processes.

Twinning-and-diffing solves the false sharing problem because changes made by different processes to different parts of the same page are tracked and may be merged at a later stage.

Use of twinning-and-diffing is only effective for relaxed consistency models (i.e. weak ordering and release consistency) because such models allow propagation of changes to be delayed. For sequential consistency, where no distinction is made between non-synchronizing and synchronizing accesses, every access could potentially be a synchronizing access. Thus, if twinning-and-diffing were to be used together with sequential consistency, then for the summation example given previously, twins and diffs would need to be made on every update of the partial sum array, S . (Each time a diff is made, overheads proportional to the page size would be incurred — as the twinning involves copying the whole page and diffing requires the whole page to be read.)

2.2.3 TreadMarks

TreadMarks [43, 42, 9] is a page-based SDSM system. It improves on Munin by using lazy rather than eager release consistency. This delays the propagation of modifications to an acquire operation rather than eagerly propagating modifications on a release operation.

TreadMarks also uses an invalidation protocol together with lazy release consistency (LRC). In the invalidation protocol, diffs are not sent to the acquiring process on the acquire access; rather, only write notices that indicate the set of modified pages are sent. Pages indicated by the write notices are invalidated when the notices are received by the acquiring process, and diffs for an invalidated page are only requested in the future if the process accesses the page.

Use of the invalidation protocol requires each process to store modifications that have been made to shared memory so that they may be sent if requested by a remote process. This protocol raises the possibility that stored diffs may overlap in that they modify the same region. To address this, TreadMarks uses timestamps to order diffs temporally. Storage of diffs also raises the possibility that there may be too many diffs to store in available memory. To address this issue TreadMarks uses "garbage collection."

2.2.3.1 Timestamps

As noted above, it is necessary for stored diffs to be applied in the correct order when they are later required, otherwise the shared memory would be inconsistent and also incoherent. (Memory may become incoherent when different processes apply the diffs in different orders resulting in disagreement on the correct value.)

For example, if process P3 has just performed an acquire access, and the previous two processes to release were P2 and P1 (with P2 being the most recent), then overlapping modifications by P2 and P1 must be applied in the correct order (P1's changes, and then P2's changes).

To determine the correct order in which to apply diffs, TreadMarks divides a process' execution into distinct intervals that are punctuated by synchronizing access (release/acquire); that is, an interval contains a set of consecutively executed non-synchronizing accesses. Each interval is associated with a virtual time such that, if i is the timestamp for the present interval, then the timestamps for the previous and next intervals would be $i - 1$ and $i + 1$ respectively. Therefore, by associating modifications with the interval in which they were made, it is possible to tell which order they should be applied to memory.

Intervals are managed using an array of times called vector time [50]. The length of the array is equal to the number of processes and is indexed by process ID. Let

$V_i[0 \dots P-1]$ be a vector time at process i ; where P is the number of processes. Then at each process i , $V_i[i]$ will indicate the virtual time of the current interval at i . However, for each other process j ($j \neq i$), $V_i[j]$ will instead hold the virtual time of the latest interval that process i had witnessed from process j ; that is, write notices from j up to that interval have been received.

On an acquire access, a request is sent to the last process that performed a release access. Let i be the acquiring process and j be the last process that performed a release access. The request would then contain the vector time V_i . When received by process j , the following steps are taken:

1. (Pack the write notices that process i has not witnessed.) For each process u such that $V_i[u] < V_j[u]$, pack into a reply message the write notices from process u that occurred in intervals $V_i[u] \dots V_j[u]$ at u .
2. (Calculate a new vector time for process i , $V_{i'}$) For each process u , let $V_{i'}[u] \leftarrow \max(V_i[u], V_j[u])$
3. (Reply.) Send the packed write notices together with the new vector time $V_{i'}$.

When the reply is received at process i , the write notices are processed and the modified pages indicated in the notice are invalidated. Write notices are also stored as they provide information about where diffs should be requested from in the future.

2.2.3.2 Garbage Collection

Invalidation is used in TreadMarks' implementation of LRC. While this helps to reduce the amount of diff data that needs to be communicated (since diffs are not communicated if no other process requests them), substantial amounts of memory are required to store diffs for processes that have not requested them, just in case they request them sometime in the future [77]. The lifetime of a diff tends to grow exponentially with respect to the number of processes because the chance that not all processes will require the diff increases with the number of processes. TreadMarks employs a garbage collection phase periodically when memory usage exceeds a threshold [42]. During garbage collection, each process i will:

1. (Validate every page that has been modified at i .) This involves fetching diffs from other processes, as indicated by its write notice records.
2. (Update the copy-set of every page.) The copy-set of a page indicates which processes have valid copies of that page.
3. (Reset.) Invalidate every other page and discard all records of diffs, write notices, and also intervals (i.e., virtual time is reset).

Access to invalid pages after garbage collection can be resolved by requesting a page from any of the processes in the copy-set for that page.

The protocol synchronizes all processes and thus attempts are made to have garbage collection coincide with TreadMarks barrier operations (which synchronizes processes). The implication of this is that while some barriers may exhibit low overheads, since diffs are not sent, others may exhibit higher than normal overheads when garbage collection is performed.

To reduce the impact of garbage collection, Amza *et al.* [11, 10] describe adaptive protocols that switch between “single-writer” and “multiple-writer” modes. Memory consistency is maintained using sequential consistency when in the single-writer mode, while the multiple-writer mode is supported using release consistency. Switches between update and invalidate protocols are also discussed. These adaptive techniques reduce the number of diffs generated and likewise the impact of garbage collection.

2.2.4 Home-based Lazy Release Consistency SDSMs

Home-based lazy release consistency (HLRC) introduced by Zhou *et al.* [77] addresses the diff storage problem faced by TreadMarks. In HLRC SDSMs, each shared page is assigned a home. On a release access, processes will send all outstanding modifications to their respective home processes; that is, the modifications for page p would be sent to the home of page p . On an acquire, any page which has been modified by other processes will have been invalidated, so access to that page is resolved by fetching an updated copy from the assigned home process. Thus, the home process for a page serves as a central point where all diffs for that page are sent, and from where all updated views are obtained. Memory requirements for storing diffs in HLRC are lower than for LRC because the lifetime of a diff is only as long as the time to process the release access.

The notion of a home for each page introduces the concept of *page locality*. A page is considered to be local to a process if the process is also the home for the page. Otherwise, the page is considered to be remote to that process. Memory access overheads are comparatively lower for local pages than for remote pages as the local copy of the page defines the up-to-date state of the page (also referred to as the global view of the page). Accesses to remote pages, on the other hand, involve data transfer if the page is invalid, and incur twinning costs if the access is a write. Subsequently, if the page is modified, the modifications need to be diffed and sent to the page home at the next release. Therefore, it is desirable to map page homes to processes that access them regularly in order to minimise the overheads associated with accessing remote pages [76, 72]. Home assignment may be fixed when memory is allocated, or dynamically reassigned by home-migration.

SCASH [35] is a home-based SDSM system that allows shared memory regions to be allocated with three different page placements:

- *Direct*: The homes for the allocated pages are all set to the same specified process.
- *Round-robin*: The homes for the allocated pages are distributed to all processes in a round-robin fashion.
- *Blocked*: Consecutive pages are divided into P approximately equal sized blocks; where P is the number of processes. Each process is then assigned to be the home for a block of pages.

Home-migration is supported in SCASH by exposing an interface that allows the user to specify explicit home migration using the `scash_home` operation.

Page accesses patterns can be classified as single-writer or multiple-writer. Single-writer scenarios are common in parallel applications [10, 27].

During barrier points (full synchronization of all processes), a home-based SDSM may migrate the home of a page to a process if that process is the single-writer for the page. This is justified because the only modification to the page exists at the single-writer process, thus it holds the most up-to-date view of the page.

Home-migration when used in this way reduces communication overhead, as no diff needs to be sent. It also serves to keep shared memory coherent. JIAJIA [39] is a home-based SDSM that implements home-migration at barrier points.

For home-migration to work, it is necessary for all processes to know where the current page home is, or at least be able to find the page home when this information is needed. Barrier-based home-migration as described above is limited to barrier points. This is relatively easy to implement. To enable home-migration outside of barriers is more complex. Fang *et al.* [27] describe how home-migration is implemented for their distributed Java virtual machine (JVM). As Java does not have barrier operations, home-migration has to be performed in a one-sided fashion. The challenge in this situation is to share information about the new home process for a page (or in the case of the JVM, a shared object). Fang *et al.* describe three techniques for sharing this information in a one-sided fashion:

- *Broadcast*: This technique broadcasts the change in page home to all other processes. The danger of this approach is that when a process accesses an invalid page and sends a page fetch request to the page home, it receives a home migration notice for the page while waiting for the page data to arrive. Thus, precautions must be taken to handle this situation correctly.
- *Home Manager*: This technique assigns each page to a home manager process which maintains a pointer to the current home for that page. Thus, when a page

is migrated, only the home pointer at the home manager needs to be updated. When a message for some SDSM operation is delivered to the wrong process the message can be redirected to the home manager, and then on to the correct home process. (SCASH has been extended to support home-migration using the home manager technique [34].)

- *Forwarding Pointer*: In this technique, communication is performed only between the current home process and the former home process. Since all other processes know at least one of the former home processes, messages can be forwarded through the chain of former home processes until they are delivered to the correct home process. The advantage of this approach is that messages for operations that require communication with the home process while home-migration is taking place either arrive at the former home process before or after the migration is complete. Thus, inconsistent states such as that described for the broadcast technique are avoided. (The approach taken by this technique is similar to IVY's "Dynamic Distributed" algorithm for tracking the last writer for each shared page; see section 2.2.1.)

The trade-off to consider when picking a dynamic home migration technique is between the cost of performing the migration, and the impact of the technique on the overheads of all subsequent SDSM operations.

In the broadcast technique, the migration is costly as it requires communication with all other processes, however, subsequent operations are unaffected and interact directly with the new home process.

At the other extreme, the forwarding pointer technique has less migration overhead, requiring communication only between the new and former home processes, but messages for subsequent SDSM operations may need to be forwarded multiple times in order to be delivered to the correct home process.

The home manager technique lies in between the broadcast and forwarding pointer techniques in terms of migration cost and impact on subsequent operations.

2.3 Message Passing Interface

Message passing is a programming paradigm that exposes the physical layout of distributed memory systems. In message passing, multiple processes execute in parallel, using messages to share data and perform process synchronization. On a distributed memory system, processes may execute at different nodes and messages may be sent across the network linking these nodes. The resemblance to the hardware makes message passing an intuitive paradigm for writing programs on cluster computer systems.

Message passing programming can be done directly using the Application Programming Interfaces (APIs) of hardware interconnects such as Quadrics [61], Myrinet [16], or InfiniBand [1]. However, this approach has poor portability across different clusters with different networking hardware, and thus message passing programming is more commonly done using abstract APIs such as Unix Sockets [67], Parallel Virtual Machine (PVM) [29], or the Message Passing Interface (MPI) [51, 52]. Of these, MPI is now the dominant method for message passing programming on cluster computers.

One of the key reasons for the success of MPI is that it was defined collaboratively by implementors (e.g. manufacturers of high-performance network interconnects), library writers (that use MPI), and programmers [32]. Involvement of implementors improves the portability of the interface as any potential implementation issues can be brought to light and resolved at the design stage. Involvement of library writers and programmers ensures that every addition to the MPI standard has a purpose; “while no one was using all of the MPI-1 routines, essentially all MPI-1 routines were in use by someone” [31].

Presently, message passing functionality prescribed by the MPI standard is categorized in two parts: MPI-1 [51] and MPI-2 [52]. MPI-1 covers fundamental message passing operations for point-to-point and collective operations. MPI-2 introduces Remote Memory Access (RMA), dynamic process management, parallel I/O, and a number of other changes.

The following discussion focuses on aspects of MPI that were used to implement the Danui SDSM library or are relevant to SDSM systems. Readers may refer to [51] and [52] for more complete information about the MPI standard.

2.3.1 MPI Communicators

MPI is designed to be used by library implementors in a modular fashion. A cornerstone concept in MPI that makes this possible is the use of MPI communicators. Each communicator defines a communicator space that is isolated from other communicator spaces. This is useful for library implementation as it can be used to isolate internal library communication from that used by the program using the library.

2.3.2 MPI Point-to-point Communication

Point-to-point communication involves the transfer of a message between a sender and a receiving process. This is fundamental to message passing as it can be used to implement collective communication operations (that may involve more than two processes). The following describes 1) blocking and non-blocking communication, 2) how message buffers are specified, and 3) how messages and processes are identified.

2.3.2.1 Blocking and Non-blocking MPI Communication

Point-to-point communication may be blocking or non-blocking. The difference between these are characterized by the guarantees, pertaining to memory access on message buffers, that the MPI implementation must provide. Such accesses occur as part of the communication process: the sender reads from a send buffer, while the receiver writes into a receive buffer.

For calls to blocking communication functions (`MPI_Send` and `MPI_Recv`), access to the message buffer (read/write) is guaranteed to have completed when the call returns.

For calls to non-blocking communication functions (`MPI_Isend` and `MPI_Irecv`), access to the message buffer may not have completed when the call returns. Instead, non-blocking communication functions initialize a handle (`MPI_Request`) that can then be used with other functions (e.g. `MPI_Test`, `MPI_Wait`) to verify completion of the memory accesses to the message buffer.

2.3.2.2 Message Buffers

Message buffers are specified using three parameters: address, count, and data type. The address parameter specifies the location of a message buffer in the process' address space, the count parameter specifies the number of elements, and the data type parameter specifies the type of element.

Data types are specified using a `MPI_Datatype` object. Basic data types are `MPI_INT` for integers, `MPI_DOUBLE` for doubles, etc.

Use of data types addresses two issues: Firstly, it allows processes executing on different nodes of a heterogeneous systems to communicate seamlessly. For example, nodes using different byte-orders for integers ("endianness") can still communicate using the `MPI_INT` data type as the MPI library can reorder bytes accordingly. Secondly, it enables user defined data types to be constructed from existing data types. This is useful, for example, for creating data types that represent non-contiguous regions (e.g. by using `MPI_Type_vector`).

2.3.2.3 Message Tags and Process IDs

As mentioned earlier communication between processes is performed in the context of a communicator space represented by a communicator object. In point-to-point communication, destination and source processes are identified using unique process ranks (or IDs) within the communicator space. Messages are also identified via message tags.

A sender process must specify a destination process ID and a message tag. A receiver process may specify a source process ID or elect to accept a message from any process by specifying `MPI_ANY_SOURCE` as the source. In a similar fashion, a receiver

process may specify a specific message tag or elect to accept a message of any tag by specifying `MPI_ANY_TAG`. When a receive operation that used wildcard values `MPI_ANY_SOURCE` and `MPI_ANY_TAG` completes, the source process ID and message tag can be obtained from a `MPI_STATUS` object that is initialized by the receive operation.

2.3.3 Collective Communication

Collective communication operations involve all processes within an MPI communicator space and require that each process call the same collective operation with matching parameters before the operation can complete. There are a variety of collective operations including: `MPI_Barrier`, `MPI_Reduce`, and `MPI_Allreduce`.

`MPI_Barrier` provides synchronization for processes of a specified communicator. No message buffers are used with `MPI_Barrier` operations.

A selection of MPI collective operations for broadcast, scatter, gather, all-gather, and all-to-all collective operations are shown in figure 2.2. Each row of the grids represent a data location at a process. The five collective operations illustrated are:

- **MPI_Bcast:** This is used to broadcast data from a process to all other processes. The source process is known as the root process. In the figure, root process P0 broadcasts A0 to all other processes. After the operation (grid G2), all processes have A0.
- **MPI_Scatter:** This is used to distributed multiple similarly sized message buffers to multiple processes; where the number of send buffers is the same as the number of processes. In the figure, root process P0 scatters the values A0...A3 (grid G3) to the other processes. After the operation (grid G4), all processes have values from P0 in their receive buffers that is the send buffer from P0 that corresponds to their process rank.
- **MPI_Gather:** This is the opposite of `MPI_Scatter`. Starting from grid G4, each process initially has values in a send buffer. After the gather operation (grid G3), the root process (P0) has the data from all the send buffers.
- **MPI_Allgather:** This gathers data from all processes to all processes. Unlike `MPI_Gather` there is no root process for this operation. The effect is similar to `MPI_Gather` except that all processes will have all the gathered values.
- **MPI_Alltoall:** This is a mix of `MPI_Scatter` and `MPI_Allgather` operations. In this operation, all processes have multiple send buffers containing data specific to each remote process. Similarly, all processes have multiple receive buffers to receive data from all remote processes.

Figure 2.2: Data movement in collective communication operations (Adapted from figure 4.1 of “MPI: A Message-Passing Interface Standard” [51]). Each row of a grid represents data locations at a process.

Figure 2.3: Data movement in reduction communication operations.

`MPI_Reduce` and `MPI_Allreduce` are collective communication functions that involve computation on elements in a message buffer that reduce buffer values into, for example, a sum (`MPI_SUM`), a maximum (`MPI_MAX`), etc. Figure 2.3 illustrates data movement in reduction operations. For `MPI_Reduce`, values from each process `A0...A3` are reduced into `X` at root process `P0`. For `MPI_Allreduce`, values from each process `A0...A3` are reduced into multiple copies of `X` at all processes. MPI provides several reduction operations. For example, `MPI_SUM` can be used to obtain a global sum, and `MPI_MAX` can be used to obtain a global maximum.

2.3.4 Remote Memory Access

MPI-2 [52] extends the MPI standard with interfaces for remote memory access (RMA) operations. To enable RMA operations, MPI-2 provides the `MPI_Win_create` collective operation to associate arbitrary memory regions with a “window” that will be exposed to RMA operations. The address and size of the memory region supplied is allowed to be different at each process. In fact, the size may even be zero should a process not wish to expose any memory for RMA operations originating from other processes. After the `MPI_Win_create` operation, remote memory can be addressed by providing the window (as context), the process rank, and an offset.

Three functions are provided to access memory that has been associated with a

window:

- **MPI_Put**: Write to memory at a remote process.
- **MPI_Get**: Read from memory at a remote process.
- **MPI_Accumulate**: Contribute a partial value to a global value at a remote process. For example, a partial sum may be contributed by incrementing a global sum at a remote process using the **MPI_SUM** operation (similar to reduction operations).

These functions can only be used in epochs that are defined as follows:

1. Access epochs define when RMA operations may be used.
2. Exposure epochs define when local memory may be the target of RMA operations originating from remote processes.

There are three different approaches to enter epochs: collectively, pair-wise, and one-sided. The collective approach involves all processes while the pair-wise approach involves groups of processes. Both these approaches are referred to as “active” RMA operations. The third approach enables a process to effect an exposure epoch at a remote process without active coordination from that process, as such the associated operations are said to be “passive.” Details for each approach are as follows:

1. Collectively: **MPI_Win_fence** is a collective function that synchronizes all processes and begins access and exposure epochs at all processes. That is, all processes may issue RMA operations to any remote process. The epochs are terminated by a second call to **MPI_Win_fence**. RMA operations are guaranteed to have completed upon returning from the second **MPI_Win_fence** call.
2. Pair-wise: An access epoch is started at a process and an exposure epoch is started at a second process.
 - (a) **MPI_Win_start** is used by the first process to initiate an access epoch. **MPI_Win_complete** is later used to terminate the access epoch. RMA operations are guaranteed to have completed upon returning from the **MPI_Win_complete** call.
 - (b) **MPI_Win_post** is used by the second process to initiate an exposure epoch. **MPI_Win_wait** is later used to terminate the exposure epoch. A **MPI_Win_wait** call will block until the first process has called **MPI_Win_complete**.
3. One-sided: **MPI_Win_lock** and **MPI_Win_unlock** is used to obtain access to a window at some target process. Acquiring a lock on a memory region at a target process begins an access epoch at the locking process and an exposure epoch at

the target process. Releasing the lock terminates the epochs at both processes. Locks may be acquired using either exclusive (`MPI_LOCK_EXCLUSIVE`) or shared (`MPI_LOCK_SHARED`) modes. Obtaining an exclusive lock guarantees that only the process with the lock has an access epoch. Obtaining a shared lock does not have that guarantee, but it does imply that no other process has an exclusive lock.

Conceptually, each process has private and public copies of memory regions that have been associated with an `MPI_Win` handle. Local memory accesses are performed on the private copy, while RMA operations from remote processes are performed on the public copy. The synchronization functions listed above cause the private and public copies to be reconciled. For one-sided synchronization involving `MPI_Win_lock` and `MPI_Win_unlock`, modifications by a remote process to the public copy of some target process is not guaranteed to have propagated to the private copy of the target process until the target process calls `MPI_Win_lock` and `MPI_Win_unlock` on its own window. As a result of this conceptual model, three restrictions on RMA operations apply [52]:

1. "A location in a window must not be accessed locally once an update to that location has started, until the update become visible in the private window copy in process memory."

This restriction is necessary since it would be erroneous to rely on information from a region that is being modified.

2. "A location in a window must not be accessed as a target of an RMA operation once an update to that location has started, until the update becomes visible in the public window copy."

This is the reverse of the first restriction. The exception, for this restriction, is when `MPI_Accumulate` is used for reduction as the same data has to be updated in this case.

3. "A put or accumulate must not access a target window once a local update or a put or accumulate update to another (overlapping) target window have started on a location in the target window, until the update becomes visible in the public copy of the window. Conversely, a local update in process memory to a location in a window must not start once a put or accumulate update to that target window has started, until the put or accumulate update becomes visible in process memory. In both cases, the restriction applies to operations even if they access disjoint locations in the window."

The first part of this restriction says that updates from any source may not overlap. The second part restricts concurrent access to private and public copies of the same window, even if they are at disjoint locations in the window. That

is, at any exposure epoch, either the private or public copy may be modified, but not both.

2.3.5 MPI and Threading

MPI supports the use of multiple threads at four different levels that can be specified by initializing MPI using the `MPI_Init_thread` function.

1. `MPI_THREAD_SINGLE` is used when the thread that initialized MPI will be the only thread executing.
2. `MPI_THREAD_FUNNELED` is used when multiple threads will be used by the process but only the thread that initialized MPI will make calls to MPI functions.
3. `MPI_THREAD_SERIALIZED` is used when multiple threads will be used by the process and multiple threads would make calls to MPI functions. However, threads will only do so one at a time.
4. `MPI_THREAD_MULTIPLE` is used when multiple threads will be used by the process and multiple threads will make calls to MPI functions. In addition, they may do so concurrently.

2.4 Related Work

There has been limited work on integrated message passing and SDSM systems. There has been slightly more work on integrated message passing with hardware distributed shared memory systems. Some of the early work in these areas are briefly described below.

2.4.1 MIT Alewife Machine

The MIT Alewife machine [7, 6, 5] is an MPP system with a hardware DSM system that fits between a processor and the underlying network to provide a shared memory programming environment. Even so, processors still maintain direct access to the network; enabling “controlled breaking of the shared-memory abstraction by the programmer” [44]. Message passing operations supported by this bypass include: 1) *processor-to-processor* communication involving data transfers between the registers of sender and receiver processors; and 2) *Source-and-destination-coherent* communication involving memory to memory data transfers that also keeps caches coherent at the source and destination processors.

2.4.2 Stanford FLASH

The Stanford FLASH (FLexible Architecture for SHared memory) Multiprocessor [37] is an MPP where each node has a controller, called MAGIC (Memory and General Interconnect Controller), that contains its own processor which enables shared memory to be managed using a mix of hardware and software. A connection-based message passing protocol that transfers data between sender and receiver in a byte-stream was implemented.

2.4.3 TreadMarks Extension

The TreadMarks [43, 42] SDSM was extended by Mirchandaney *et al.* [53] to support data pre-fetch operations that could be inserted by a compiler or the programmer: `DSM_send` and `DSM_recv`. These operations take a shared memory region and a process ID as arguments. `DSM_send` indicates to a matching `DSM_recv` that modifications in the specified shared memory region may be fetched. `DSM_recv` requests modifications from the specified process and ignores any modifications to the region from other processes. The receive call is blocking and will complete only when the matching send has been issued. Since modifications to the region are only obtained from the process that calls `DSM_send`, the implementation saves cost by reducing the number of interrupts to other processes.

2.4.4 Brazos

Brazos [65] is an SDSM that supports an SMMP programming environment [64] where message passing is expressed by a programmer using a “large subset” of MPI [51] operations. Message buffers used with these operations may refer to the same shared memory region, in which case DSM coherency mechanisms would be used to provide coherent message passing. This involves propagation of modifications made to the region to receiver processes; the region would be “invalid on all machines except the receiver[s].”³ In other words, message passing is a synchronization operation between two or more processes that includes pooling of all modifications to receivers.

2.5 Alternative Programming Models

This section describes other programming models including Linda [21], Global Arrays [55, 56, 54], and Partitioned Global Address Space languages [26, 75, 68, 24, 23]

³The quoted text — “invalid on all machines except the receiver” — is taken from an example involving `MPI_Send` and `MPI_Recv` that has only one receiver. Although there is no mention of how `MPI_Bcast` would have been handled, it is reasonable to assume that the same treatment would have been given to all receivers.

that provide alternatives to SDSM and message passing programming.

2.5.1 Linda

Linda [21] is a “coordination language” that is organized around a “tuple space.” The tuple space can serve as both a place for storing shared values (in tuples), as well as a space for coordinating/synchronizing processes. Interaction with the tuple space can be achieved via the following atomic operations:

- `out()`: Places a tuple into the tuple space.
- `in()`: Get a tuple from the tuple space; also removes the tuple from the space.
- `rd()`: Get a tuple from the tuple space; but leave the tuple in the space.
- `inp()`: A non-blocking version of `in()`.
- `rdp()`: A non-blocking version of `rd()`.
- `eval()`: Creates processes to perform operations on some data (an “active tuple”) and then place the results (a “passive tuple”) in the tuple space when operations are completed.

This set of operations makes it possible to coordinate several inter-process interactions: *Point-to-point* communication can be achieved by having a sender process use `out()` to place a tuple in the tuple space that is retrieved by a receiver process using the `in()` operation at some later time. *One-to-many* communication can be achieved by having a sender process use `out()` to place a tuple in the tuple space that is retrieved by multiple receivers using `rd()` later. *Mutual exclusion* can be achieved on any arbitrary tuple since `in()` operations will block until its requested tuple is available in the tuple space — in this case, `in()` is used to acquire a lock, and `out()` would be used to release the lock; a try-lock effect (acquire if available, but to not block otherwise) can be achieved using `inp()`. Shared atomic counters, reduction operations, etc. can be implemented in similar fashions. Finally, *overlapping of computation and communication* is made possible using `eval()`.

Linda’s tuple space concept contrasts with page-based SDSM systems in two ways. Firstly, the tuple space has to be accessed by special operations. In contrast, page-based SDSMs support shared memory access that is similar to a sequential program. Secondly, operations like `eval()` and process synchronization from blocking on `in()` calls mean that the tuple space is more than just shared memory.

2.5.2 Global Arrays

Global Arrays (GA) [55, 56, 54] is a toolkit for multiple processes to concurrently manage and access distributed multidimensional arrays. Motivation for developing the toolkit stems from the observation that, increasingly, parallel computing architectures have memory hierarchies that lead to Non-Uniform Memory Access (NUMA); and that parallel programming for such architectures would require extra programming effort. GA intends to reduce this effort.

Global arrays are distributed matrices that are partitioned into multiple parts that are each stored at a different process. A portion of a distributed matrix that is stored at a process is local to that process, but remote to all other processes. Data can be stored in these matrices by using `put()` — sends/writes the contents of some buffer into a distributed matrix. Data can be retrieved by using `get()` — request and receive or reads the contents of some part of a distributed matrix into a buffer. Reduction operations are supported by `accumulate()`, and operations on shared counter are provided by `get-and-increment()`⁴.

Operations are classified as follows: 1) One-sided — shared memory access that involves only the process that called the operation. 2) Collective — involves all processes. 3) Local — operations that do not require communication. 4) Atomic — operations that involve mutual exclusion. 5) Non-blocking — operations that initiate data transfer and return immediately. Completion of non-blocking operations can be ascertained through the use of various wait operations (e.g., `NGA_NbWait`).

The GA programming model differs from page-based SDSM programming models in the way shared data is conceptually represented and how that data is accessed. In GA, shared data consists of distributed multidimensional matrices, whereas in a page-based SDSM shared data is just a region in shared memory and any meaning that may be attached to that region is entirely user defined. In GA, shared data is accessed via functions such as `put()` or `get()` that transfer data from non-shared buffers to and from matrices, whereas in a page-based SDSM shared data is accessed in the same way as local data. Programs using GA have to copy to and from matrices, while programs using a page-based SDSM can read and write directly from shared memory. However, page-based SDSM systems incur additional overheads in order to provide such convenience. Twinning-and-diffing, for example, is an expensive SDSM overhead. In contrast, GA avoids these overheads by limiting access to matrices to GA functions. As such access detection mechanisms like twinning-and-diffing are entirely unnecessary, since the programmer specifies exactly what should happen using `put()`, `get()`, or other higher-level operations.

⁴More Global Arrays operations can be found at “Global Arrays Operations” <http://www.emsl.pnl.gov/docs/global/Capi.html> (accessed January 2010).

2.5.3 Partitioned Global Address Space Languages

Partitioned Global Address Space (PGAS) languages are based on the idea that a global address space (shared memory) is partitioned and distributed to participating processes. In contrast, SDSM systems do not state whether the shared address space is partitioned. That is, the key difference is that PGAS languages make shared memory partitioning explicit to the programmer. In this respect, PGAS languages are similar to GA, however PGAS languages do not require `put()/get()` style operations to be used when accessing shared memory. Examples of PGAS languages are Unified Parallel C (UPC) [26], Co-Array Fortran [57], Titanium [75], Fortress [68], Chapel [24], and X10 [23].

There are two different approaches for implementing PGAS languages, each of which expose locality to the user in different ways. In the first approach, processes may access any part of the global address space, regardless of where it is mapped (e.g. UPC). However, access to remote partitions can be expected to involve higher overheads. This approach encourages writing parallel programs that access local partitions, but also allows remote partitions to be accessed (at additional costs). In the second approach, processes may only access local partitions; but more processes can be started at remote locations in order to access data at those partitions “locally” (e.g. X10). Conceptually, communication between locations (called “places” in X10), is limited to instructions for creating new processes, and transferring of the results. In contrast to the first approach, the second approach can be said to enforce locality (since processes may only access local partitions).

2.6 Summary

This chapter reviewed SDSM implementation techniques that exploit the semantics of relaxed memory consistency models to reduce data transfer between processes. SDSM systems enable shared memory programming on distributed memory environments.

Other programming models were also introduced that attempt to expose data locality to the programmer (Global Arrays and PGAS languages). These, however, require the programmer to make use of special operations/syntax to properly place data across the distributed memory landscape. More importantly, these models expose data locality to programmers so that programs may be structured to capitalize on the data layout.

From this, it can be argued that a home-based SDSM system, coupled with home-migration capabilities, could achieve similar performance if programmers adhere to the same programming discipline of structuring memory accesses of individual processes into separate localities, as if they were partitioned. This is because data that has

migrated to a process is cheaper to access than data that is “home” at another process, i.e. local access is cheaper than remote access. In addition, home-migration enables data locality to be automatically readjusted to adapt to different program phases at runtime.

This chapter also reviewed MPI-2 RMA features that may be potentially useful for asynchronous one-sided data transfers in SDSM libraries. However, among the restrictions for using MPI-2 RMA features, one particular restriction makes these features unsuitable for SDSM implementation. This is the restriction on concurrent access to private and public copies of a window *even if the accesses occur at disjoint locations in the window*. This is a severe restriction that limits a home-based SDSM’s ability to fetch updates from remote memory or propagate changes to remote memory. Interestingly, the rationale for this restriction is given as:

“Without this constraint, the MPI library will have to track precisely which locations in a window were updated by a put or accumulate call.” ([52], p366.)

In other words, this restriction was included so that a mechanism like twinning-and-diffing, as used by some of the SDSMs, would not be necessary.

Chapter 3

Design of the Danui SDSM Library

The main design goals for the Danui Software Distributed Shared Memory (SDSM) library are portability, and modularity.

Portability in this project refers to the ability to use the system across the different networking options available for cluster computers. This goal is achieved by taking advantage of the portability and availability of highly tuned versions of the Message Passing Interface (MPI) [51]. A similar strategy was used to improve portability of the SCASH [35] SDSM, replacing its underlying communication subsystem called PM [69], with an MPI-based implementation of PM functionality [58]. This MPI version of the PM communication subsystem provided basic message passing and remote memory access (RMA) functionality. Danui takes a different approach by using MPI pervasively throughout the SDSM library. The rationale being that an additional communication abstraction on top of MPI is unnecessary, since the MPI standard is an abstraction in itself; and that portability issues, if any, should be addressed at the MPI level.

Modularity improves the maintainability and extensibility of software systems; qualities that facilitate the study of alternative solutions, making modularity a fitting design goal for a research SDSM. In this regard, the Danui SDSM library is segmented into three modules, Event, Lock, and Heap:

- The *Event* module provides a mechanism to invoke pre-defined functions (event handlers) at remote processes (similar to remote procedure calls). This mechanism is required for the higher level SDSM modules to support operations that are triggered by asynchronously occurring shared memory accesses. The flexibility of this approach allows for functionality such as RMA to be mimicked. For example, an event handler may be created to retrieve data from a remote address location. The Event module also supports sending of events that originate from

event handlers; allowing, for example, complex protocols that involve forwarding of events through arbitrary process topologies.

- The *Lock* module provides inter-process synchronization. Although synchronization has implications for memory consistency, the Lock module focuses on the synchronization aspect of the problem. The objective is to separate the memory consistency concerns from synchronization so that several consistency models can be implemented without having to revisit the lock synchronization problem each time. As such, it is left to higher level interfaces that are exposed to the user to impose memory consistency on top of the basic synchronization provided by the Lock module.
- The *Heap* module provides a globally shared address space and forms the main part of the Danui SDSM. Several sub-problems partition this module into smaller sub-modules. The most significant of which is the Memory Consistency Manager (MCM) that determines how shared memory consistency is maintained by the SDSM. The MCM is organized in a hierarchy of classes that implement several variations of home-based memory consistency models.

Section 3.1 details the design of the Event module; section 3.2 describes how locks are implemented for the SDSM; and section 3.3 shows how the Heap module provides a globally shared address space. A summary of this chapter is given in section 3.5.

3.1 The Event Module

The Event module provides a mechanism for invoking event handler functions at remote processes (similar to a remote procedure call). Event handler functions are user defined, can make use of MPI operations, and can send more events to arbitrary processes; allowing for a wide breadth of inter-process protocols to be implemented.

For example, a remote memory read operation can be implemented using the event module as follows. The operation involves a requesting process and a remote process. At the requesting process, an event is sent to the remote process asking for data from a memory region at that process, and then a `MPI_Recv` call is made to receive that data into a local buffer. At the remote process, the event is received, its associated handler function is invoked, that responds by calling `MPI_Send` to send the required data to the requesting process.

This section describes how Danui defines event handlers, associates them with event IDs, and then delivers event requests to remote processes. The use of a background daemon thread to receive events and how the daemon thread may itself send events is then described.

3.1.1 Defining Event Handlers

Event handlers are functions that are defined outside of the Event module. These functions must conform to one of two different event handler function prototypes.

The first function prototype consists of *source-process* and *param-buffer* parameters. The *source-process* parameter specifies the ID of the process that sent the event, and the *param-buffer* is a buffer for storing parameters in a format that is specific to each event handler.

The second function prototype consists of the same parameters but includes an additional *context-object* parameter. This is an arbitrary object that is associated with an event and is provided to its handler each time that event is received.

No return value is expected from event handler functions.

Handler functions have to be associated with event IDs, and context objects (if any), before events for those handlers may be communicated.

3.1.2 Associating Event IDs to Handlers

Registration is the process of associating event IDs to event handler functions, and context objects if any. This is implemented by the REGISTER operation that adds entries to a lookup table that is indexed by event IDs. The REGISTER operation accepts an *event-handler* and a *context-object*¹ as input. Steps for this operation are as follows:

1. (Generate event ID.) Save the current *event-count* as the generated *event-ID* and increment the event count. (The *event-count* tracks the number of events that have been registered.)
 - (a) Set *event-ID* $\leftarrow$ *event-count*.
 - (b) Increment *event-count* $\leftarrow$ *event-count* + 1.
2. (Add entry.) Store the *event-handler* and *context-object* in *handler* and *context* arrays, using the *event-ID* as index.
 - (a) Set *handler*[*event-ID*] $\leftarrow$ *event-handler*.
 - (b) Set *context*[*event-ID*] $\leftarrow$ *context-object*.

Due to the way event IDs are generated, event registration must occur in the same order at all processes otherwise event handlers would be mapped to different event IDs at different processes. This is guaranteed in the Danui SDSM by making initialization a collective operation involving all processes; as such initialization only completes when all SDSM events have been registered.

¹The *context-object* may be NULL if there is none.

Figure 3.1: Message buffers are divided into the header and parameter partitions. The header partition contains the event ID, source process ID, and the destination process ID; while the parameter partition is used to store event specific parameter data, if any. The *param-size* parameter required by various functions for sending events (see text) refers to the number of bytes, from the start of the parameter partition, to include when sending the message buffer. (The *param-size* can be zero.)

3.1.3 Sending Events

Events are sent by packing the event ID and any event specific parameters into a message buffer that is then sent to the destination process. (Receiving and handling of this message is covered later in section 3.1.4.) As shown in figure 3.1, the message buffer is divided into header and parameter partitions. The header partition holds information about the event ID, the source process ID, and the destination process ID. The parameter partition is a buffer for storing event specific parameter data.

Three alternative functions for sending events are provided: a basic blocking send, a buffered blocking send that saves on memory copy costs, and a non-blocking send:

- **EVENT-SEND(*event-ID*, *param-buffer*, *param-size*, *destination-process-ID*):** This is a blocking send function. Calling this function statically allocates a message buffer in the function's call stack². The input parameters *event-ID* and *destination-process-ID* are recorded in the header partition of the message buffer along with the sender's process ID. This is followed by the contents of *param-buffer* (*param-size* bytes) that is copied into the parameter partition of the message buffer. Lastly, `MPI.Send` is used to send the message buffer to the destination process.
- **EVENT-BUFFER-SEND(*event-ID*, *param-buffer*, *param-size*, *destination-process-ID*):** This is also a blocking send function like `EVENT-SEND`. However, no message

²The Event module specifies the maximum event message buffer size as a constant.

buffer will be statically allocated in the function's call stack. Instead, the *param-buffer* provided must point to a message buffer that has sufficient space before the *param-buffer* address to write header information. This function avoids having to copy the contents of *param-buffer* into a separate message buffer. The final message is sent using `MPI_Send`.

- `EVENT-NONBLOCKING-SEND(event-ID, param-buffer, param-size, destination-process-ID)`: This is a non-blocking send function. Like `EVENT-BUFFER-SEND`, there must be sufficient space before the *param-buffer* address to write header information. The final message is sent using a non-blocking `MPI_Isend` call. This returns an `MPI_Request` handle that is in turn returned to the user to verify, at some later stage, if the non-blocking event communication has completed³.

The amount of data sent by the functions described above is the size of the header plus the size of the event parameters (provided as *param-size*).

3.1.4 Event Daemon Thread

An event daemon thread is created at each SDSM process to receive event messages and execute their respective event handler functions. When an event message is received, its event ID is extracted from the header partition of the message. This ID is used to obtain the registered event handler function and context object (if any). Depending on whether a context object was found, the daemon thread will be able to determine which of the two function prototypes should be used when calling the event handler function.

Calls to event handler functions must never block indefinitely as the daemon thread would otherwise be unable to receive more event messages. For this reason, events sent from handler functions must be sent using non-blocking MPI functions. Yet using the `EVENT-NONBLOCKING-SEND` operation described earlier from an event handler is problematic, because the operation defers the responsibility for ensuring completion to the caller. To address this issue, the event daemon implements a queuing mechanism for sending events originating from handler functions. The tasks of sending these events and tracking their completion can therefore be performed outside of event handler function calls. This queue, called the send-queue, is maintained using a first-in-first-out policy.

The event daemon executes an event-loop that provides the services described above. Figure 3.2 presents the state machine for the event-loop. Activities and transitions pertaining to each state of the event-loop are elaborated as follows:

³The same semantics as `MPI_Isend` apply in that a completed event send operation does not imply that the receiver has received the event.

Figure 3.2: The event-loop that is executed by the daemon thread. Activities pertaining to each of the three states — “Wait,” “Handle Event,” and “Inspect Send-queue” — are elaborated in the text.

- *Wait:* The daemon thread waits for an event to be received, or for the send operation of a queued event to complete.

These receive and send operations are initiated as non-blocking MPI calls. For receiving event messages, the `MPI_Recv_init` persistent receive operation is used since the parameters are the same for every event message received. For sending queued events, the `MPI_Isend` operation is used (persistent send is not possible since the destination process and message buffer varies with each queued event). `MPI_Waitsome` is used to block on the `MPI_Request` handles from these non-blocking calls.

The event-loop enters the “Handle Event” state if an event is received, and enters the “Inspect Send-queue” state if the send operation of a queued event completes. In the latter case, the event related to the completed send is also removed from the head of the send-queue before entering the “Inspect Send-queue” state.

- *Handle Event:* The daemon thread extracts the event ID from the header of the event message and invokes its associated event handler function as described earlier.

As the event handler function may have queued an event (or several events), the event-loop enters the “Inspect Send-queue” state after returning from the handler function.

- *Inspect Send-queue:* The daemon thread checks if there already is a send operation in progress. If a send is in progress, nothing needs to be done. Otherwise, the daemon thread checks if there are queued events that need to be sent, and if so a send is initiated for the first event in the send-queue.

The event-loop returns to the “Wait” state.

In summary, events originating from event handlers are sent using the send-queue to avoid deadlocks. Such deadlocks may occur as the daemon thread is unable to receive events while executing an event handler.

An alternative design would be to create new threads for each event as they arrive, so that if a handler function call blocks, other events may still be received. However event handlers typically access data structures that would require the protection of mutual exclusion. Thus even if new threads were created to handle each event, use of mutual exclusion would result in serialization of the threads. For this reason, a design decision was made to implement the single thread design.

3.2 The Lock Module

The Lock module provides inter-process synchronization that can be used to provide mutual exclusion for SDSM programs. As previously noted, inter-process synchronization has implications for memory consistency. However this module focuses only on the synchronization aspect of the problem, with the intention of separating memory consistency and synchronization concerns so that different solutions for either can be explored with limited impact on the other.

The Lock module supports SET-LOCK, TRY-LOCK, and UNSET-LOCK operations. Briefly, the SET-LOCK operation is used to acquire ownership of a lock, that only one process may own at any point in time. The TRY-LOCK operation *attempts* to acquire a lock and will succeed if no other process is holding the lock at that time; and will fail otherwise⁴. Finally, the UNSET-LOCK operation releases ownership of a lock.

Lock acquisition can be viewed as a process queuing problem where the process at the head of a lock-queue is the owner of the lock. When viewed this way, the steps of the SET-LOCK operation can be described as follows:

1. Append calling process to the end of the lock-queue.
2. Wait until calling process is at the head of the lock-queue.

While TRY-LOCK operation can be described as:

1. If the lock-queue is empty:
 - (a) Insert calling process into the lock-queue.
 - (b) Return SUCCESS.

⁴A boolean value is returned to indicate if a TRY-LOCK attempt has succeeded (true) or failed (false).

Figure 3.3: The lock master synchronizes access to the lock and maintains an internal queue of process IDs. The lock master accepts *Set*, *Try*, and *Unset* messages, and responds with *Acquire* or *Fail*. In the figure, process 2 is at the head of the queue. This indicates that process 2 is currently holding the lock.

2. Else, return UNSUCCESSFUL.

And, the UNSET-LOCK operation would involve just removing the calling process from the head of the lock-queue.

Two alternative lock-queue implementations are provided by the Danui SDSM, each of which is expected to excel over the other depending on the use case. One uses a centralized approach called Master Queue, while the other uses a decentralized approach called Distributed Queue. The implementation details of these alternatives are described below. This is followed by a description of nested lock usage.

3.2.1 The Master Queue

The Master Queue implementation designates a master process for each lock at which the state of the lock-queue is maintained⁵. Operations on a given lock are implemented as protocols that synchronize with the master process of that lock. Figure 3.3 shows the messages that a master process may expect to receive (*Set*, *Try*, and *Unset*) and those that it may send (*Acquire* and *Fail*). Implementation details for each lock operation are as follows:

- The SET-LOCK operation is used to acquire a lock. It involves appending the calling process to the lock-queue and waiting until the process is at the head of the queue. The calling process may perform this task without any communication if it is also the master process for that lock.

However, if it is not the master process, a *Set* message will be sent to the master process, which upon receiving this message will append it to the lock-queue. After sending the *Set* message, the requesting process will wait until it receives an *Acquire* message.

⁵SCASH [35] also uses a centralized master queue to manage locks.

- The TRY-LOCK operation is used to attempt to acquire a lock. It involves appending the calling process to the lock-queue only if the queue is empty. The calling process may perform this task without communication if it is also the master process for that lock. In which case, it will return success (true) if it added itself to the lock-queue, or unsuccessful (false) otherwise.

However, if it is not the master process, then a *Try* message will be sent to the master process, which upon receiving this message will either: 1) append the requesting process to the lock-queue and send an *Acquire* response if the lock is available, or 2) send a *Fail* response if the lock is unavailable. At the requesting process, the TRY-LOCK operation returns successful (true) if *Acquire* was received, or returns unsuccessful (false) if *Fail* was received.

- The UNSET-LOCK operation is used to release a lock. It involves removing the calling process from the head of the lock-queue (it is an error if the calling process is not the head of the lock-queue). The calling process may perform this task without communication if it is also the master process for that lock.

However, if it is not the master process, then an *Unset* message will be sent to the master process, which upon receiving this message will remove the calling process from the head of the lock-queue. No reply message is sent to the calling process for this operation. Instead, an *Acquire* message will be sent to the process at the head of the lock-queue, if any.

3.2.2 The Distributed Queue

The Distributed Queue implementation is based on a description, by Fu and Tzeng [28], of how locks are implemented in TreadMarks [43, 42, 9]. The implementation maintains the state of a lock-queue in a decentralized manner; where each process in the distributed lock-queue maintains a *next-pointer* that indicates the ID of the next process in the queue, if any. For example, a queue form by next-pointers at processes P_1 , P_2 , and P_3 , in that order, may be visualized as follows:

$$\text{next-pointer: } (P_1) \rightarrow P_2 \rightarrow P_3$$

(Parenthesis are used to indicate if the lock is being used by the process at the head of the queue. In the above “ (P_1) ” indicates that process P_1 is using the lock. Conversely “ P_1 ” with no parenthesis indicates that P_1 is not using the lock.)

The main benefit of the Distributed Queue approach is realized when lock ownership is transferred from a releasing process to the next process in the lock-queue, since only one message between the pair is required. In comparison, the Master Queue

implementation described previously requires two messages — one from the releasing process to the lock master, and another from the master to the acquiring process.

Appending a process to a distributed queue is more complicated because it requires the next-pointer of the last process in the queue to be updated; and which process that is is not immediately apparent. In Danui, this problem is solved in a manner described by Fu and Tzeng [28]. A master process, known to all processes, is appointed to maintain an additional pointer field called the *tail-pointer*. This pointer stores the ID of the last process in the lock-queue, that is correspondingly referred to as the tail-process. Extending the lock-queue example described above, assume that process P_0 is the master process where a tail-pointer pointing to P_3 (the tail-process) is maintained as follows:

$$\begin{array}{ll} \text{tail-pointer:} & P_0 \rightarrow P_3 \\ \text{next-pointer:} & (P_1) \rightarrow P_2 \rightarrow P_3 \end{array}$$

Thus, to append a process to the distributed queue, a request can be sent to the master process that can then be forwarded to the tail-process, where the requesting process *may be appended to the queue, or not*. Whether the requesting process is appended to the queue depends on whether the request pertains to a SET-LOCK or a TRY-LOCK operation; and for the latter operation, also if it succeeds.

Knowing whether a requesting process is appended to the lock-queue is important for the master process of a lock as it is responsible for updating the tail-pointer when the last process in the lock-queue is changed. If the routed message pertains to a SET-LOCK request, then the tail-pointer is updated to the process ID of the requesting process; as regardless of the availability of the lock that process would certainly be the next tail-process when the routed message arrives at the present tail-process. However, if the routed message pertains to a TRY-LOCK request, then the master process cannot know for certain if the request would succeed; as the information required to decide the outcome of a TRY-LOCK request is only available at the tail-process⁶. Consequently, it is incorrect for the master process to assume that the requesting process would be the next tail-process.

Observe that a TRY-LOCK operation can only succeed if the queue is empty. The empty queue situation is illustrated as follows:

$$\begin{array}{ll} \text{tail-pointer:} & P_0 \rightarrow P_3 \\ \text{next-pointer:} & P_3 \end{array}$$

(Note that there are no parenthesis around P_3 indicating that although the lock is at

⁶The information required to decide the outcome of a TRY-LOCK request is available only at the tail-process. TRY-LOCK requests succeed if there are no processes using the lock and this situation is indicated by an empty lock-queue. The queue is empty when, the *tail-process* is not waiting for and not using the lock.

process P_3 , it is not being used.) Thus, if a TRY-LOCK request message originating from P_1 arrived at P_3 , a small trick can be used to give the lock to P_1 , while keeping P_3 as the last process in the queue. That is rather than appending, the requesting process P_1 is instead *inserted* at the front of the lock-queue. In which case, the queue state would change to the following:

$$\begin{array}{ll} \text{tail-pointer:} & P_0 \rightarrow P_3 \\ \text{next-pointer:} & (P_1) \rightarrow P_3 \end{array}$$

where 1) P_1 now has the lock and is using it, 2) P_3 is the last process of the lock-queue, and 3) the tail-pointer at the master process P_0 remains accurate.

Conceptually, a lock is a token that is passed among processes; and at a process, the lock may or may not be used. That is, the location of a lock and how that lock is being used are separate concepts. Another concept is that processes not waiting for a lock may also be in a lock-queue. This enables the TRY-LOCK operation to be supported as detailed earlier. These concepts mean that use of next and tail pointers alone are insufficient to fully model the complex interactions between processes and a lock. As such, a state machine is used to model a process' relationship with a lock. (One state machine is required for each process/lock pair.)

This relationship is shown in figure 3.4. There are a total of eight states, and these are characterized by the following hierarchy:

1. The process has the token, and

(a) is using the lock, and

- i. there is a next-process: (LOCKED-N state.)
- ii. there is no next-process: (LOCKED state.)

(b) is not using the lock: (AVAIL state.)

2. The process does not have the token, and

(a) wants to use the lock, and

- i. there is a next-process: (WAITING-N state.)
- ii. there is no next-process: (WAITING state.)

(b) does not want to use the lock but is in the queue, and

- i. there is a next-process: (QUEUED-N state.)
- ii. there is no next-process: (QUEUED state.)

(c) does not intend to use the lock: (NIL state.)

3.2.2.1 Set-Lock Operation

The SET-LOCK operation appends the calling process to the lock queue and blocks until the lock token is acquired. Prior to calling the SET-LOCK operation, a lock may be in one of the following four states: AVAIL, NIL, QUEUED, or QUEUED-N.

- When in the AVAIL state, the lock will be transitioned into the LOCKED state; no communication is required for this case.
- When in the NIL state, a *Set* message will be sent to the tail process via the master process.

When this message is received by the master process, and it is not also the tail process, the message will be forwarded to the tail process. In either case, the tail-pointer (at the master process) will be updated to the ID of the requesting process, since this process will be the next tail process.

At the tail process, receiving the *Set* message will result in either of the following transitions depending on the state of the lock: QUEUED to QUEUED-N, WAITING to WAITING-N, LOCKED to LOCKED-N, or AVAIL to NIL. Note that it is an error to receive the *Set* message when already in one of the “-N” states, as by definition a tail process is at the *end* of the queue and should not have any next process.

- When in the QUEUED or QUEUED-N state, the lock will be transitioned to the WAITING or WAITING-N states respectively. This is because the process is already in the lock queue and will eventually receive an *Acquire* message. No communication is required for this case.

Figure 3.5 illustrates how the SET-LOCK operation is performed. In the example, P0 is the lock master, and as shown in panel (a) the lock queue initially contains processes P1 and P2, with P1 using the lock. In panel (b), process P3 uses the SET-LOCK operation to make a lock request. The *Set* message is sent to the tail process P2 via the lock master P0. After forwarding the message, P0 updates the tail process ID to P3; and when P2 receives the forwarded message, it updates the next process ID to P3. Effectively placing P3 in the lock queue; where it will remain in the WAITING state until it receives an *Acquire* message from P2.

3.2.2.2 Try-Lock Operation

The TRY-LOCK operation attempts to acquire a lock and, depending on the availability of the lock, may succeed or fail. The operation involves sending a *Try* message to the tail process (via the master) and waiting for either a *Try-Success* or *Try-Failed* response

(a) Initial lock queue

(b) P3 does Set-Lock

Figure 3.5: Distributed Queue SET-LOCK operation. Panel (a) describes the initial state of a lock queue consisting of processes P1 and P2. Process P0 is the lock master and P3 is initially not involved with the lock. The state of the lock at each process is shown. In the first part of panel (b), P3 invokes SET-LOCK, sending the *Set* message to the lock master (P0) which forwards the request to the tail process (P2). In the second part of panel (b), the tail pointer at the lock master and the next pointer at P2 are updated to P3. The updated lock states are indicated; P2 transitions to WAITING and P3 to WAITING.

message. This is similar to the SET-LOCK operation with the only difference being that *Try-Success* and *Try-Failed* messages are used in place of an *Acquire* message.

Prior to calling the TRY-LOCK operation on a lock, the lock may be in one of the following four states: AVAIL, NIL, QUEUED, or QUEUED-N.

- When in the AVAIL state, the lock will be transitioned into the LOCKED state; no communication is required for this case. The operation returns “successful” (true).
- When in the QUEUED or QUEUED-N states, no transitions are performed and no communications are required. The operation returns “unsuccessful” (false) as being in the queue implies that another process is using the lock.
- When in the NIL state, a *Try* message will be sent to the tail process via the master process, forwarding if necessary as with the SET-LOCK operation. Unlike the SET-LOCK operation, however, the tail-pointer at the master will not be updated to the ID of the requesting process.

At the tail process, receiving the *Try* message triggers the following actions:

1. If the lock is in the AVAIL state, then the lock is transitioned into the QUEUED state and the requesting process is sent a *Try-Success* message.
2. For all other states, the requesting process is sent a *Try-Failed* message.

If a *Try-Success* message is received at the requesting process, the ID of the next process is set to the sender of the message (i.e., the tail process) and the lock is transitioned into the LOCKED-N state. This indicates that the lock is being used by the requesting process and that there exist a next process in the lock queue. The operation returns “successful” (true).

Otherwise, if a *Try-Failed* message is received, the operation returns “unsuccessful” (false).

Figure 3.6 shows examples of TRY-LOCK attempts that are: a) unsuccessful and b) successful. In both examples, process P0 is the master, process P2 is the tail, and P1 is the process that initiates the TRY-LOCK attempt. These examples are elaborated below:

- *Panel (a), unsuccessful attempt:* The lock is in the LOCKED state at the tail process P2, indicating that the lock is being used by P2. Thus, the *Try-Failed* message is sent to P1 in response.

(a) Try-Lock Failed

(b) Try-Lock Success

Figure 3.6: Distributed Queue TRY-LOCK operation. (a) A failed attempt involving where three messages are sent but no states are changed. (b) A successful attempt where the lock is transferred from P2 to P1. P2 enters the QUEUED state and the tail pointer at P0 (the lock master) continues to point at P2. P1 enters the LOCKED-N state with P2 as its next process.

- *Panel (b), successful attempt:* The lock is in the AVAIL state at the tail process P2, indicating that the lock is not being used by any process. Thus, the *Try-Success* message is sent to P1 in response.

The lock is transitioned into the QUEUED state at P2, as it is now part of the lock queue (tail-pointer at master process P0 continues to point at P2), but is not waiting to use the lock.

The lock is transitioned into the LOCKED-N state at P1 when the *Try-Success* message is received and the next process pointer is set to P2.

The overall effect on the lock queue is that P1 has been inserted at the front of the queue and the tail process is unchanged.

As previously noted, the QUEUED state is special as it allows a process to be part of the lock queue while not actually intending to use the lock. Panel (a) of figure 3.7 illustrates what happens when another process P3 performs the SET-LOCK operation

(a) P3 does Set-Lock

(b) P1 does Unset-Lock

Figure 3.7: Demonstrating the role of the QUEUED state. (a) P3 joins the lock queue using the SET-LOCK operation. (b) P1 releases and transfers the lock to P2, which forwards the *Acquire* message on to P3. (P1 does not know about P3, only that the next process in the queue is P2.)

while the tail process is in the QUEUED state. As described earlier, the *Set* message is sent to the tail process P2 via the lock master. When this is received at P2, the next pointer is set to P3 and the lock state transitioned into QUEUED-N. Also, the tail pointer at the master is changed to point at P3. In panel (b), the lock is released by P1 and the *Acquire* message is sent to P2. This is then forwarded by P2 on to P3.

3.2.2.3 Unset-Lock Operation

The UNSET-LOCK operation releases the lock token to the next process in the queue. Prior to calling the UNSET-LOCK operation on a lock, the lock must be in either the LOCKED or LOCKED-N states.

- When in the LOCKED state, the lock will be transitioned into the AVAIL state; no communication is required as there are no processes waiting for the lock.
- When in the LOCKED-N state, the lock will be transitioned into the NIL state and an *Acquire* message will be sent to the next process in the queue.

(a) Unset-Lock from Locked-N state

(b) Unset-Lock from Locked state

Figure 3.8: Distributed Queue UNSET-LOCK operation. (a) The lock is transferred from P1 to P2 by using the *Acquire* message. P1 transitions into the NIL state and P2 into the LOCKED-N state. (b) P1 is also the tail process and therefore has no next process to which the lock should be transferred; hence, no communication is required. The lock at P1 transitions to the AVAIL state (the lock token is coloured white to illustrate this).

Figure 3.8 illustrates examples of UNSET-LOCK operations when a lock is in the LOCKED-N and LOCKED states:

- *Panel (a), from LOCKED-N state:* P1 is initially in the LOCKED-N state and the next process P2 is in the WAITING-N state. The lock token, in this case, is released to P2 by sending an *Acquire* message to P2. Following this, the lock will be in the NIL state at P1, and in the LOCKED-N state at P2.
- *Panel (b), from LOCKED state:* P1 is initially in the LOCKED state; that is, there are no processes waiting to use the lock after P1. Thus the UNSET-LOCK operation is performed by transitioning the lock at P1 into the AVAIL state.

3.2.3 Nested Lock Usage

Lock usage is described as “nested” when the SET-LOCK or TRY-LOCK operations are used to acquire a lock which the calling process is already using; and that this process later performs the same number of UNSET-LOCK operations to release the lock. Support for nested lock usage is helpful for applications with multiple subsystems that cross-call functions in an unpredictable fashion. For example, a function a_1 in subsystem A acquires a lock on A , calls a function b_1 in subsystem B , which calls another function a_2 in subsystem A to obtain necessary information and a_2 *also acquires a lock on A* .

Nested usage is supported by associating each lock at a process to a nest-count that is zero when the lock is not being used, and non-zero if the lock is being used by the process. Each SET-LOCK operation on a lock will inspect the nest-count and actual lock operations are only performed if the nest-count is zero. In any case (zero or non-zero), the nest-count will be incremented. Thus, the nest-count of a lock reflects the number of times SET-LOCK (and/or successful TRY-LOCK) operations have completed at that process which have yet to be released via UNSET-LOCK. (TRY-LOCK operations always return “successful” when the nest-count is non-zero.)

UNSET-LOCK operations on a lock begin by decrementing the nest-count and only perform the actual lock operation when the nest-count has been decremented to zero.

3.3 The Heap Module

The Heap module implements a shared memory heap from which processes may request shared memory regions. The shared memory heap is a contiguous region of the process address space that starts at the same address and is of the same size at all processes. This shared address space is partitioned into pages, which defines the heap’s coherence granularity. Coherence mechanisms are provided that are based on an invalidation protocol.

Memory consistency of the shared memory heap is maintained according to the OpenMP [59] memory model. In the OpenMP memory model, global and temporary views of shared memory are defined. While there is one global view for *all* threads, there is one temporary view for *each* thread. All memory accesses by threads perform with respect to their temporary views. This means that the temporary view at a thread may differ from the global view and by extension also differ from the temporary views at other threads. The temporary view at a thread is synchronized with the global view by performing a flush operation. The requirements for correct flush operation are as follows:

1. All modifications that have been made to the temporary view of the calling thread

must be visible at the global view by the end of the flush operation.

2. All modifications that have been made visible to the global view, including those from other threads, must be visible at the temporary view of the calling thread by the end of the flush operation.

It is worth noting that a shared memory system with only a global view would still satisfy the requirements listed above. This is because the requirements only apply when a flush operation completes. In other words, the requirements listed above define *late bounds* for when modifications have to be globally visible.

It is also important to note that when we say a change is visible at some location, it does not mean that the change is at that location. Rather, it means that the next process to access that location will witness the change. To understand this difference, consider the use of invalidation based protocols where making a change visible can mean telling remote processes *about a change* (but not the actual change), so that relevant precautions can be taken to ensure that the change would be retrieved in future if there is a need to do so.

A sub-module called Memory Consistency Manager (MCM) is used to implement the memory model described above. This sub-module provides three operations:

- **FAULT(*address*)**: This operation is executed when a segmentation fault signal is caught for memory access to the specified address. The faulted memory access is repeated after this operation completes.
- **FLUSH**: This operation is executed explicitly by the programmer. It provides the same guarantees as the OpenMP flush described above.
- **BARRIER**: This operation is executed explicitly by the programmer. The *effect* of this operation should be the same as performing the following sequence: FLUSH all processes, synchronize all SDSM processes, and FLUSH all processes.

The module is organized in a hierarchy of MCM classes as illustrated in figure 3.9. The top tier of the hierarchy consists of the “MCM” class. This class represents an abstraction layer that defines the FAULT, FLUSH, and BARRIER operations as *abstract* functions.

The second tier of the hierarchy consists of the “Home-based MCM” class. This class extends from the MCM class and provides implementations of the FAULT and FLUSH functions. The BARRIER function remains undefined at this tier of the hierarchy.

The third tier of the hierarchy consists of the “Invalidate All MCM,” “Selective Invalidation MCM,” “Write-set MCM,” and “Scalable MCM” classes. These classes extend from the Home-based MCM class and provide different implementations of the BARRIER function. These differences are summarized as follows:

Figure 3.9: MCM class hierarchy. The MCM class defines an abstract interface consisting of the FAULT, FLUSH, and BARRIER operations. The Home-based MCM class implements FLUSH and FAULT. The classes in the shaded box are all children of the Home-based MCM class and implement different BARRIER operations. The BARRIER functions implemented by the Write-set MCM and Scalable MCM classes provide frameworks within which migration policies can be effected. These policies are provided by their child classes.

- *Invalidate All MCM*: This class provides a BARRIER implementation that invalidates all remote pages. This implementation is characterized by low bookkeeping overhead but may have higher than necessary coherency overheads as it may invalidate more pages than necessary.
- *Selective Invalidation MCM*: This class provides a BARRIER implementation that invalidates only remote pages that have been modified by another process. Coherency costs may be lower but it additional bookkeeping overheads are required to track memory writes for local pages (Invalidate All MCM did not require this).
- *Write-set MCM*: This class provides a BARRIER implementation that is a framework for home-migration. This framework can be customized by various policies, three types have been implemented and are provided by the three child classes of the Write-set MCM class.
- *Scalable MCM*: This class provides a BARRIER implementation that is also a framework for home-migration that can be customized by various policies. It differs from the Write-set MCM in that communication and computation are more scalable.

The following subsections describe the Home-based MCM and the four MCMs that implement the BARRIER function.

3.3.1 The Home-based MCM Class

The Home-based MCM class implements the MCM class using a home-based approach. This approach was selected for Danui as the intention was to support an integrated shared memory and message passing programming environment. Since message passing is most effective when data dependencies are regular and for large contiguous regions [44], and because these are also favourable access patterns for home-migration, it can be assumed that for these types of applications, memory writes in a home-based SDSM would primarily be to local pages. Given that modifying local pages does not incur twinning-and-diffing coherency costs, the performance of home-based SDSMs can be expected to be better than homeless SDSMs for these types of applications.

The Home-based MCM class partitions a shared heap into pages, each of which has the following properties:

- *Home*: Each page has a home process at which the global view for that page is maintained. The copy of a page at its home process is also referred to as the *master copy*.
- *Locality*: Each process considers a page to be a *local page* if the home for the page is at that process. Otherwise, the page is referred to as a *remote page*.

Table 3.1: Page state names and abbreviations.

State Name	Abbreviation
Local Unmapped	LUM
Local Read-only	LRO
Local Read-write	LRW
Remote Unmapped	RUM
Remote Read-only	RRO
Remote Read-write	RRW

- *Memory protection:* This is used to trap access to pages and may be set to unmapped (invalid), read-only, or read-write.
- *Page state:* The page state is derived by combining locality and memory protection properties. This gives rise to the six possible states and abbreviations listed in table 3.1. Modeling page states in this combined form allows state machines for page states to model home-migration as transitions between local and remote states.

In addition to MCM operations (FAULT and FLUSH), the Home-based MCM class also implements coherency operations for home-based SDSM systems. Details for home-based coherency operations, and MCM operations are described in the following subsections.

3.3.1.1 Home-based Coherency Operations

The Home-based MCM implements coherency mechanisms for validating a page, tracking modifications to pages, and for sending modifications to a page to its home process. Three operations are provided by the Home-based MCM: PAGE-FETCH, MAKE-TWIN, and SEND-DIFF.

PAGE-FETCH(*page-number*): This operation fetches the specified page from its home process. Steps for this operation are as follows:

1. Set $M \leftarrow$ an alternative memory map for the specified page is obtained so that data may be received without triggering segmentation faults.
2. Issue a non-blocking `MPI_Irecv` to receive page data into M from the home process of the specified page.

3. Send a “Page Fetch” event to the home process.
 - When received by the Event daemon at the home process, an event handler is executed that issues the matching `MPI_Rsend`⁷ to transfer page data to the requesting process.
4. Wait for the `MPI_Irecv` communication to complete.
5. Free alternative memory map *M*.

`MAKE-TWIN(page-number)`: This operation makes a copy of the specified page that is called the twin of that page. This operation is performed before modifications are made to a remote page.

`SEND-DIFF(page-number)`: This operation compares the twin and current temporary view of the specified page to encode a diff that is then sent to the home process for that page. Steps for this operation are as follows:

1. Encode modifications into a *diff* [13, 22]. The format used for representing a diff is a buffer containing a list of modified regions. The front of the buffer is used to store an integer that indicates the number of regions in the list. This is followed immediately by the encoded regions. Each region consists of a header (indicating the page offset at which the region begins and the length of the region) and modified data. Using this format, a diff can be created by comparing a modified page with its twin in one pass. The twin is destroyed after diffing.
2. Issue a non-blocking `MPI_Isend` to send the diff to the home process.
3. Send a “Send Diff” event to the home process.
 - When received by the Event daemon at the home process, an event handler is executed that issues the matching `MPI_Recv` to receive the diff into a buffer. The diff is then applied to the master copy by copying the modified region data to their respective locations in the master copy.
 - Some additional actions may be performed at the home process depending on whether the reason for sending the diff was for a `FLUSH` or `BARRIER` operation. These will be mentioned later when `FLUSH` and `BARRIER` operations are described.

4. Wait for the `MPI_Isend` communication to complete.

⁷`MPI_Rsend`, “ready send,” can be used when the user can guarantee that the matching receive call has been posted.

3.3.1.2 MCM Operations

The Home-based MCM implements two of the three abstract MCM operations; namely **FAULT** and **FLUSH**.

FAULT(*address*): The **FAULT** operation is called when a segmentation fault signal is caught for memory access to the specified *address*. This operation is implemented by mapping the input address to a page number and, based on the page state, calling a correspondingly named operation that accepts a page number as input. For example, a fault on a page in the LRO state would be resolved by calling **LRO-FAULT**. These state specific fault operations are abstract at this level of the class hierarchy and would be implemented by child MCM classes if required.⁸ This separation allows MCM specific actions to be implemented.

FLUSH: This operation is implemented by sending all modifications to their respective page homes and then invalidating all copies of remote pages. A counter *expected-acks* is used to record the number of synchronizing acknowledgement messages expected. This counter is zero at the start of the **FLUSH** operation and should be zero again when it completes. The steps for this operation are as follows:

1. For each RRW page:
 - (a) Call **SEND-DIFF** to send modifications made to the page to its home process.
 - (b) Increment the *expected-acks* counter.
 - (c) Invalidate the page by transitioning it into the RUM state.

When a diff is received, its encoded modifications are applied to the master copy of the page. In addition, a “Diff Ack” event will be sent back to the flushing process. The handler for this event decrements the *expected-acks* counter.

2. For each RRO page, invalidate the page by transitioning it into the RUM state.
3. Wait for acknowledgement messages for each diff sent so that the operation only completes when it is certain that all diffs have been received and applied at their home processes; that is, they are globally visible. This involves waiting for the *expected-acks* counter to be zero.

⁸An MCM only needs to implement the state specific fault operations for the states which it uses; e.g., the Invalidate All MCM does not use the LRO state and consequently has no need for a LRO-FAULT operation.

3.3.2 The Invalidate All MCM Class

The Invalidate All MCM class provides an implementation of the BARRIER operation that invalidates all remote pages.

The page states used by this MCM are: LRW, RUM, RRO, and RRW. That is, local pages are always read-writable. This MCM is only concerned with monitoring access to remote pages. Of these states, only the RUM and RRO states result in page faults. Their corresponding state-specific fault operations are described below:

- RUM-FAULT(*page-number*): This operation uses the PAGE-FETCH operation implemented by the Home-based MCM to fetch an updated copy of the specified page. The page is then transitioned into the RRO state.
- RRO-FAULT(*page-number*): This operation uses the MAKE-TWIN operation implemented by the Home-based MCM on the specified page. The page is then transitioned into the RRW state.

The BARRIER operation is implemented in a similar fashion to the FLUSH operation except that it involves all processes and, as such, individual acknowledgement messages (as used by FLUSH operations) can be avoided. To achieve this, the BARRIER operation uses a combination of `MPI_Allreduce` and `MPI_Barrier` functions to collectively synchronize delivery of diffs. Steps for the BARRIER operation are as follows:

1. Initialize data structures used for synchronizing diff delivery.
 - (a) Zero *nsends* array. This array is used to record the number of diffs sent to each remote process; e.g., *nsends*[*u*] will reflect the number of diffs that have been sent to process *u*.
 - (b) Set *nrecv* $\leftarrow$ 0. This counts the number of diffs received.

Before the barrier can complete, all diffs that have been sent must have been received and applied to their master copies. The *nsends* array and *nrecv* counter are used later to ensure that all diffs have been applied.

2. Synchronize processes; `MPI_Barrier`.
3. Diff and invalidate all remote pages that have been modified. For each RRW page:
 - (a) Call SEND-DIFF to send modifications made to the page to its home process.
 - (b) Increment *nsends*[*home-process*].
 - (c) Invalidate the page by transitioning it into the RUM state.

Figure 3.10: Page state machine for the Invalidate All MCM. Local pages are always read-writable and remain in the LRW state.

When a diff is received, its encoded modifications are applied to the master copy of the page. Since the diff originated from a BARRIER operation, the *nrecv* counter that was initialized to zero at the beginning of the barrier is incremented.

4. Invalidate all RRO pages.
5. Obtain the number of diffs this process should expect to receive. `MPI_Allreduce` is used to sum the *nsends* array at each process into another array, *expected-recvs*. Thus, for each process *i*, the value in *expected-recvs[i]* indicates the total number of diffs it should receive.
6. Wait until the *nrecv* counter, which is incremented by the Event daemon each time a diff is received, is equal to *expected-recvs[this-process]*.
7. Synchronize processes; `MPI_Barrier`. Completing this synchronization implies that all processes have received all their diffs.

Figure 3.10 shows the page state machine that describes how the Invalidate All MCM manages each page. The state machine is driven by the RUM-FAULT, RRO-FAULT, FLUSH, and BARRIER operations. In general, fault operations lead to relaxation of memory protection, while FLUSH and BARRIER operations lead to tightening of memory protection. Local pages for this MCM always remain in the LRW state.

3.3.3 The Selective Invalidation MCM Class

The Selective Invalidation MCM class provides an implementation of the BARRIER operation that selectively invalidates only the remote pages that have been modified by *another process*. To understand why this is consistent with the memory model, observe that the purpose of invalidating a page is to ensure that all globally visible modifications that are not evident to the local temporary view become visible when

the page is next accessed. Thus, a remote page only has to be invalidated when there are modifications from other processes that have been made globally visible.

This barrier requires processes to know about modifications made by other processes so that each process can make decisions about invalidation. This in turn requires each process to maintain a set of pages which it has modified so that it may share this information during barrier operations. As mentioned in the MCM class hierarchy listed earlier, each process uses a bit-vector called a write-set to record the pages that have been modified by the process when using the Selective Invalidation MCM. All bits in write-sets are initially zero, and are only set to 1 when a page is modified. This includes both local and remote pages. Thus, in addition to the LRW, RUM, RRO, and RRW states (as described for the Invalidate All MCM class previously), the Selective Invalidation MCM class also uses the LRO state; which allows write faults to occur on local pages. State-specific fault operations for the LRO, RUM, and RRO states are described below:

- **LRO-FAULT(*page-number*)**: The appropriate bit of the write-set is set to 1 and the page is transitioned into the LRW state.
- **RUM-FAULT(*page-number*)**: The PAGE-FETCH operation implemented by the Home-based MCM is used to fetch an updated copy of the specified page. The page is then transitioned into the RRO state.
- **RRO-FAULT(*page-number*)**: The MAKE-TWIN operation implemented by the Home-based MCM is used on the specified page. The page is then transitioned into the RRW state and the appropriate bit of the write-set is set to 1.

It is important to note that a 1-bit in a write-set indicates that some page has been modified, and not that the page has modifications that need to be communicated to the home process. Thus, although FLUSH operations invalidate remote pages, the write-set bits for invalidated pages are not set to 0 during FLUSH operations. Leaving the bits set to 1 ensures that other processes with copies of the page would invalidate those copies at the subsequent BARRIER operation.

The Selective Invalidation MCM BARRIER implementation uses the same mechanism for synchronizing diff delivery as was described for the Invalidate All MCM previously. This involves initialization of the *nsends* array and *nrecv* counter at the start of the operation, an `MPI_Allreduce`, a waiting step, and an `MPI_Barrier` at the end. Differences between these BARRIER implementations lie between the synchronization steps. These differences are listed below:

- A step for combining write-sets from all processes. Write-sets are combined using `MPI_Allgather` such that a *combined-set* is a concatenation of write-sets from all processes. Note that `MPI_Allgather` is used, so each process has a combined-set.

- A step for processing the combined-set to determine if a page should be invalidated.
- The order in which remote pages are invalidated. This BARRIER implementation invalidates RRO pages before RRW pages as RRW pages that are not invalidated transition into the RRO state rather than RUM.

Steps for the Selective Invalidation MCM BARRIER are as follows:

1. Initialize data structures used for synchronizing diff delivery. (Same as Invalidate All MCM BARRIER.)
 - (a) Zero *nsends* array. This array is used to record the number of diffs sent to each remote process; e.g., *nsends*[*u*] will reflect the number of diffs that have been sent to process *u*.
 - (b) Set *nrecv* $\leftarrow$ 0. This counts the number of diffs received.
2. Synchronize processes; `MPI_Barrier`.
3. Combine individual write-sets into combined-sets at each process using `MPI_Allgather`. (A combined-set is a concatenation of write-sets from all processes.)
4. Process the combined-set. An array of counters *changed* is used to count the number of modifications to a page. The array is initially zero and counting is capped at 2, because the operation only needs to differentiate between: *changed*[*p*] = 0 — no modifications were made to page *p*, *changed*[*p*] = 1 — one process modified page *p*, and *changed*[*p*] = 2 — more than one process modified page *p*. Thus, for each process and page, *u* and *p* respectively:
 - (a) If *changed*[*p*] < 2, and process *u* has modified *p*, then: Set *changed*[*p*] $\leftarrow$ *changed*[*p*] + 1
5. Invalidate RRO pages that have been modified. For each RRO page *p*:
 - (a) Invalidate page *p* if *changed*[*p*] > 0. Otherwise, the page is left in the RRO state.
6. Invalidate RRW pages that have been modified by more than one process. For each RRW page *p*:
 - (a) Call SEND-DIFF to send modifications made to the page to its home process.
 - (b) Increment *nsends*[*home-process*].
 - (c) If *changed*[*p*] = 2, invalidate page *p* by transitioning into the RUM state. Otherwise, keep the page valid by transitioning into the RRO state.

Figure 3.11: Page state machine for the Selective Invalidation MCM. Barriers are labeled as, Barrier-1, or Barrier-2 depending on whether one, or more-than-one process have modified the page respectively. (Barrier-1,2 is used when a transition applies to both the Barrier-1 and Barrier-2 cases.)

7. Transition all LRW pages back to the LRO state so that write access may be detected again after the BARRIER operation.
8. Zero write-set.
9. Obtain the number of diffs this process should expect to receive. `MPI.Allreduce` is used to sum the `nsends` array at each process into another array, `expected-recvs`. Thus, for each process i , the value in `expected-recvs[i]` indicates the total number of diffs it should receive.
10. Wait until the `nrecv` counter, which is incremented by the Event daemon each time a diff is received, to be equal to `expected-recvs[this-process]`.
11. Synchronize processes; `MPI.Barrier`. Completing this synchronization implies that all processes have received all their diffs.

Figure 3.11 shows the page state machine for the Selective Invalidation MCM. With respect to each page, BARRIER operations are labeled as Barrier-1, or Barrier-2. These labels correspond to the values stored in the `changed` array, which range from 0 to 2. (There is no Barrier-0 as no transitions arise when a page has not been modified.)

Notice that a RRW page is transitioned into the RRO state for the Barrier-1 case, which occurs when only one process has modified the page. This is correct because the process at which the page is in the RRW state is the only process to have modified that page. Given this, only one diff is received for the page at the home process and the master copy is identical to the modified copy after that diff has been applied. Thus, there is no need to invalidate the modified copy.

The Write-set MCM class, described next, takes this insight further by migrating the page home to the process that modified the page. This achieves better performance as no diff needs to be sent back to the original home process.

3.3.4 The Write-set MCM Class

The Write-set MCM class provides an implementation of a BARRIER operation that supports home-migration. Similar to the Selective Invalidation MCM, this MCM uses write-sets to record and share the set of pages that have been modified at each process. However, the Write-set MCM also generalizes the treatment of this information into strategies that decide if a page should be invalidated, or migrated. Each strategy is implemented as an IMMIGRATION operation that makes decisions based on the write-set information from all processes. The IMMIGRATION operation is defined as follows:

IMMIGRATION(combined-set): This operation processes a combined-set (concatenation of individual write-sets from each process) into decisions, for all pages, that are represented by a *modified-set* and a *home-map*. The *modified-set* is a set of pages that have been modified and the *home-map* is an array that maps pages in the modified set to home processes. Home-migration occurs when the home-map maps a modified page to a different process from its present home process.

The steps for the Write-set MCM BARRIER are described below. Labels “Barrier-L” and “Barrier-R” denote different barrier cases with respect to *specific pages*.⁹

1. Initialize structures for synchronizing diff delivery.
2. Use MPI_Allgather to combine *write-sets* into a *combined-set*.
3. *modified-set*, *home-map* $\leftarrow$ IMMIGRATION(*combined-set*).
4. For each modified page *p* in the *modified-set*:
 - (a) Let $u \leftarrow \text{home-map}[p]$
 - (b) Update the SDSM’s page table to reflect that *u* is now the home process for page *p*.
 - (c) If *u* is this process: “Barrier-L”
 - i. Transition page *p* into the LRO state.
 - ii. Delete twin data if there exists any.
 - (d) Else: “Barrier-R”

⁹A barrier may be labeled Barrier-L with respect to a page, but be labeled Barrier-R for another page. These labels will be used to describe transitions for a page state machine later.

Figure 3.12: Page state machine for the Write-set MCM.

- i. If there is a twin:
 - A. Call SEND-DIFF to send modifications made to the page to process u .
 - B. Increment $nsends[u]$.
 - ii. Invalidate the page by transitioning it into the RUM state.
5. Remainder of the BARRIER operation; waiting for diffs to be delivered. (As before.)

Figure 3.12 shows the page state machine for the Write-set MCM class. This page state machine differs from the previous page state machines in that there are transitions between local and remote states; indicating home-migration. Barriers are classified as Barrier-L or Barrier-R for local and remote transitions respectively. Barrier-L barriers transition pages into the LRO state, while Barrier-R barriers transition pages into the RUM state.

Three different IMMIGRATION operations have been implemented by separate MCM classes that extend from Write-set MCM. Namely, the Write-set Single-writer MCM, Write-set Lowest-ID MCM, and Write-set N-Homes MCM classes.

- The Write-set Single-writer MCM class provides an IMMIGRATION operation that performs home-migration in single-writer scenarios, which occur when pages are modified by only one process.
- The Write-set Lowest-ID MCM and Write-set N-Homes MCM classes provide IMMIGRATION operations that performs home-migration even for multiple-writer scenarios. They differ only in how the new home process is selected.

In the Lowest-ID case, the writer process with the lowest process ID is selected to be the new home-process; and in the N-Homes case, the writer process with the least number of local pages is selected to be the new home-process. The objective of the N-Homes approach is to distributed page homes evenly to avoid contention for the attention of one or a few processes. (Contention is undesirable as daemon threads in the Event module can only service one handler at a time.)

For both implementations, no migration is performed if one of the writers is also the home process, or if a page was flushed before the barrier.

All implementations of the IMMIGRATION operation above perform iterations over a combined-set. This means that every combination of process ID and page number has to be inspected. This is true even if none or only a couple of the pages were modified. These scalability issues are addressed by the Scalable MCM class that is described next.

3.3.5 The Scalable MCM Class

The Scalable MCM class provides an implementation of a BARRIER operation that supports home-migration. Unlike the Write-set MCM described previously, the Scalable MCM does not rely on bit-vector write-sets. Instead, computation and communication steps that are designed to scale with the number of modified pages and number of processes are used to generate and disseminate migration/invalidation decisions.

The main steps of the Scalable MCM BARRIER operation that differentiates it from the Write-set MCM are outlined in figure 3.13 and detailed below:

1. *Pack Write-notices:* For each RRW page, a write-notice has to be sent to its home process. To facilitate communication of these notices (essentially page numbers), the page numbers of modified pages are packed into a message buffer such that pages belonging to the same process are contiguously located within the message buffer.

The size and number of messages sent to each process are aggregated globally using `MPI_Allreduce`. So that receive buffers of the appropriate size can be prepared.

2. *Send Write-notices:* Each contiguously packed list of page numbers is sent to their respective home processes. Non-blocking send is used so that processes may issue all sends and proceed with the next step. (This is possible as the buffers for the send operations are at different portions of the message buffer that was packed in the previous step.)
3. *Process Write-notices into Migration-notices:*

- (a) Receive page number lists from remote processes (those sent in the previous step). Each pages list received is processed before receiving the next list. For each page number in a page list, an operation called PROCESS-WRITE-NOTICE is called to decide whether where a page should migrate (migrating to the original home indicates no migration).
 - (b) Migration decisions made in the previous step are arranged into a message buffer such that pages that should migrate to the same process are contiguously located within the buffer. Each contiguous list of page numbers is preceded by the process ID and number of pages in the region. This embedded information is necessary as migration-notices will be concatenated in the next step (via `MPI_Allgatherv`).
4. *Share Migration-notices:* Migration-notices are shared using `MPI_Allgatherv`. This function allows buffers of different lengths (at each process) to be gathered. After this step, all processes will have access to all migration-notices.

The remainder of the BARRIER is similar to the Write-set MCM BARRIER implementation as migration-notices provide the same level of information as modified-set and home-map. The page state machine for this MCM is identical to the Write-set MCM's page state machine shown in figure 3.12.

Two alternative implementations of the PROCESS-WRITE-NOTICE operation, used in step 3 of the BARRIER operation, are available. The first is based on a single-writer strategy, while the second is based on a multiple-writer strategy.

- Scalable Single-writer MCM: This MCM extends from the Scalable MCM class and provides an implementation of the PROCESS-WRITE-NOTICE operation that migrates a page to a process only if that process is the only writer to that page.
- Scalable Multiple-writer MCM: This MCM extends from the Scalable MCM class and provides an implementation of the PROCESS-WRITE-NOTICE operation that migrates a page to one of the writers to that page. The original home process is always given priority if it is a writer. Otherwise, the process from which the first write-notice for a page is received would be selected to be the new home process.

Figure 3.13: The Scalable MCM. Write-notices for modified pages are first gathered at their respective home processes, where migration decisions for each page are made and a migration-notice created. Next, migration-notices are shared so that all process can know of all the migration decisions. These decisions are used to guide diffing and invalidation activity.

3.4 Danui Application Programming Interface

This section describes the Danui SDSM Application Programming Interface (API). Initialization in Danui is a two step process. The first step uses the `DSM_Init_arg` function listed below to setup MPI and remove all Danui SDSM command line options from `argc` and `argv`.

```
void DSM_Init_arg(int *argc, char ***argv);
```

The `DSM_Init_arg_mpi` function listed below may alternatively be used. The difference using `DSM_Init_arg` and `DSM_Init_arg_mpi` is that the former initializes MPI for the user, while `DSM_Init_arg_mpi` assumes that MPI has already been initialized and that the SDSM should use the provided `MPI_Comm` handle. The latter approach enables multiple instances of the SDSM to be initialized for disjointed groups of processes.

```
void DSM_Init_arg_mpi(int *argc, char ***argv, MPI_Comm comm_to_dup);
```

Once either of the initialization functions mentioned above have completed, the following Danui functions become available for use.

```
int DSM_Get_pe(void);
```

```
int DSM_Get_ps(void);
```

```
MPI_Comm DSM_Get_comm(void);
```

The `DSM_Get_pe` and `DSM_Get_ps` functions return the process ID and number of processes respectively. The `DSM_Get_comm` function returns an MPI communicator handle that can be used for MPI communication. Values returned by the `DSM_Get_pe` and `DSM_Get_ps` functions are consistent with `MPI_Comm_rank` and `MPI_Comm_size` calls on communicators returned by `DSM_Get_comm`.

The purpose of making these functions available is so that, if necessary, processes can communicate to determine the amount of shared memory that a program would require. Once this amount is known, the second initialization step can be performed using the `DSM_Init_mem` function listed below.

```
void DSM_Init_mem(size_t dsmsize);
```

Form here on, shared memory operations can be used.

Shared memory may be obtained using four different allocator functions as listed below. Those with function names containing the term “collective” have to be called by all processes, in which case the address of the allocated region is returned to all processes. Functions with names containing the term “reserved” will allocate memory that is aligned to an SDSM page, and also guarantee all of the allocated pages will not be used for any other allocated region.

```

void* DSM_Alloc(size_t size);
void* DSM_Alloc_collective(size_t size);
void* DSM_Alloc_reserved(size_t size);
void* DSM_Alloc_reserved_collective(size_t size);

```

Shared memory may be deallocated using either of the `DSM_Free` or `DSM_Free_collective` functions. As before, functions with names containing the “collective” term have to be called by all processes. The address provided for deallocation must have been an address that was returned by an allocator function.

```

void DSM_Free(void *addr);
void DSM_Free_collective(void *addr);

```

The `FLUSH` and `BARRIER` MCM operations can be invoked using similarly named `SDSM` functions.

```

void DSM_Flush(void);
void DSM_Barrier(void);

```

Locks can be created, used, and destroyed through the functions listed below. The return value of the `DSM_Mutex_try` function is 1 if the lock was obtained, or 0 if the attempt failed.

```

void DSM_Mutex_init(DSM_Mutex_t *mutex);
void DSM_Mutex_init_collective(DSM_Mutex_t *mutex);
void DSM_Mutex_lock(DSM_Mutex_t mutex);
int DSM_Mutex_try(DSM_Mutex_t mutex);
void DSM_Mutex_unlock(DSM_Mutex_t mutex);
void DSM_Mutex_destroy(DSM_Mutex_t *mutex);

```

Finally, the `DSM_Finalize` function can be used to finalize the `SDSM`. If the `SDSM` was initialized using `DSM_Init_arg` (and not `DSM_Init_arg_mpi`), then underlying MPI library is also finalized.

```

void DSM_Finalize(void);

```

3.4.1 Module Selection

Alternative implementations of the Lock and Heap modules were presented. Selection of Lock and Heap implementations can be done using the following command line options:

- Lock module options:

- “--dsm lock mastq” — Use Master Queue implementation.
- “--dsm lock distq” — Use Distributed Queue implementation (default).

- Heap module options:

- “--dsm heap mcm invall” — Use Invalidate All MCM.
- “--dsm heap mcm selinv” — Use Selective Invalidation MCM.
- “--dsm heap mcm sw” — Use Write-set Single-writer MCM.
- “--dsm heap mcm lowid” — Use Write-set Lowest-ID MCM.
- “--dsm heap mcm nhomes” — Use Write-set N-Homes MCM.
- “--dsm heap mcm ssw” — Use Scalable Single-writer MCM.
- “--dsm heap mcm smw” — Use Scalable Multiple-writers MCM (default).

3.4.2 Example Code: Finite-difference Approximation of the 2D Laplace Equation

An example code using the Danui SDSM API is presented below. The code implements a finite-difference approximation of a 2D Laplace equation and is shown in two parts; the `main()` function, and computation functions `grids()` and `lap()`. For simplicity, all variables have been declared as `static` in the global scope of the process.

The `main()` function is shown in figure 3.14. It initializes the SDSM, sets up the problem size that is fixed at $N = 512$, calculates the work division offset (b) and length (c), initializes the SDSM shared memory, and allocates two shared grids. Next, the grids are initialized using `grids()`, the processes are synchronized using `DSM_Barrier`, and the main computation begins in `lap()`. A sum of all the values in the final grid (`old`) is computed and reported by process P0; after which the program terminates.

The `grids()` and `lap()` functions are shown in figure 3.15. The `grids()` function initialize the original state of the grids to have values of 100 at the edges and 1 in the interior. The `lap()` function performs the main computation for 10 iterations that are synchronized by `DSM_Barrier` calls.

```

#include <dsm.h>
#include <stdio.h>
static int N, b, c, myid, nproc, i, j, k;
static double *new, *old, *tmp, sum;
static void grids(void);
static void lap(void);
int main(int argc, char **argv) {
    DSM_Init_arg(&argc, &argv);
    myid = DSM_Get_pe();
    nproc = DSM_Get_ps();
    N = 512; c = N/nproc; b = myid*c;
    if (myid == nproc-1) c += N/nproc;
    DSM_Init_mem(sizeof(double)*N*N*2);
    new = DSM_Alloc_collective(N*N * sizeof(double));
    old = DSM_Alloc_collective(N*N * sizeof(double));
    grids();
    DSM_Barrier();
    lap();
    if (myid == 0) {
        sum = 0.0;
        for (i=0; i<N*N; i++) sum += old[i];
        printf("sum = %lf\n", sum);
    }
    DSM_Finalize();
    return 0;
}

```

Figure 3.14: main() function of the 2D Laplace example.

```

static void grids(void) {
    for (i=b; i<b+c; i++)
        for (j=0; j<N; j++)
            if (i==0 || i==N-1 || j==0 || j==N-1)
                new[i*N+j] = old[i*N+j] = 100.0;
            else
                new[i*N+j] = old[i*N+j] = 1.0;
}

static void lap(void) {
    for (k=0; k<10; k++) {
        for (i=b; i<b+c; i++)
            for (j=0; j<N; j++)
                if (!(i==0 || i==N-1 || j==0 || j==N-1))
                    new[i*N+j] = (old[(i+1)*N+(j)] + old[(i-1)*N+(j)]
                                   + old[(i)*N+(j+1)] + old[(i)*N+(j-1)])/4;
        DSM_Barrier();
        tmp = new; new = old; old = tmp;
    }
}

```

Figure 3.15: grids() and lap() functions of the 2D Laplace example.

3.5 Summary

The Danui SDSM is build using three modules: Event, Lock, and Heap. The Event module provides a mechanism for executing predefined functions known as event handlers at remote processes in a one-sided fashion. Previous implementations of MPI-based SDSMs had to work with MPI implementations that did not support calls from multiple threads concurrently. The implementation of the Danui SDSM acknowledges that implementations of the MPI standard [51, 52] have since improved to a point where calling the library from multiple threads is supported by prominent MPI implementations (e.g. Open MPI [3]).

The Lock module focuses on the synchronization aspect of the mutual exclusion problem. Two approaches were implemented: Master Queue, and Distributed Queue. The idea of the *Master Queue* approach is that each lock is managed by a designated process known as the master. This process manages a queue of processes that are blocked waiting for the lock. This approach is also used by the SCASH [35] SDSM. In contrast, the *Distributed Queue* approach manages this queue in a distributed fashion by having the processes waiting for a lock form a queue by maintaining pointers to the next process in the queue. The advantage of this approach is that transfer of a lock from a releasing process to an acquiring process can be done using one message; whereas in the Master Queue approach release and acquire are coordinated by the

Master Queue, requiring two messages. The Distributed Queue implementation is based on a description, by Fu and Tzeng [28], of how locks are implemented in TreadMarks [43, 42, 9].

The Heap module provides a shared address space across multiple processes. Memory consistency is managed by a hierarchy of so called Memory Consistency Managers (MCMs). The implementations focus on home-based approaches as the intention is to support an integrated shared memory and message passing programming environment. In total seven different MCMs may be instantiated, these are as follows:

- Invalidate All MCM
- Selective Invalidation MCM
- Write-set Single-writer MCM (Write-set MCM using a single-writer policy.)
- Write-set Lowest-ID MCM (Write-set MCM using a multiple-writers policy that selects the writer with the lowest process ID.)
- Write-set N-Homes MCM (Write-set MCM using a multiple-writers policy that selects the process that hosts the least number of pages.)
- Scalable Single-writer MCM (Scalable MCM using a single-writer policy.)
- Scalable Multiple-writers MCM (Scalable MCM using a multiple-writers policy.)

Chapter 4

Performance Characterization of the Danui SDSM

This chapter characterizes the performance of the Danui SDSM. It details cost models for SDSM operations from each of the three main SDSM modules introduced in chapter 3; namely, the Event, Lock, and Heap modules.

The Event module provides the SDSM with functionality for a process to invoke the execution of code at a remote process. This functionality is used by the Lock and Heap modules to implement various SDSM operations. Thus the performance of the Event module is a cornerstone that must be characterized before models for the Lock and Heap modules may be developed. In this chapter the different types of event communication are identified, and benchmarks are described to measure each type.

The Lock module provides processes with synchronization functionality that can be used to serialize access to critical regions of a program. Alternative implementations provide centralized and decentralized approaches for maintaining queues of processes waiting for access to locks. In this chapter the synchronization problem is characterized by the level of contention on a lock. A detailed analysis of lock operation costs with respect to each contention level is given in terms of the number of events communicated. Designs for benchmarks to measure each lock operation and contention case are detailed and timing measurements provided. These measurements are verified using an event count analysis and timing measurements for the Event module.

The Heap module provides processes with a shared address space. Memory consistency for this address space is managed by so called Memory Consistency Managers (MCMs) that provide three main operations: FAULT, FLUSH, and BARRIER. Alternative implementations are provided for the BARRIER operation. Cost models for all operations are developed, and further analysis for the BARRIER operations compares the alternative implementations with respect to different numbers of processes, SDSM pages, and modified SDSM pages.

This chapter also introduces a benchmark called MCBENCH [71] that we developed for measuring the time spent by a shared memory system in coherency tasks for maintaining memory consistency. MCBENCH is used in this chapter to compare the alternative MCMs.

A critical path performance model is developed that extends our previous work reported in [20]. The extended model improves on the original model by removing the assumption of load balanced computation and incorporating the more detailed cost models developed here for the Heap module. An issue that was raised in [20] was that critical path models required more detailed profiling information and were in general harder to apply. An approach for addressing this called Modeled Time is presented that builds the extended critical path model into the Danui SDSM. This includes a special timing call to provide wall-times that have been adjusted by the cost models.

Modeled time is compared against actual times to check the accuracy of the critical path model and the underlying cost models. The MCBENCH benchmark and an implementation of a 2D FFT problem were used to obtain modeled and actual times.

Section 4.2 studies costs related to the Event module. Section 4.3 presents cost models for the Lock module operations. Section 4.4 presents cost models for the Heap module operations. Section 4.5 compares the performance of the various barrier alternatives. Section 4.6 introduces the memory consistency benchmark MCBENCH [71], which is then used to compare the performance of seven MCM implementations. Section 4.7 introduces a performance model for the SDSM. Section 4.8 summarizes the findings of this chapter.

First, however, before characterizing performance of the SDSM, the latency and bandwidth performance of the MPI library running on the evaluation system used in this work is assessed.

4.1 Latency and Bandwidth of MPI on the Evaluation System

Danui uses the Message Passing Interface (MPI) to perform inter-process communication. This section profiles the performance of MPI running on the evaluation system used in this work.

The evaluation system comprises a cluster with 156 nodes. Each node contains two quad-core 3.0GHz Intel Harpertown CPUs and 16GB of memory. The network used is Gigabit Ethernet and IP over InfiniBand (IPoIB)¹. The version of MPI used is Open

¹ Although an InfiniBand network is available on the cluster, multi-threaded MPI usage on InfiniBand is not stable in Open MPI 1.4. The `-mca btl tcp` configuration was used; which uses both Gigabit Ethernet and IP-over-InfiniBand (IPoIB) for communication. However, for the purpose of developing performance models, any network is usable.

Table 4.1: Specifications of hardware and software used in this evaluation.

Item	Details
Nodes	156
CPU	Two quad-core 3.0GHz Intel Harpertown CPUs with 1600MHz front side bus. (per node)
Cache	6MB L2 cache per quad-code CPU.
Memory	16GB DDR2-800
Network	DDR InfiniBand (using IPoIB) and Gigabit Ethernet
Operating System	CentOS 5.2 Linux distribution (based on RHEL5.2)
MPI	Open MPI 1.4

MPI 1.4. Full details of this system are given in table 4.1.²

Although the cluster had 156 nodes, the maximum number of cores that could be requested from the resource management system was 64. With eight cores per node, the maximum number of nodes that could be requested was eight.

The latency and bandwidth of MPI on the evaluation system was measured using the Ohio State University MPI Benchmarks 3.1.1 (OMB-3.1.1) [2]. Details of the latency and bandwidth benchmarks and the resulting performance on the evaluation system are given in the following subsections.

4.1.1 MPI Latency Benchmark

OMB-3.1.1 provides two latency benchmarks, one is single-threaded while the other is multi-threaded:

- Single-threaded latency is measured using two processes. The first process sends a message of some size to the second process and waits for a reply. The second process waits for a message from the first process and responds by sending a message of the same size back to the first process. Messages are sent using blocking MPI functions (`MPI_Send` and `MPI_Recv`). The whole procedure is repeated for a specified number of repetitions.
- Multi-threaded latency is also measured using two processes. In this case, the second process creates two threads, each of which performs `MPI_Recv` and `MPI_Send` operations (as in the single-threaded case), but for *only half of the total number of repetitions*. The threads run concurrently, and messages are not distinguished by way of message tags. Thus each `MPI_Send` issued by the first process can be received by an `MPI_Recv` call executed by either thread at the second process in a non-deterministic fashion (at least until one thread has completed all its repetitions).

²More information about the cluster is available here <http://nf.nci.org.au/facilities/xe/hardware.php>

Timing measurements are taken at the first process for some number of repetitions and averaged.

4.1.2 MPI Bandwidth Benchmark

Bandwidth is measured using two processes and two different types of messages: data and acknowledgement. *Data messages* differ from *acknowledgement messages* in that different data message sizes are used for each test case, while acknowledgement messages are of fixed size and used purely for synchronizing the processes. Benchmark communication is structured as follows:

- The first process sends multiple data messages to the second process using `MPI_Isend`, blocks on `MPI_Waitall` for the non-blocking calls to complete, and then waits for an acknowledgement message from the second process using `MPI_Recv`.
- The second process issues multiple `MPI_Irecv` calls to receive the data messages from the first process. It then blocks on `MPI_Waitall` for the non-blocking calls to complete, and then sends an acknowledgement message to the first process using `MPI_Send`.

The purpose of communicating multiple data messages using back-to-back non-blocking calls is to saturate the network. Timing measurements are taken at the first process for some number of repetitions and averaged.

4.1.3 Measured Latency and Bandwidth

Results for the latency and bandwidth measurements are plotted in figure 4.1. Panel (a) plots the latency for the single- and multi-threaded cases, while panel (b) plots the bandwidth.

Latency for the single-threaded case is ~ 35 microseconds for small messages sizes up to 128 bytes. For the same message sizes, multi-threaded latency is ~ 1.5 times greater at about 50 microseconds. This higher latency is expected as the MPI library has to operate in a thread-safe manner. For larger message sizes the difference between the single and multi-threaded results disappears. For example, the latency for a 4MB message was measured to be 19116 and 18713 microseconds for the single and multi-threaded cases respectively; indicating that at these sizes, communication overheads overshadow thread-safety overheads.

For the bandwidth benchmark peak performance is ~ 1.9 Gbps, obtained for messages that are 32KB or larger. The drop in bandwidth observed at 64KB, 128KB, and 256KB is explained by a switch from eager to rendezvous protocols [17]; for Open MPI 1.4, this switch occurs at 64KB.

Figure 4.1: MPI latency and bandwidth measurements for Open MPI 1.4 over Gigabit Ethernet.

4.2 Event Module Performance Model

The Event module provides infrastructure for invoking event handler functions at remote processes. In this section the latency for sending events is measured and contrasted with the MPI latencies observed in the previous section.

When the Event module is initialized at a process, the main thread at that process creates a background daemon thread to receive event messages. Events are always received by the daemon thread, but may be sent from both the main and the daemon thread. Therefore, between processes two different event communication classes are possible; main-to-daemon, and daemon-to-daemon. Benchmarks for measuring the latency for these communication classes are described as follows:

- *Main-to-daemon latency:* This is measured using two processes. The first process sends an event of some size to the second process and waits for an acknowledgement message using `MPI_Recv`. When the event is received by the daemon thread at the second process, an acknowledgement message is sent back to the first process using `MPI_Send`. Timing measurements are recorded at the first process and start from before sending the event and end when the acknowledgement message is received. The cost of the acknowledgement message (obtained in the same way as the MPI latency measurements previously) is subtracted from the measured time to obtain the latency for sending an event of that size.
- *Daemon-to-daemon latency:* This is measured using two processes and two events, X and Y . The first process sends event X to the second process and waits for an

Table 4.2: Latency (μs) for main-to-daemon and daemon-to-main event communication and cost of sending events in the “send-and-forget” manner (see section 4.2). MPI latencies using the multi-threaded latency benchmark (see section 4.1.1) are provided for comparison. (All timing measurements are averages.)

Bytes	Main-to-Daemon	Daemon-to-Daemon	Send-and-Forget	MPI Latency
16	58	59	9	52
32	61	66	9	52
64	64	63	9	53

acknowledgement message. At the second process, receipt and execution of event X cause it to send event Y to the first process. Receipt and execution of event Y at the first process then cause it to send event X back to the second process. Timing measurements are recorded at the second process when it executes event X . Global variables are used to track the number of times event X has been received to determine when timing should start and end. When the measurement is complete, the second process sends an acknowledgement message that contains the measured time back to the first process. (In this case, the measured time consists only of daemon-to-daemon event communication, so no subtraction of acknowledgement message cost is necessary.)

In some cases, events are sent without expecting any response. Referred to as *send-and-forget* communication, this style of communication is used, for example, by the UNSET-LOCK operation in the Lock module (see section 3.2). Measuring the cost of this operation is therefore necessary to characterize some of the SDSM operations.

To measure the cost of send-and-forget communication, two processes and an event that returns immediately are used. The first process sends the event to the second process multiple times. Timing measurements are recorded at the first process.

Event message sizes range from 12 bytes (size of the event header) to 64 bytes (header plus 52 bytes of parameter data). Event latency measurements were made using parameter sizes of 4, 20, and 52 bytes which correspond to event message sizes of 16, 32, and 64 bytes respectively.

Table 4.2 lists the latency for the two event communication classes and the cost of send-and-forget event sends. The table also includes MPI latencies for messages of equivalent sizes using the multi-threaded latency benchmark detailed in section 4.1.1.

The latencies for the main-to-daemon and the daemon-to-daemon event communications do not differ significantly. On average, event communication has a latency of ~ 62 microseconds. This is about 10 microseconds higher than the measured MPI

latency (multi-threaded case) and can be attributed to the additional tasks that the Event module needs to perform; this includes preparing an event message for sending, invoking its event handler at the receiver, etc.

The cost for the send-and-forget event communication is 9 microseconds for all three event message sizes. This form of communication involves preparing event messages and calling `MPI_Send`. For small message sizes, `MPI_Send` returns after it copies the message into internal buffers for sending in the background. In such cases, the `MPI_Send` call returns before the message has been delivered to the receiving process. Thus the latency for the send-and-forget event communication is mostly the cost of preparing the event message. This involves configuring the event header, copying event parameters into the message, etc. As such, the cost of the send-and-forget event communication explains a significant portion of the 10 microseconds difference between event communication and MPI latencies observed above.

In summary, on the evaluation system the latency for communicating an event is ~ 62 microseconds, while the cost for a send-and-forget event communication is ~ 9 microseconds. Cost models for the Lock and Heap modules described in the sections that follow will refer to event communication latency and send-and-forget communication cost as E and e respectively.

4.3 Lock Module Performance Model

Lock operations include SET-LOCK, TRY-LOCK, and UNSET-LOCK. These operations are synchronization primitives. In the case of SET-LOCK, it is possible to block while waiting for a lock to become available. In such a scenario, care must be taken not to confuse delays resulting from synchronization with lock overheads. Figure 4.2 illustrates this difference. In the figure, a process P_0 calls SET-LOCK on a lock that is held by process P_1 ; thus P_0 has to wait for the lock to become available. At some later time, P_1 calls UNSET-LOCK to release the lock; after which P_0 acquires the lock and completes the SET-LOCK operation.

As the lock was not available when process P_0 called SET-LOCK at time t_0 , but only when process P_1 calls UNSET-LOCK at time t_1 , the period from t_0 to t_1 is a synchronization delay. The remainder of the SET-LOCK call, from time t_1 until the call returns at t_3 , is the SET-LOCK overhead. This overhead is also referred to as the *lock transfer cost* as it is the time spent transferring a lock between the process releasing the lock and the next process in the lock queue. (Note that for the UNSET-LOCK operation, the whole duration of the call, time t_1 to t_2 , is an overhead because UNSET-LOCK is a non-blocking operation.)

On the other hand, the whole duration of a SET-LOCK call should be regarded as an overhead when a lock is available at the time of the call, as there would be

Figure 4.2: Difference between synchronization delays and lock overheads. Initially, process P1 is holding the lock and there is no other process waiting for the lock. At time t_0 , process P0 calls SET-LOCK while process P1 is still holding the lock. The duration of the SET-LOCK call until P1 calls UNSET-LOCK at t_1 is a synchronization delay. The remainder of the time spent in the SET-LOCK call, t_1 to t_3 , is an overhead. For process P1, the whole duration of the UNSET-LOCK call is an overhead.

no synchronization delay. Thus the amount of contention for lock usage needs to be considered when modeling the performance of lock operations. Two contrasting lock contention scenarios are considered:

1. *High contention:* When contention is high, SET-LOCK operations will experience synchronization delays, UNSET-LOCK operations will result in lock transfer to another process, and TRY-LOCK operations will be unsuccessful.
2. *Low contention:* When contention is low, SET-LOCK operations will not have any synchronization delays, UNSET-LOCK operations will not be followed by transfer of the lock, and TRY-LOCK operations will be successful.

A characterization of Lock module operations in terms of the number of events communicated is provided below for both lock queue implementations: Master Queue and Distributed Queue (see section 3.2). Following this characterization, several benchmarks are described to measure lock operation overheads.

4.3.1 Lock Event Count Analysis

The Lock module provides two alternative lock queue implementations: Master Queue and Distributed Queue. In the Master Queue implementation, each lock is associated with a lock queue that is maintained at a designated process called the lock master. In the Distributed Queue implementation, the lock queue is maintained in a distributed fashion by having each process in the queue keep a pointer to the next process in the queue, if any. It is important to note that the Distributed Queue implementation

still requires a master process to track the location of the tail process (last process in the queue). This analysis seeks to characterize the overheads of these alternative lock implementations to determine when one implementation should be used in preference to the other.

A concept called process locality is used here to describe where physically a process that calls a lock routine is executing with respect to the lock master. The following notation is used to describe process locality. R is used to denote the calling process and M the master process. For the Master Queue implementation, we consider the following scenarios:

- $R \rightarrow M$: Process R is not the lock master.
- (R, M) : Process R is also the lock master.

For the Distributed Queue implementation, the master process only serves as an intermediate process through which messages bound for the tail process, denoted as T , are routed. The following locality permutations for processes R , M , and T will be considered:

- $R \rightarrow M \rightarrow T$: All three processes are different and messages have to be sent first to M and then forwarded to T .
- $R \rightarrow (M, T)$: Processes M and T are the same, thus, messages sent to M are not forwarded further.
- $(R, M) \rightarrow T$: Processes R and M are the same, thus, the location of T is known to R and a message may be sent directly to T .
- (R, T) : Processes R and T are the same, thus, no communication is required. Whether M is also the same process or another process does not matter as the objective is to get a message from R to T .

Table 4.3 shows the number and types of events required to perform SET-LOCK, TRY-LOCK and UNSET-LOCK operations for each lock implementation and contention level. The table is organized in two halves. The left half side shows the counts for the Master Queue implementation while the right half side shows the counts for the Distributed Queue implementation. Each is further divided into “Low” and “High” contention sections. These are in turn divided into the different process locality permutations described above.

For the Master Queue implementation, the SET-LOCK (low contention) and TRY-LOCK (low and high contention) operations require two events when $R \neq M$. These events are: 1) R sends a request to M , and 2) M responds to R ’s request. The SET-LOCK (high contention) case also requires two events, although not for the same

Table 4.3: The number of events required to perform SET-LOCK and TRY-LOCK operations varies depending on where the requesting process R is with respect to the master process M . For the Distributed Queue implementation, the location of the tail process T is also important (if not more important). E is the event communication latency; and e is the send-and-forget overhead (see section 4.2).

Operation	— Master Queue —				— Distributed Queue —							
	— Low —		— High —		— Low —				— High —			
	$R \rightarrow M$	(R, M)	$R \rightarrow M$	(R, M)	$R \rightarrow M \rightarrow T$	$R \rightarrow (M, T)$	$(R, M) \rightarrow T$	(R, T)	$R \rightarrow M \rightarrow T$	$R \rightarrow (M, T)$	$(R, M) \rightarrow T$	(R, T)
SET-LOCK	$2E$	0	$2E$	0	$3E$	$2E$	$2E$	0	E	E	E	n/a
TRY-LOCK	$2E$	0	$2E$	0	$3E$	$2E$	$2E$	0	$3E$	$2E$	$2E$	n/a
UNSET-LOCK	e	0	e	e	0	0	0	0	e	e	e	n/a

reasons. In this case, a lock request event is sent to M , however this overlaps with lock usage and as a consequence this does not constitute an overhead. Rather the two events arises from transferring the lock from the releasing process, to the lock master and then onto the waiting process. In contrast, the UNSET-LOCK requires one send-and-forget event communication, except for the case where the releasing process is also the master process and there are no other processes waiting for the lock (low contention). For the $R \rightarrow M$ cases, a *Release* event is sent to master process M . For the high contention (R, M) case, an *Acquire* event is sent to the next process in the lock queue

For the Distributed Queue implementation, SET-LOCK operations costs more in the low contention case than in the high contention case. This is due to the three-way communications between processes R , M , and T . Consider the “ $R \rightarrow M \rightarrow T$ ” process locality case. When contention is low, three events are required to obtain the lock. These are: 1) a request event from R to M , 2) a forwarding of the event from M to T , and 3) an acquire event from T back to R . However, when contention is high, the first two events overlap with lock usage at some other process (not necessarily T). Thus, the overhead incurred would only be that for the final event when T releases the lock to R . The next two process locality cases, where $R \neq T$, require one event less when contention is low as having $R = M$ or $M = T$ eliminates the need to forward events. One event is still required when contention is high to transfer the lock from T to R . The case where $R = T$ represents the trivial case for the Distributed Queue implementation as zero events are required to get the request to the tail process. Note that this case is not applicable for the high contention scenario, because having $R = T$ when R is

requesting the lock implies that no process is using the lock; that is, this is not a high contention scenario.

TRY-LOCK operations for the Distributed Queue implementation require the same number of events for both high and low contention scenarios as it is not a blocking operation. Finally, the UNSET-LOCK operation requires the releasing process to send an *Acquire* message to the next process in the queue if any. Thus, one send-and-forget event communication is needed for the high contention scenario; and no communication is required for the low contention scenario.

4.3.2 Lock Cost Measurement

A set of micro-benchmarks were designed to create low and high contention scenarios. All measurements are made using the worst case process locality for each implementation (i.e. $R \rightarrow M$ for the Master Queue, and $R \rightarrow M \rightarrow T$ for the Distributed Queue).

Measuring lock overheads for the low contention scenario is straightforward as there is no need to arrange for another process to acquire the lock beforehand. The following steps describe how measurements were taken for the low contention case:

1. Set $T_{\text{set},\text{low}} \leftarrow$ Time taken to call SET-LOCK.
2. Set $T_{\text{unset},\text{low}} \leftarrow$ Time taken to call UNSET-LOCK.
3. Set $T_{\text{try},\text{low}} \leftarrow$ Time taken to call TRY-LOCK.

Measuring lock overheads for the high contention situation requires additional preparation to ensure that 1) the SET-LOCK and TRY-LOCK operations are called only when another process is already holding the lock, and 2) the UNSET-LOCK operation is only called when another process is waiting for the lock. Additionally, for the SET-LOCK operation, the time resulting from synchronization delays must be factored out.

The cost of the TRY-LOCK operation is measured for the high contention scenario using two processes. The first process begins by calling SET-LOCK to obtain the lock. Following this, both processes synchronize. The second process, then calls TRY-LOCK which is unsuccessful. Timing measurements for several repetitions of the TRY-LOCK call are taken at the second process and averaged.

The costs for the SET-LOCK and UNSET-LOCK operations are also measured using two processes. To simulate a high contention scenario, we want the lock operations to occur in the sequence listed below:

Process P0:

1. SET-LOCK

2. -

3. UNSET-LOCK (Transfer.)

Process P1:

1. -

2. SET-LOCK (Blocks.)

3. -

While it is possible to use synchronization to ensure that process P0 has called SET-LOCK before process P1 calls SET-LOCK, it is not possible for P1 to synchronize with P0 while it is blocked in the SET-LOCK call. Thus, P0 cannot rely on synchronization to know when to call UNSET-LOCK. Moreover, as shown in figure 4.2, the portion of the SET-LOCK duration that needs to be measured begins while process P1 is blocked and unable to make timing measurements.

To overcome this problem, DELAY routines that execute for some fixed amount of time are used. The following steps describe how DELAY calls are used, together with SET-LOCK and UNSET-LOCK calls, to measure the lock overheads:

Process P0:

1. SET-LOCK

2. Synchronize

3. For N repetitions:

(a) DELAY

(b) UNSET-LOCK

(c) SET-LOCK

Process P1:

1. -

2. Synchronize

3. For N repetitions:

(a) SET-LOCK

(b) DELAY

(c) UNSET-LOCK

Figure 4.3 illustrates how these steps fit together for $N = 1$. In this figure, d is the duration of the DELAY operation, u is the duration of the UNSET-LOCK operation, and $s + s'$ forms the duration of the SET-LOCK operation. The s portion of the SET-LOCK operation represents the time spent waiting for the lock to be available. This is a synchronization delay and should not be attributed to the cost of the SET-LOCK operation. The s' portion of the SET-LOCK operation begins as soon as the other process calls UNSET-LOCK. This represents the lock transfer overhead which the benchmark aims to measure.

The time, T , is measured at P0 and is equal to $d + u + s + s'$ (for this $N = 1$ example). It can be seen from the figure that the duration of $u + s$ at P0 is equivalent to the duration $s' + d$ at P1. Thus, T can be simplified to equal $2(d + s')$. Since T and d are known, the lock transfer overhead s' can be obtained.

Table 4.4 lists the timing measurements for the two different lock implementations, the three different lock operations, and the two different contention levels.

Figure 4.3: Measuring high contention SET-LOCK and UNSET-LOCK overheads. The duration of the SET-LOCK operation is partitioned into s and s' parts; with s being the part where the process is blocked due to lock synchronization, and s' the overhead of transferring the lock. The breakdown of the timed region T at P0, thus, consist of durations $d + u + s + s'$ (in this figure, only one iteration is shown). It can be seen that $u + s$ is equal to $s' + d$. Thus, T can be simplified to $2(d + s')$. Since T and d are known durations, the overhead of the lock transfer (s') can be solved.

Table 4.4: Timing measurements for the Master Queue and Distributed Queue lock implementations; for each operation and contention level. (Timing results in microseconds.)

Operation	Master Queue		Distributed Queue	
	Low	High	Low	High
SET-LOCK	103	111	179	54
TRY-LOCK	119	113	177	167
UNSET-LOCK	10	8	0	8

For the Master Queue implementation, the results show average overheads of ~ 112 microseconds for the SET-LOCK and TRY-LOCK operations, and ~ 9 microseconds for the UNSET-LOCK operation. From table 4.3 these two values correspond to $2E$ and e respectively.

For the Distributed Queue implementation, overheads for the SET-LOCK (low contention) and TRY-LOCK operations average 174 microseconds. This corresponds to $3E$ in table 4.3. The SET-LOCK (high contention) was measured to be 54 microseconds. This corresponds to E in table 4.3. Finally, UNSET-LOCK overheads were measured to be 0 and 8 microseconds for low and high contention levels respectively. This corresponds to 0 and e in table 4.3.

In summary, the results obtained from the benchmarks detail above are explained by the event count analysis given in table 4.3.

The results suggest that the choice of lock implementation depends on the level of contention and whether TRY-LOCK operations are used. If high contention is expected and TRY-LOCK operations are not used, then it is better to use the Distributed Queue implementation. Otherwise, it is better to use the Master Queue implementation.

Although the criteria given above pick the Master Queue implementation in all but one case, the Distributed Queue implementation has been selected as the default lock implementation for the Danui SDSM. This choice was arrived at by considering that a low lock usage frequency will likely result in low contention, while a high lock usage frequency will likely result in high contention. Therefore, it is more important to optimize for the high contention case than it is for the low contention case.

4.4 Heap Module Performance Model

The Heap module provides the shared address space that the SDSM presents to user applications. This space is partitioned into pages, with consistency maintained at the page level by a Memory Consistency Manager (MCM). This section characterizes the cost of the FAULT, FLUSH, and BARRIER MCM operations.

4.4.1 Fault Cost Model

The time spent in FAULT operations depend on the state of the fault page. This may be LRO, RUM, or RRO.

Resolving LRO and RRO page faults involves only local action by the faulting process. Resolving the RUM page fault involves use of the PAGE-FETCH operation detailed in section 3.3.1.1, and requires inter-process communication. Since in all cases the operations may be affected by the size of the SDSM page, a linear function on the SDSM page size was used to model the costs of all three page fault cases. This is given

Figure 4.4: Page fault costs as page size increases on the evaluation system (see section 4.1). Costs for faults on remote pages increase as the page size increases. Costs for faults on local pages remain constant.

as follows:

$$F(b_{page}) = c_1 + c_2 b_{page}; \quad (4.1)$$

where b_{page} is the page size.

The cost of page faults for each state are obtained by measuring the time required to access pages in the LRO, RUM, and RRO states. For each page state, multiple pages of that state are accessed and an average time is computed. To minimize the time associated with accessing the page, only one byte is accessed from each page.

Figure 4.4 plots the page fault costs for each fault type as the page size increases (in 4KB increments). The results show that the cost of LRO faults remains at ~ 4 microseconds regardless of the SDSM page size. For RUM faults, the costs increase with page size due to the inter-process communication required to fetch the page. For RRO faults, the costs also increase with page size as twinning becomes more expensive for larger pages. Cost models for all three page faults are as follows (times given in μs):

$$\begin{aligned} \text{Cost of LRO fault, } F_{lro}(b_{page}) &= 4 \\ \text{Cost of RUM fault, } F_{rum}(b_{page}) &= 156.18 + 6.528 \times 10^{-3} \times b_{page} \\ \text{Cost of RRO fault, } F_{rro}(b_{page}) &= 4.71 + 0.761 \times 10^{-3} \times b_{page} \end{aligned} \quad (4.2)$$

4.4.2 Flush Cost Model

FLUSH operations are used to synchronize shared memory. Following a FLUSH operation, the SDSM guarantees that all modifications that were made at the calling process are completed (made globally visible). In addition, all completed modifications

at remote processes are made visible to the calling process.

The first part of the guarantee is achieved by sending modifications made to RRW pages back to their home processes as diff messages. An acknowledgement message is received for each diff message sent (otherwise the SDSM cannot guarantee that the modifications have been completed). The second part of the guarantee is achieved by invalidating all remote pages. By doing so, subsequent accesses to remote pages will trigger page fetch actions that update the pages from their respective home processes; thereby bringing into view any completed modifications by remote processes to those pages.

The FLUSH implementation consists of three steps. The first step is a loop over all RRW pages. For each RRW page, modifications are encoded into a diff message and sent to the home process for that page, then the page is invalidated. The second step invalidates all RRO pages. Finally, in the third step, the calling process waits for all acknowledgements to be received.

When a diff is received at a home process, the modifications in the diff are applied to the actual shared page and an acknowledgement is sent back in reply. These acknowledgement messages are sent using an event, which allows acknowledgements to be received in the background at the flushing process (by the Event daemon thread). Thus, by the time the flushing process executes the third step, it only needs to wait for the remainder of the acknowledgement messages.

Figure 4.5 illustrates how computation and communication may overlap during flush operations. A “One Diff” case is shown on the left. Here, the time spent by the flushing process P0 is divided into two parts, costing c_1 and c_2 time respectively. This gives a total time of $c_1 + c_2$. A “Many Diff” case is shown on the right. Here n diffs are sent and acknowledgements are received in the background. The total time for this case is $nc_1 + c_2$.

Cost c_1 represents the cost of encoding and sending a diff. The cost of this activity is determined by both the page size and the size of the generated diff.

Thus, the cost of diffing page p , can be written as: $c_3b_{page} + c_4b_{diff,p} + c_5$; where b_{page} is the page size and $b_{diff,p}$ is the diff size for page p . Based on this, the cost of diffing in a FLUSH operation can be expressed as follows:

$$D(b_{page}, P_{rrw}, n_{rrw}) = (c_3b_{page} + c_5)n_{rrw} + c_4 \sum_{p \in P_{rrw}} b_{diff,p}; \quad (4.3)$$

where n_{rrw} is the number of RRW pages, and P_{rrw} is the set of RRW pages.

After diffing, the FLUSH must also invalidate RRO pages. This cost can be expressed as follows:

$$I(n_{rro}) = n_{rro}c_6 + c_7; \quad (4.4)$$

Figure 4.5: Computation and communication can overlap when many diffs need to be sent as acknowledgements are sent using events and received in the background (by the Event daemon thread) while the flushing process (P0) continues to encode and send more diffs.

where n_{rro} is the number of RRO pages.

The FLUSH cost model is a combination of equations 4.3 and 4.4. However, as RRO pages are invalidated after sending all diffs, it is possible for this cost to overlap with the waiting period for the last acknowledgement message. Let this waiting period be denoted as A :

$$A = c_2 \quad (4.5)$$

Letting n_{rrw} and n_{rro} denote the number of RRW and RRO pages respectively, the time for a flush operation can now be written as:

$$T_{\text{FLUSH}} = D(b_{\text{page}}, P_{\text{rrw}}, n_{\text{rrw}}) + \max(A, I(n_{\text{rro}})) \quad (4.6)$$

To verify the above cost model, FLUSH operation costs were measured for different number of modified pages and diff sizes (per modified page). In each case the diff size was fixed to be the same for all modified pages. This allowed the summation in equation 4.3 to be simplified to $c_4 b_{\text{diff}} n_{\text{rrw}}$; where b_{diff} is the diff size, which is now the same for all pages. In addition, the number of RRO pages was kept at zero. These simplification enables the cost model to be written as follows:

$$(c_3 b_{\text{page}} + c_4 b_{\text{diff}} + c_5) n_{\text{rrw}} + A. \quad (4.7)$$

Figure 4.6a plots the time measured as a function of the number of RRW pages, for

Figure 4.6: a) FLUSH cost as the number of RRW pages increases, for different diff sizes. b) Per-diff and per-flush costs against diff size (obtained through linear regression of the plots in (a); $r = 1.00$ for all plots).

a variety of diff sizes (13 bytes to 18436 bytes). A strong linear relationship between the flush cost and the number of RRW pages is observed (correlation coefficient $r = 1$ for all plots).

The intercept of these plots correspond to A in equation 4.7. As the intercepts for the smaller diff sizes vary wildly, an average of the intercepts for plots pertaining to diff sizes greater than 4KB was used. This gives a value for A of ~ 178 microseconds.

The slope for each plot in figure 4.6a is given as $c_3b_{page} + c_4b_{diff} + c_5$ by equation 4.7. In order to obtain values for c_3 , c_4 , and c_5 , the slope for each case in figure 4.6a is plotted against the diff size. This is shown in figure 4.6b. A correlation coefficient of 0.99 between the value of the slopes and the diff size is observed.

Further regression analysis is used on figure 4.6b to obtain the values of c_4 and $c_3b_{page} + c_5$ as the slope and intercept respectively ($c_4 = 7.25 \times 10^{-3}$). More values for $c_3b_{page} + c_5$ are obtained by repeating the experiments using different page sizes. Results are shown in figure 4.7 from which the values $c_3 = 1.04 \times 10^{-3}$ and $c_5 = 14$ are obtained.

The remaining unknowns in the FLUSH cost model are the constants for equation 4.4 that describes invalidation cost $I(n_{rrw})$. This cost is measured by calling FLUSH with varying numbers of RRO pages and zero RRW pages. Performing these experiments leads to a value of ~ 3 microseconds for c_6 while c_7 is ~ 0 microseconds.

The overall cost for a FLUSH operation on the evaluation system can be summarized

Figure 4.7: Diffing cost against page size.

as follows:

$$\begin{aligned}
 T_{\text{FLUSH}} &= D(b_{\text{page}}, P_{\text{rrw}}, n_{\text{rrw}}) + \max(A, I(n_{\text{rro}})) \\
 D(b_{\text{page}}, P_{\text{rrw}}, n_{\text{rrw}}) &= (1.04 \times 10^{-3} b_{\text{page}} + 14) n_{\text{rrw}} + 7.25 \times 10^{-3} \sum_{p \in P_{\text{rrw}}} b_{\text{diff},p} \\
 I(n_{\text{rro}}) &= 3n_{\text{rro}} \\
 A &= 178
 \end{aligned} \tag{4.8}$$

4.4.3 Barrier Cost Model

A BARRIER operation synchronizes processes and in doing so orders shared memory accesses into those that happen *before* the operation and those that happen *after*. Thus, the objective of the BARRIER operation is to ensure that all modifications to shared memory that occurred before the BARRIER are globally visible to all processes after the BARRIER.

The MCM hierarchy described in section 3.3 has seven alternative MCMs that may be instantiated. For the purpose of modeling the performance of these implementations, the MCM classes are organized into groups according to similarities in their implementation. Figure 4.8 shows how these groups are organized as the Basic, Write-set, and Scalable barrier groups. Characteristics of each group are as follows:

1. *Basic Barrier Group*: This group consists of only the Invalidate All MCM, which provides a basic barrier implementation that sends all modifications to remote pages back to their respective home processes and then invalidates all remote pages. Collective MPI operations are used to ensure that all modifications have been applied at their home processes before any process completes its BARRIER operation.

Figure 4.8: Organization of MCM classes into barrier groups. Classification is based on implementation similarities.

2. *Write-set Barrier Group*: This group consists of the Selective Invalidation MCM, and the Write-set MCM. The latter supports home-migration and has three descendants that provide different strategies for home-migration. MCMs in this group are related by the use of bit-vector data structures, called write-sets, that each process uses to record a set of pages it has modified since the last barrier. (Each bit in a write-set corresponds to a page.) The implemented BARRIER operations are also similar. During a BARRIER, write-sets are combined into a combined-set; of which every process has an identical copy. Each process then scans its copy of the combined-set to determine the pages to invalidate, migrate, etc. The same mechanisms as in the Basic barrier group are used ensure that modifications have been applied before any process completes the BARRIER operation.
3. *Scalable Barrier Group*: This group consists of the Scalable MCM and its descendant MCM classes. The Scalable MCM was developed to address the poor scalability of the MCMs in the Write-set barrier group. The poor scalability arises during the processing of the combined-set where the number of bits to be processed is a multiple of the number of processes and the number of shared pages. The Scalable MCM solves this scalability issue by providing a BARRIER implementation that scales with the actual number of modified pages and processes.

The scalable BARRIER implementation is especially beneficial when home-migration is supported, because home-migration helps to minimize the number of remote pages modified. In which case, it is even more desirable to have BARRIER operations that scale together with the number of modified pages.

This section develops cost models for each of the three barrier groups.

4.4.3.1 Cost Model for the Basic Barrier Group

Steps for the Basic Barrier are illustrated in figure 4.9. First, an `MPI_Barrier` is used to synchronize processes. This synchronization is necessary to ensure that data structures for synchronizing diff delivery used in the previous BARRIER call have been reset. Second, diffing and invalidation are performed for RRW and RRO pages. An array of counters (*nsends*) is maintained during this step that counts the number of diffs a process has sent to each other process. (For each process *u*, counter *nsends*[*u*] corresponds to the number of diffs that have been sent to *u*.) Third, an `MPI_Allreduce` operation is performed on the *nsends* array using the summation operation. After this operation, the value *nsends*[*u*] will indicate the number of diffs process *u* must receive. Fourth, each process waits until it has received all diffs. Finally, an `MPI_Barrier` is used to synchronize processes to ensure that all diffs have been received before any

Figure 4.9: Basic Barrier. This barrier group is modeled after the FLUSH operation. `MPI_Barrier` calls are used to synchronize processes at the beginning and end of the barrier. Diffs are sent between these calls. `MPI_Allreduce` is also used so that each process may determine the number of diffs it can expect to receive. This eliminates the need to send acknowledgement messages (as is done in FLUSH).

process exits the BARRIER operation.

The `MPI_Barrier` and `MPI_Allreduce` collective operations are used in the Basic Barrier. Cost functions for these are denoted as B and R respectively. `MPI_Barrier` can be performed using $O(\lg n)$ communication steps, with n being the number of processes. Thus,

$$B(n) = c_1 \lg n + c_2. \quad (4.9)$$

`MPI_Allreduce` is similar to `MPI_Barrier` in that $O(\lg n)$ communication steps are taken. However, data also needs to be transferred at each communication step. Thus,

$$\begin{aligned} R(b, n) &= (c_1 b + c_2) \lg n + c_3 \\ &= c_1 b \lg n + c_2 \lg n + c_3 \end{aligned} \quad (4.10)$$

where b is the size of the reduction array in bytes.

The cost of `MPI_Barrier` is measured for $n = 2, 4, 8$ to be 62, 152, and 231 microseconds respectively. Through regression analysis, $c_1 = 84.5$ and $c_2 = -20.7^3$; correlation

³We recognize that " $c_2 = -20.7$ " is unphysical. However barrier cost for the smallest number of processes that require synchronization $n = 2$ would give a physical result.

Figure 4.10: MPI_Allreduce cost plotted against a range of buffer sizes for $n = 2, 4, 8$ processes.

coefficient $r = 1.0$. Thus, on this evaluation system,

$$B(n) = 84.5 \lg n - 20.7. \quad (4.11)$$

The cost of MPI_Allreduce is measured for $n = 2, 4, 8$ with buffer size $b = 64 \dots 1024$ (incrementing in steps of 64 bytes)⁴. Figure 4.10 plots the cost of MPI_Allreduce. To obtain the values of c_1 , c_2 , and c_3 , regression analysis is used to obtain intercepts and slopes for each plot. The slope for some n gives the value of $c_1 \lg n$ for that value of n . Thus, dividing the slope by $\lg n$ gives an estimate for c_1 . Next, when $b = 0$ equation 4.10 is reduced to $c_2 \lg n + c_3$, thus using regression analysis on the intercepts of the plots in figure 4.10 gives estimates for c_2 and c_3 as the new slope and intercept respectively. From the analysis above, $c_1 = 19.8 \times 10^{-3}$, $c_2 = 83.2$, and $c_3 = -15.4$. Thus, on this evaluation system,

$$R(b, n) = (19.8 \times 10^{-3}b + 83.2) \lg n - 15.4. \quad (4.12)$$

Cost models for diffing and invalidation were given previously as D and I in equations 4.3 and 4.4 respectively.

The time spent waiting for all diffs to arrive is the difference between 1) the time required for the last diff that was sent to arrive at its destination, and 2) the time period from when the last diff was sent until the completion of the MPI_Allreduce. As such, we model the time required for the last diff to arrive at its destination rather than

⁴Only small buffers were measured as the scaling for small buffers and big buffers differ. For the purpose of modeling the Basic Barrier, a small buffer size is sufficient as MPI_Allreduce is used to accumulate arrays of n integers. Thus, measurements for 1024 bytes is sufficient for modeling costs up to $n = 256$.

the waiting time. A similar cost was modeled for the FLUSH cost model in equation 4.6 as the acknowledgement time (A) for the last diff sent. The Basic Barrier case differs only in that no acknowledgement messages are sent. Since acknowledgement messages are sent using an event (costing E), the time required for the last diff to arrive at its destination is approximated as $A - E$.

Given the above, the overall cost of the Basic Barrier can be described by the following cost model:

$$\begin{aligned} &\text{Cost of Basic Barrier} \\ &= B(n) + \max \left(\begin{array}{l} \max_i (D_i + I_i) + R(bn, n), \\ \max_i (D_i + A - E) \end{array} \right) + B(n); \end{aligned} \quad (4.13)$$

where n is the number of processes and bn is the byte size of the *nsends* array that holds the number of diffs a process has sent to other processes (this is used for synchronizing diffs, see section 3.3.2 for details).

From equation 4.13 it is apparent that there are two critical paths between the `MPI_Barrier` calls that determine the cost of the Basic Barrier.

- The top path corresponds to all processes having received their diffs before or at the same time as the `MPI_Allreduce` (R) completing.
- The bottom path occurs when some waiting is required after `MPI_Allreduce`. In this case, the longest duration between the `MPI_Barrier` calls is determined by the process that spends the longest time diffing. Note that invalidation and `MPI_Allreduce` costs I and R are not included in the bottom path as they overlap with the diff communication time $A - E$.

The Basic Barrier cost model presented above describes diffing, invalidation, and diff delivery synchronization activities that are common to the remainder of the barriers.

4.4.3.2 Write-set Barrier Group

The Write-set Barrier group is used by the Selective Invalidation, Write-set Single-writer, Write-set Lowest-ID, and Write-set N-Homes MCMs. A write-set is a bit-vector that is used to mark the pages which a process has modified (a 1-bit at the i 'th bit indicates that page i has been modified). Each process updates its write-set when a write fault occurs. At a Write-set Barrier, write-sets from all processes are concatenated into a combined-set at each process. Using the combined-set, each process is then able to make decisions about whether a page should be invalidated, or migrated and to where. (No further communication is required to share the results of these decisions as algorithms that process combined-sets are deterministic.)

Figure 4.11: The Write-set Barrier is an extension of the Basic Barrier. Two additional steps are included to combine write-sets into combined-sets (using `MPI_Allgather`) and to process the combined-sets. While the amount of diffing and invalidations performed may differ from the Basic Barrier, the way they are performed is the same.

The steps involved in the Write-set Barrier are illustrated in figure 4.11. They are largely the same as the Basic Barrier with the exception of write-set concatenation and processing. A combined-set is created by concatenating write-sets using the `MPI_Allgather` collective operation (marked as *G* in figure 4.11). Following this, each process runs the same algorithm on the combined-set to decide if pages should be invalidated or migrated.

The `MPI_Allgather` function can be performed in $\lg n$ communication steps. However, the amount of data that is communicated in each step doubles. That is, if b is the size of the send buffer, then the amount of data sent by each process will increase in each step as $b, 2b, 4b, \dots, \frac{nb}{2}$ (n is the number of processes). Let k denote the step count in the range $0 \dots \lg n - 1$, and the communication time at each step is expressed as $c_1 2^k b + c_2$. The cost of `MPI_Allgather` can be denoted as:

$$\begin{aligned}
 G(b, n) &= \sum_{k=0}^{\lg n - 1} (c_1 2^k b + c_2) + c_3 \\
 &= c_1 b (2^{\lg n + 1} - 1) + c_2 \lg n + c_3
 \end{aligned}
 \tag{4.14}$$

To derive values for c_1 , c_2 , and c_3 the cost of `MPI_Allgather` was measured for $n = 2, 4, 8$ with $b = 4, 8, 16, \dots, 65536$. Compared to `MPI_Allreduce`, it was necessary to make

measurements for a larger buffer size to accommodate the likely size of the write-sets. At 65536 bytes, a write-set for 524288 shared pages can be accommodated. The same regression analysis as used in the Basic Barrier for `MPI_Allreduce` was undertaken for `MPI_Allgather`. (The only difference is that c_1 is obtained by dividing the gradient by $2^{\lg n+1} - 1$.) For the evaluation system, this led to:

$$G(b, n) = 4.26 \times 10^{-3} b(2^{\lg n+1} - 1) + 114 \lg n - 49.6. \quad (4.15)$$

Processing the combined-set is an operation that scales with the total number of bits that need to be inspected. This is a multiple of the number of processes and the number of shared pages managed by the SDSM. Let W be the cost function for processing a combined-set, m the number of shared pages managed by the SDSM, and n the number of processes. The cost model for W can then be expressed as:

$$W(m, n) = c_1 mn + c_2. \quad (4.16)$$

To determine the value of c_1 and c_2 , timing measurements for Write-set Barriers were taken for $n = 2, 4, 8$ and, values of m adjusted such that $mn = 8000, 16000, 24000, 32000, 40000$. Figure 4.12 plots the resulting barrier costs against the number of bits (i.e., mn). Barrier costs increase linearly with respect to the number of bits. Each MCM, however, scales at a different rate as the slope represents the value of c_1 for an MCM. For each MCM, the average of the three slopes was taken as the estimate for c_1 . c_2 was ignored as it is insignificant (there is no code that executes outside of the mn loops). For the evaluation system this resulted in values for c_1 of 6.4×10^{-3} , 6.19×10^{-3} , 5.42×10^{-3} , and 9.38×10^{-3} , for the Selective Invalidation, Write-set Single-writer, Write-set Lowest-ID, and Write-set N-Homes MCMs respectively.

Apart from the inclusion of G and W costs, the model is the same as the cost model for the Basic Barrier in equation 4.13. The cost model for Write-set Barriers is expressed as:

$$\begin{aligned} & \text{Cost of Write-set Barrier} \\ &= B(n) + \left(G(b, n) + W(m, n) \right) + \max \left(\begin{array}{l} \max_i (D_i + I_i) + R(bn, n), \\ \max_i (D_i + A - E) \end{array} \right) + B(n); \end{aligned} \quad (4.17)$$

where m is the number of SDSM pages managed by the SDSM.

4.4.3.3 Scalable Barrier Group

The Scalable Barrier group is used by the Scalable Single-writer and Scalable Multiple-writers MCMs. The main motivation for this MCM is to avoid having to process

Figure 4.12: Barrier synchronization cost against the size of the combined-set in bits. The number of bits is calculated as mn ; where m is the number of shared pages managed by the SDSM and n is the number of processes.

combined-sets. This cost was shown (equation 4.16) to scale as the product of the number of processes and the number of shared pages. In place of write-sets, the Scalable Barrier relies on write-notice for RRW pages which are sent only to the respective page homes, where a migration decision is made. Collective operations are later used to share the results of these decisions so that all processes can act on them accordingly.

Figure 4.13 illustrates the steps involved in the Scalable Barrier. As for the Write-set Barrier, the Basic Barrier is extended with activities prior to diffing and invalidation. Specifically, five new steps are added:

- *Step 1:* This step packs the page numbers of pages modified at a process into a partitioned array such that page numbers for pages belonging to the same remote process are contiguously located in the same partition of the array. This involves: 1) Counting the number of unique remote processes, the number of pages to be sent to each remote process, and the offsets into the array that determines where each partition begins. 2) Packing the page numbers of modified pages into the array partitions defined by the counting step.

Counting and packing can be done in $O(m)$ time, where m is the number of pages modified at the process doing the counting. Given this, the cost for these operations can be expressed by cost function S_1 :

$$S_1(m) = c_1m + c_2 \quad (4.18)$$

- *Step 2:* This step shares information about the partitioned array so that each process knows how many messages it should expect to receive (d') and the max-

Figure 4.13: Scalable Barrier. A weakness of the Write-set barrier is the need to process combined-sets. This barrier avoids this problem by sending write-notice for modified pages only to the home processes for those pages.

imum size message it should prepare to receive. Two `MPI_Allreduce` calls are used to share this information.

The cost for this step is $2R$.

- *Step 3:* This step issues non-blocking sends for each array partition prepared in the first step to their respective destination processes.

Let cost function S_3 represent the cost for sending these messages to d unique destinations:

$$S_3(d) = c_3d + c_4 \quad (4.19)$$

- *Step 4:* This step involves receiving write-notice from remote processes and making a migration decision for each write-notice. Migration decisions are packed into another partitioned array.

Let m' be the number of write-notice (and migration decisions), and let S_4 be the cost function for this step. Then:

$$S_4(m') = c_5m' + c_6 \quad (4.20)$$

- *Step 5:* Finally, this step merges the migration decisions made at each process using two collective operations. `MPI_Allgather` is used to share the total length of the partitioned array produced in the previous step with all processes. These lengths are then used with `MPI_Allgatherv` to gather the migration decisions made at each process.

Cost for `MPI_Allgatherv` not only moves with the various buffer sizes at each process but also with the order in which they are merged at each communication step. In this evaluation, the size of the largest buffer is used together with the cost model for `MPI_Allgather` (see equation 4.14).

Values for constants c_1 – c_6 appearing in equations 4.18–4.20 were derived in an analogous manner to that used in sections 4.4.3.1 and 4.4.3.2. For the evaluation system, this led to the following cost functions for S_1 , S_3 , and S_4 :

$$\begin{aligned} S_1(m) &= (2.19 \times 10^{-2})m + 1.74 \\ S_3(d) &= 9.7d + 2.8 \\ S_4(m') &= (1.22 \times 10^{-1})m' + 124 \end{aligned} \quad (4.21)$$

Note that cost functions for steps 2 and 5 are cost functions for MPI operations.

The Scalable Barrier cost model extends from the Basic Barrier cost model in equation 4.13 by inserting cost functions for the five new steps after the first `MPI_Barrier`

but before diffing and invalidation. This is:

$$\begin{aligned}
 & \text{Cost of Scalable Barrier} \\
 & = B(n) \\
 & + \max_i (S_{1,i}(m)) + 2R(bn, n) \\
 & + \max_i (S_{3,i}(d) + S_{4,i}(m')) + G(1, n) + G_V(bn'_{rw}, n) \\
 & + \max \left(\begin{array}{l} \max_i (D_i + I_i) + R(bn, n), \\ \max_i (D_i + A - E) \end{array} \right) + B(n);
 \end{aligned} \tag{4.22}$$

4.5 Comparison of Barrier Costs

Section 4.4.3 presented a detailed analysis of the costs associated with three different groups of barriers in the Danui SDSM. Factors affecting the performance of the barrier operations were shown to include the number of processes, the number of shared pages, and the number of diffs. In this section we examine experimentally the effect of these factors on barrier costs.

4.5.1 Barrier Cost versus Number of Processes

The larger the number of processes, the bigger the synchronization problem. The implemented BARRIER operations rely on MPI_Barrier to synchronize processes. Two MPI_Barrier calls are used, one at the beginning and another at the end. Between these calls, memory consistency related communication are performed, some of which involve using MPI collective operations; various combinations of MPI_Allreduce, MPI_Allgather, and/or MPI_Allgatherv calls.

Figure 4.14 shows cost as a function of the number of processes for the seven different SDSM BARRIER implementations. For these measurements, the SDSM was configured to manage only one shared page. This is to reduce the overheads that are not related to process synchronization. (The effect of the number of shared pages on barrier cost is explored later.)

Not surprisingly, the costs of all barrier implementations increase with the number of processes. The results also exhibit a noticeable clustering into three groups. From higher to lower costs, these clusters correspond to MCMs in the Scalable Barrier, Write-set Barrier, and Basic Barrier groups. These differences reflect the increase in synchronization overheads as the complexity of the barrier implementation is raised.

The Scalable Barriers are the most costly as they rely on the most number of collective MPI operations between the first and last MPI_Barrier calls. These include three MPI_Allreduce calls, an MPI_Allgather, and an MPI_Allgatherv. The Write-set Barriers are the next most costly, lying between the Scalable and Basic barrier groups.

Figure 4.14: Cost of the BARRIER operation is measured using 2, 4, and 8 processes. The SDSM is configured to manage a shared memory address space of only one page (4KB).

These barriers rely on less MPI collective operations than the Scalable Barriers, but still require an additional `MPI_Allgather` more than the Basic Barrier. Finally, the Basic Barrier group costs the cheapest as the only MPI collective operation used between the `MPI_Barrier` calls is an `MPI_Allreduce` call.

4.5.2 Barrier Cost versus Number of Shared Pages

A scalability issue pertaining to the Write-set Barriers was previously highlighted. Specifically, the issue is with how write-sets are merged into combined-sets, and then how these combined-sets are processed. Write-sets are bit-vectors containing m bits, one for each shared page. Combined-sets are concatenated write-sets from each process containing mn bits, one for each process and shared page. Processing combined-sets is therefore an $O(mn)$ operation (see function W in equation 4.16).

Figure 4.15 shows barrier costs as the number of shared pages increases. Costs for the Write-set Barriers increase as the number of shared pages increase. Costs for the Scalable and Basic Barriers are not affected by the number of shared pages. Thus, although the Scalable Barriers have more synchronization overheads than Write-set Barriers, they provide lower barrier costs when the number of shared pages increases.

Another observation is that costs for the Write-set N-Homes BARRIER are higher than those for the other Write-set Barrier implementations. This is due to how bits in a combined-set are processed. Recall that a combined-set is a concatenation of write-set bits from each process; such that bits can be contiguously accessed by arranging loop iteration as: “for each process,” and then “for each page.” In order to achieve a better distribution of new homes, the N-Homes BARRIER rearranges these iterations in

Figure 4.15: Cost of the BARRIER operation is measured for increasing number of shared memory pages (4000–20000 shared pages). No pages were modified and two processes were used.

the opposite order. This results in strided memory accesses and reduced performance. Calculations in section 4.4.3.2 for the coefficients in equation 4.16 indicated that the Write-set N-Homes implementation incurred a cost of 9.38×10^{-3} microseconds per bit in a combined-set. By comparison the Write-set Lowest-ID implementation incurred a cost of 5.42×10^{-3} microseconds per bit.

4.5.3 Barrier Cost versus Number of Diffs

Modifications that have been made to remote pages need to be communicated back to their respective home processes assuming home-migration is not possible. The cost of processing these modified remote pages is modeled by function D in equation 4.3. This predicts that costs scale linearly with the number of diffs and also with the total number of bytes sent.

Figure 4.16 plots the cost of BARRIER operations as the amount of communicated diffs increases. Unlike the previous measurements, the results do not exhibit any clear distinction between the different BARRIER implementations. This is because the costs for diffing are considerably more expensive than the difference in the synchronization costs between BARRIER implementations. The results clearly illustrates two points:

- It is best for performance to have processes modify only local pages; or mostly local pages. This way, no diff communication is required.
- It is best to use a BARRIER implementation that is capable of home-migration. Although synchronization costs may be higher for these implementations, the cost savings from reduced diffing are far greater.

Figure 4.16: Cost of the BARRIER operation is measured for different amounts of diffs that the operation has to communicate. (Home-migration was inhibited in these tests.) The SDSM was initialized with 20000 shared pages and two processes were used.

4.6 MCBENCH: A Memory Consistency Benchmark

This section compares the performance of the different Memory Consistency Manager (MCM) implementations using a micro-benchmark called MCBENCH [71] that was developed to measure memory consistency overheads.

MCBENCH generates memory consistency workloads by executing a series of Change and Read phases that are separated by BARRIER operations. Shared data is modified during Change phases and, in each subsequent Read phase, read accesses are arranged such that a process always reads what another process wrote during the previous Change phase. Thus, a memory consistency workload is generated which the underlying shared memory system must resolve in some fashion.

The Change and Read phases are executed for several iterations and a timing measurement is taken. This measurement includes other costs that are not related to memory consistency overheads. To separate these costs from the memory consistency costs, a *reference time* is measured by repeating the same iterations but using private instead of shared data. The difference between the two elapsed times is then the time associated with the memory consistency operations.

To ensure that the same memory operations are performed on both the private and shared data, the MCBENCH Change and Read-phase iterations are implemented as a kernel function that accepts an arbitrary array. Figure 4.17 illustrates how an array of a bytes is divided into chunks of fixed size c bytes which are then assigned to processes in a round-robin fashion. In the Change phase, each process modifies the bytes in their respective chunks. A barrier is used to separate the Change phase from the next Read

phase, where the round-robin distribution used in the previous Change phase is shifted such that, had the array been a shared data array, each process would now read the Chunks previously modified by their neighbours. In the case of a shared data array, the size of the array a is also the total number of modified bytes that must be consistently observed in each Read phase. Thus, this number represents the *memory consistency workload* in bytes that the underlying memory system must handle.

MCBENCH was used to study the scalability of the MCM options in relation to the number of processes (n). Scalability with respect to n is especially relevant for SDSM systems because the primary means for increasing computational power in distributed memory systems is to add more computing nodes.

The chunk sizes (c) used are 4, 2048, and 4096 bytes in length; and the array size (a) is 4MB for all measurements. The 4-byte chunk size represents a worst case scenario as all processes will modify every page (different portions) in each Change phase. As a consequence, each process will have to perform memory consistency tasks for the whole array even though it only modified $1/n$ of the array. The 2048-byte chunk size is exactly half a page size, and each page will be modified by two processes in each Change phase (the array is aligned to page boundaries). This chunk size is chosen with the intent to investigate the impact of multiple-writer home-migration. The 4096-byte chunk size is exactly the size of a page. As such this case is expected to be favourable to MCMs in the Write-set and Scalable Barrier groups.

4.6.1 MCBENCH Results: 4-byte Chunk Size

Figure 4.18 shows MCBENCH results where modifications are made to 4-byte chunks. Costs scale as $O(\frac{n-1}{n})$ for all MCMs; where n is the number of processes. This scaling arises because each process accesses all pages and $\frac{n-1}{n}$ of the pages are remote pages (home-migration is not possible since pages are always modified by their home processes).

Although scaling is the same for all MCMs, the Invalidate All MCM exhibits the highest costs. This is due to the BARRIER call that separates the Read phase in one iteration from the Change phase of the next iteration. During this barrier, the Invalidate All MCM invalidates all remote pages, including those that processes have read during the Read phase. In contrast, the other MCMs recognize that no remote pages were modified during the Read phase and so they do not invalidate any remote pages. The significance of this is realized in the Change phase that follows. For the Invalidate All MCM, the invalidated remote pages would have to be fetched from their respective page homes before they may be modified. For the remainder of the MCMs, the remote pages are already valid and zero page fetch operations occur during the Change phase. In other words, the Invalidate All MCM cost is higher as page fetches

Figure 4.17: MCBENCH. An array of size a bytes is divided into chunks of c bytes. The benchmark consists of Change and Read phases that can be repeated for multiple iterations. Entering the Change phase of the first iteration, the chunks are distributed to the available processes (four in this case) in a round-robin fashion. In the Read phase after the barrier, each process reads from the chunk that its neighbour had modified during the previous Change phase. This is followed by a barrier which ends the first iteration. In the Change phase of the next iteration, processes modify the same chunks that was read in the previous Read phase. That is, shifting of the chunk distribution only occurs when moving from a Change phase into a Read phase.

Figure 4.18: MCBENCH results using a chunk size of 4 bytes and a 4MB array.

Figure 4.19: MCBENCH results using a chunk size of 2048 bytes and a 4MB array.

are performed for both Change and Read phases; whereas in the other MCMs, page fetches are only performed during a Read phase.

4.6.2 MCBENCH Results: 2048-byte Chunk Size

Figure 4.19 shows MCBENCH results when modifications are made to 2048-byte chunks. Since there are two chunks per page and a round-robin distribution is used to map chunks to processes, $\frac{2}{n}$ pages will be accessed by each process. This results in $O(\frac{1}{n})$ scaling. As with the 4-byte chunk measurements, the cost of the Invalidate All MCM is the highest. This is due to the same reason (invalidating pages which have not been modified by another process). No cost differences between single-writer and multiple-writer MCMs are observed.

Figure 4.20: MCBENCH results using a chunk size of 4096 bytes and a 4MB array.

4.6.3 MCBENCH Results: 4096-byte Chunk Size

Figure 4.20 shows MCBENCH results where modification are made to 4096-byte chunks. Similar to the previous measurements, the Invalidate All MCM performs the poorest; and for the same reason that remote pages read during Read phases are invalidated unnecessarily before entering the next Change phase. In contrast, the Selective Invalidation MCM keeps such pages valid and therefore avoids half of the page fault penalties incurred by the Invalidate All MCM. This explains why the costs for the Selective Invalidation MCM is roughly half of those for the Invalidate All MCM.

The 4096-byte chunk size also matches the SDSM page size. Moreover, as the array is aligned to page boundaries, each page in the array will be modified by at most one process during the Change phases. For the MCMs capable of home-migration, this so called single-writer situation is exploited by migrating each page to the process that modified it. Migrating a page eliminates the need to diff that page and therefore lowers the memory consistency costs. Despite this, results show that the Selective Invalidation MCM (which does not perform home-migration) performs better when the number of processes is less than five. The reason for this is that MCBENCH is designed to generate data dependencies and so pages are always migrated to the “wrong” processes. Thus, while home-migration capable MCMs avoid diffing costs, a penalty is paid as processes have to fetch every page accessed during Read phases. In contrast, Selective Invalidation MCMs access remote pages only $\frac{n-1}{n}$ of the time; although, each remote page that is accessed will not only have to be fetched, but also diffed.

Using the cost models for page fetch (RUM-Fault) and diffing in equations 4.2 and 4.3 respectively, the estimated cost for each page fetch is ~ 156 microseconds and that for each page diffed is ~ 48 microseconds. Thus, the ratio between costs for the Selective Invalidation MCM and that of a home-migration capable MCM can be expressed as

follows:

$$\frac{n-1}{n} \times \frac{156+48}{156}.$$

By solving for n such that the value of the whole expression is 1 gives an estimate of when the performance of home-migration capable MCMs will overtake Selective Invalidation MCM. In this case, $n = 4.25$. This is close to the crossover point in figure 4.20; which is at 5 or 6 processes.

4.6.4 Conclusions for MCBENCH

The results presented for the three cases above indicate that the Invalidate All MCM is the poorest performer. This is due to invalidation of all remote pages regardless of whether they need to be invalidated. In this way, the additional data bookkeeping and communication costs incurred by the other MCMs are justified.

There is no discernible difference between costs for the home-migration capable MCMs. This suggests that scalability differences between migration protocols that use write-sets or scalable communication are overshadowed by coherency costs from diffing and page fetch. However it should be noted that the objective of home-migration is to reduce the need for such coherency action. Thus, the more successful home-migration is in reducing coherency costs, the more significant the scalability differences would become.

4.7 Critical Path Performance Model

The performance of an SDSM application can be rationalized by using a performance model that applies cost models to SDSM operations (e.g. number of page fetches, diffs, etc.). In [20], we described two performance models that were driven by various page fault counts. Both performance models represented the time required for parallel execution as the sum of two parts:

- *Computation part:* This was modeled as the sequential time (T_{seq}) divided by the number of processes (n) (problems were assumed to be load balanced):

$$\frac{T_{seq}}{n}$$

- *Overhead part:* This was modeled in two ways. One was based on aggregated page fault counts, while the other used the concept of a critical path [38, 62] to select only the page faults that contributed to overheads.

- The *aggregated model* first calculates a sum of total overheads by multiplying each aggregated fault count by their respective cost coefficients. For example,

if there were n_{fetch} fetch faults and n_{write} write faults in total (i.e., summed across all processes), then the total overheads would be:

$$n_{fetch}C_{fetch} + n_{write}C_{write}.$$

A factor f that represents the proportion of the overhead that occurs serially was then used to tune the overheads in a similar fashion to Amdahl's Law [8]:

$$(n_{fetch}C_{fetch} + n_{write}C_{write}) \left(f + \frac{1-f}{n} \right).$$

- The *critical path model* views parallel execution as a series of “parallel regions” that are separated by barrier synchronization. Fault counts are sorted both by region and by process. Since computation is assumed to be load balanced, it is possible to determine which process spent the longest time in any given region. Fault counts for that process therefore contribute to the critical path. As such, the overhead cost for the critical path model is represented as follows:

$$\sum_{r \in R} \max_{p \in P} (n_{fetch,r,p}C_{fetch} + n_{write,r,p}C_{write});$$

where R is a set of parallel regions and P is a set of processes.

The critical path model was found to be more accurate than the aggregated model; however, it was also more difficult to apply.

This section describes an extended critical path performance model that builds on the previous model in the following ways: 1) it does not require programs to be load balanced; and 2) the cost models for all MCM operations developed here (FAULT, FLUSH, and BARRIER) are included. The concept of parallel regions separated by barrier synchronization is retained. As such, the extended critical path performance model describes parallel execution time as follows:

$$T_{par} = \sum_{r \in R} \max_{p \in P} (T_{reg,r,p}) + \sum_{b \in B} T_{bar,b}; \quad (4.23)$$

where T_{par} is the execution time for a parallel program; $T_{reg,r,p}$ is the execution time of parallel region $r \in R$ at process $p \in P$; and $T_{bar,b}$ is the time spent in barrier b . In turn, the execution time of a parallel region is modeled as the time spent executing application code and that spent executing SDSM operations. This is expressed as follows:

$$T_{reg} = T_{app} + \sum_{d \in D} T_{dsm,d}; \quad (4.24)$$

Figure 4.21: The duration of this parallel region is longest at P0 even though it performs substantially fewer SDSM operations than P1.

where T_{app} is the time spent executing application code; and $T_{dsm,d}$ is the time spent executing SDSM operation $d \in D$, which can be either a FAULT or FLUSH operation.

The formulation for T_{reg} in equation 4.24 above indicates that the parallel region with the longest duration cannot be solely determined by comparing costs of SDSM operations alone. Figure 4.21 illustrates this with two processes executing parallel regions between two barriers. From the figure, process P1 spends substantially more time executing SDSM operations than process P0; yet the time spent in application code at P0 is long enough for the parallel region at P0 to be the critical path. In this way, application of the extended critical path performance model is more complicated than the original critical path performance model detailed in [20].

To overcome the complexity of applying the extended critical path performance model directly, the viability of this approach was accessed by extending the Danui SDSM to include a concept of *Modeled Time*. This was implemented by providing a special timing call: `DSM_Time`. Each call to `DSM_Time` returns a wall-time that has been adjusted by replacing actual time spent in SDSM operations with modeled times. For example assume that an actual execution time (T) is composed of the following application and SDSM operation costs:

$$T = a_1 + d_1 + a_2 + d_2 + d_3 + d_4 + a_3;$$

where a_i are costs related to application code, and d_j are costs related to SDSM operations. Then the corresponding modeled time (T') would be expressed as:

$$T' = a_1 + d'_1 + a_2 + d'_2 + d'_3 + d'_4 + a_3;$$

where each d'_j is the modeled time for SDSM operation d_j . Modeled Time can be used to evaluate the accuracy of the performance model by executing the application code once to obtain the actual time, and then a second time to obtain the modeled time. The accuracy of the model will then be given as $\frac{T'}{T}$, where a value of 1.0 implies a perfectly accurate model, a value greater than 1.0 implies an overestimate, and a value less than 1.0 implies an underestimate.

Using this approach, the accuracy of the SDSM performance models developed in this chapter are evaluated for two different application benchmarks. The first is MCBENCH [71] which was introduced previously in section 4.6. This benchmark is used as a synthetic example with easily understood behaviour. The second is a 2D Fast Fourier Transform (FFT) which was ported from the SPLASH-2 [73] benchmarks. This benchmark is used as an example of an actual parallel application kernel.

4.7.1 Performance Model Accuracy: MCBENCH Case

MCBENCH [71] is a memory consistency benchmark designed to measure memory consistency overheads of shared memory systems. It was described previously in section 4.6. It involves accessing memory in alternating Change and Read phases such that data dependencies are repeatedly generated.

For the purpose of evaluating the extended critical path performance model, MCBENCH is executed once using actual timing calls and a second time using the Modeled Time extension described earlier. Only one representative MCM from each of the three barrier groups was selected for use in this test. The Basic Barrier group is represented by the Invalidate All MCM, the Write-set Barrier group is represented by the Write-set Lowest-ID MCM, and the Scalable Barrier group is represented by the Scalable Multiple-writers MCM.

Table 4.5 lists the accuracy ratios for MCBENCH using various chunk sizes and number of processes. Chunk sizes selected are not multiples of the SDSM page size so that a greater variety of page faults and SDSM coherency actions would be taken. The results indicate that the modeled time produces estimates that are within 10 percent of the actual timings.

Table 4.5: Accuracy ratios for MCBENCH using different chunk sizes (in bytes) and number of processes. The array size used is determined by multiplying the chunk size by the number of processes.

Chunk Size	Basic		Write-set		Scalable	
	$P = 2$	$P = 4$	$P = 2$	$P = 4$	$P = 2$	$P = 4$
50000	1.00	0.97	1.05	1.04	1.08	1.08
100000	0.99	0.99	1.08	1.04	1.06	1.03
150000	1.03	0.97	1.08	1.03	1.04	1.03
200000	0.96	0.96	0.98	1.07	1.02	1.01
250000	0.94	0.95	1.03	1.00	1.04	0.99
300000	0.95	0.99	1.00	1.03	1.02	1.02
350000	0.90	0.96	1.04	1.04	1.01	0.99
400000	0.99	0.96	1.03	1.04	0.98	1.03

4.7.2 Performance Model Accuracy: 2D Fast Fourier Transform Case

The 2D Fast Fourier Transform (FFT) benchmark from the SPLASH-2 [73] suite was modified to use the SDSM interface. The benchmark is an implementation of Bailey’s [12] “six step” Fast Fourier Transform (FFT) algorithm and accepts $N = 2^M$ complex data points in the form of a $\sqrt{N} \times \sqrt{N}$ matrix. (For the SPLASH-2 and this derivative implementation, M is required to be a positive even number, as this simplifies the calculation of $\sqrt{N}$ to $2^{M/2}$.) The six steps of the FFT are:

1. (Transpose.) The input matrix is transposed into a scratch matrix.
2. (FFT.) FFT is performed on the scratch matrix.
3. (Multiply.) The resulting matrix is multiplied by $e^{\pm 2\pi ijk/n}$.
4. (Transpose.) This will write back into the input matrix.
5. (FFT.) FFT is performed on the input matrix.
6. (Transpose.) Final transpose into the scratch matrix.

The six steps can be classified as either computation or transpose steps.

Computation steps, 2, 3, and 5, consists of 1D FFTs and multiply operations. All computation steps operate on values down matrix columns. As such, work sharing for computation steps is achieved by assigning each process a vertical band of columns, the width of which is $\sqrt{N}/P$; where P is the number of processes. This is matched by storing matrices in column-major form so that computation will access memory sequentially. Some padding is also inserted between the columns (in memory) such that each shared page holds data for at most one column. Figure 4.22 illustrates this arrangement. This padding helps to avoid contention for pages at the start and end of

Figure 4.22: Padding is inserted between memory allocated for columns such that no shared page ever holds the contents of more than one column. This avoids page contention.

each column. For example, if the last column for P0 is k and the first column for P1 is $k+1$, then contention is avoided as padding pushes the start of column $k+1$ out of page $j+1$ into page $j+2$.

The transpose operations are performed by copying data points from the source matrix into their respective transposed positions at a destination matrix. The implementation uses a scratch matrix together with the input matrix to perform transpose operations. Figure 4.23 shows how data moves between the input (left) and scratch (right) matrices at each transpose step; with computation always operating on the most recent destination matrix. The workload is divided by assigning each process a band of rows from the source matrix that maps onto a transposed band of columns at the destination matrix; such that the columns correspond to work sharing for computation steps. Thus, process synchronization is only required before a transpose, but not after.

Modeled Time is used to evaluate the accuracy of the extended critical path model for the FFT benchmark. As for MCBENCH, only one representative MCM from each of the three barrier groups was selected for the tests. The Basic Barrier group is represented by the Invalidate All MCM, the Write-set Barrier group is represented by the Write-set Lowest-ID MCM, and the Scalable Barrier group is represented by the Scalable Multiple-writers MCM.

Table 4.6 lists the accuracy ratios for the FFT application when using different matrix sizes (2^M). The results indicate that the modeled time produces estimates that are within 17 percent of the actual timings.

Table 4.7 lists the accuracy ratios for FFT barrier and transpose operations. These are plotted in figures 4.24 and 4.25 for barrier and transpose operations respectively. The accuracy for the FFT barrier operations show greater volatility than those for transpose operations.

Figure 4.23: Data movement between the input and scratch matrices (shown on the left and right halves respectively). Step 1 of the algorithm transposes the input matrix by copying data points to the scratch matrix. In steps 2 and 3, processes perform computation on their assigned columns of the transposed matrix (right). A barrier is required before step 4 where the computed values in the scratch matrix is transposed to the original input matrix (left). Step 5 follows with more computation along the columns, and then a barrier is used before the final transpose (step 6).

Table 4.6: Accuracy ratios for FFT for different problem sizes (2^M data points).

M	Basic		Write-set		Scalable	
	$P = 2$	$P = 4$	$P = 2$	$P = 4$	$P = 2$	$P = 4$
10	0.95	0.87	1.03	0.89	1.11	1.13
12	0.95	0.83	1.02	1.06	1.07	1.09
14	0.94	0.97	0.96	0.88	0.96	0.94
16	0.95	0.87	0.96	0.89	0.92	0.90
18	0.83	0.86	0.90	0.91	0.93	0.89

Table 4.7: Accuracy ratios for Barrier and Transpose results for FFT using different problem sizes (2^M data points).

— Barrier —						
M	Basic		Write-set		Scalable	
	$P = 2$	$P = 4$	$P = 2$	$P = 4$	$P = 2$	$P = 4$
10	0.83	0.81	1.15	0.88	1.18	1.29
12	0.77	0.65	1.03	1.4	1.12	1.28
14	0.83	0.8	1	0.67	1.07	0.92
16	0.87	0.84	0.99	0.9	0.92	0.83
18	0.88	0.67	0.99	0.85	0.8	0.71
— Transpose —						
M	Basic		Write-set		Scalable	
	$P = 2$	$P = 4$	$P = 2$	$P = 4$	$P = 2$	$P = 4$
10	0.99	0.9	0.94	0.88	0.89	0.89
12	0.99	0.88	1	0.94	1	0.96
14	0.95	0.99	0.94	0.94	0.91	0.92
16	0.96	0.87	0.95	0.88	0.91	0.9
18	0.82	0.87	0.87	0.91	0.95	0.9

Figure 4.24: Accuracy ratios for FFT barriers.

Figure 4.24 shows that the volatility in the FFT barrier estimates occurs for problem sizes up to $m = 14$, beyond which the results converge. However, this convergence appears to be lower than 1.0; which indicates that the cost models are generally underestimating.

Figure 4.25 shows that volatility is less of an issue for the transpose operations. Although, as with the FFT barrier, the results indicate that the cost models are slightly underestimating the true cost.

4.7.3 Conclusions for Critical Path Performance Model

The cost functions derived in this chapter for the various SDSM operations were evaluated by incorporating them into a critical path performance model. The critical path performance model was built into the SDSM as a special timing call (`DSM.Time`) that provided wall-times that had been adjusted by the various cost models. Using this “Modeled Time,” the accuracy of the cost functions and critical path performance model were evaluated by measuring both the actual and modeled times when executing the MCBENCH and 2D FFT benchmarks for a variety of problem sizes.

Results indicate that modeled time gives estimates for MCBENCH that are accurate to within 10 percent. However, accuracy is lower for the 2D FFT benchmark at about 17 percent.

Further analysis on barrier and transpose operations in the 2D FFT benchmark

Figure 4.25: Accuracy ratios for FFT transpose.

indicate that modeled time tends to underestimate and greater volatility is observed for the barrier operations. A possible reason for this is that the cost models do not consider the effect of contention. Contention arises when two or more processes send events to the same “server” process concurrently. However, the server process can only handle one event at a time, resulting in delays.

4.8 Summary

This chapter presented detailed cost models for the Danui SDSM operations.

Latencies for sending events using the different options provided by the Event module were measured. It was found that events are small enough so that the size of the parameters do not have significant impact on latency. The latency for event communication on the evaluation system averaged 62 microseconds. A “send-and-forget” cost was also considered where only the time of the send operation at the sender was measured (i.e., the event would still be “in-flight”). This averaged 9 microseconds.

Cost models for lock synchronization operations were then described for the two different implementations of the lock interface in terms of event counts (table 4.3 p.88). These models were developed for two different lock usage patterns; described in terms of whether there was low or high contention for a lock. Benchmarks were developed to verify these models. Care was taken to eliminate the time spent waiting in blocked

SET-LOCK operations during high contention; as these are synchronization costs, not lock costs. The results presented in table 4.4 (p.91) verified the correctness of the cost models.

Cost models for the Heap module were presented for the FAULT, FLUSH, and BARRIER operations.

The FAULT operation is described by individual cost models for LRO, RUM, and RRO page faults. Timing measurements showed that the most expensive of the three was the RUM fault, with RRO being the next most expensive, and LRO the least expensive. Also, the costs of RUM and RRO faults increased as page size increased, while the costs for LRO faults remain constant.

The FLUSH operation is described by combining cost models for diffing, invalidation, and acknowledgement. The cost of diffing was found to be dependant on the page size, the number of pages diffed and sent, and the total amount of diff data sent. For invalidation, a cost is incurred per page invalidated. A constant cost was used to represent the acknowledgement cost on the evaluation system.

The BARRIER operation is supported by seven different implementations. These were classified into three barrier groups: Basic, Write-set, and Scalable. Models for each were presented and coefficients obtained using a variety of techniques (different for each barrier type). In studying these barriers, cost models for collective MPI operations were also presented and characterized on the evaluation system.

A study of barrier costs was presented that showed that the Scalable barriers provided lower synchronization costs than the Write-set barriers. However, it was shown that the amount of diffs sent during a barrier has a far greater impact on barrier costs. This suggests that paying the slight increase in overheads for coordinating home-migration is worthwhile.

A memory consistency benchmark called MCBENCH [71] was introduced. The benchmark was used to compare the seven MCM implementations. Results show differences between Invalidate All MCM, Selective Invalidation MCM, and the home-migration capable MCMs. Little differences were observed among the home-migration capable MCMs.

Finally, a critical path performance model was introduced that incorporated the cost models of the various SDSM operations. The critical path model was evaluated by building into the SDSM a special timing call (`DSM.Time`) that returned wall-times that have been adjusted by the cost models of the SDSM operations. This was used to compare the accuracy of the critical path performance model on MCBENCH and a 2D FFT benchmark. Results show that accuracy is better for MCBENCH (10 percent) than on the 2D FFT (17 percent).

Chapter 5

Hybrid Programming Model

The Danui Software Distributed Shared Memory (SDSM) system outlined in chapter 3 provides an abstraction that enables the use of a shared memory programming model across distributed memory architectures. This is desirable from a usability standpoint as the programmer is not required to coordinate data transfers between processes; as would be the case if the message passing (MP) programming model were used.

Usability comes at a cost, however. Page-based SDSM systems such as Danui, TreadMarks[43, 42, 9], and SCASH[35] make use of the `mprotect` system call to trap accesses to invalid or read-only pages. Twinning-and-diffing is a technique for supporting multiple writers to shared pages and solves the false sharing problem; but requires copies of shared pages, called twins, to be created before shared pages can be modified. These SDSM activities, and others, lead to trade-offs between usability and performance.

Given this trade-off, it makes sense to limit the use of SDSM systems to only the situations where shared memory accesses are too complex for a message passing implementation. Zhu *et al.* [79, 78] introduce such a system where arrays in an SDSM program are sorted according to memory access patterns into those that require assistance from the SDSM, and those that do not. For the arrays that do not require SDSM assistance, message passing is used to resolve data dependencies between processes. Zhu *et al.* note, however, that access patterns for some arrays change during the course of a program's execution. In such cases, the SDSM is used to manage the array; to the detriment to performance for the parts of the program that do not require SDSM assistance to manage the array. This problem can be solved by using an SDSM system that supports the use of message passing on arrays that are managed by the SDSM, and this chapter describes how the Danui SDSM is extended to provide such support.

In Danui, data dependencies are detected and resolved using a combination of the following mechanisms: twinning, diffing, invalidation, and page-fetch. When a process makes a modification to a remote page, twinning is used to make a copy of data in a

remote page before modifications are permitted. During a barrier operation, diffing is used to compare the data in a remote page with its twin to determine where changes have been made. These differences are then sent to the home process for that page, where they are applied to the master copy. Any other remote copies of the page at other processes are also invalidated. After the barrier, performing a read access on this shared page will result in a page fault that is resolved by fetching a copy of the page from the home process. This copy will contain the modifications made by the first process. This discussion highlights three areas where performance can be improved by using a shared memory and message passing (SMMP) programming model:

1. A writer process can send updated values of a shared region directly to a receiver process. This saves on diffing and page-fetch costs.
2. A writer process can send arbitrarily sized messages to a receiver. This saves on page fetch operations that operate at page sized units.
3. A sender and a receiver process can synchronize implicitly by message passing communication. This achieves finer grained synchronization when compared to barrier operations that synchronize all processes.

Section 5.1 presents an analysis of memory access side-effects from message passing operations usage in relation to SDSM flush and barrier operations, classification of communication cases, memory consistency issues, and observations that lead to a usable solution. Section 5.2 describes how the Danui SDSM is extended to support the SMMP programming model. Section 5.3 evaluates the performance of this implementation. Finally, section 5.4 summarizes the chapter.

5.1 Analysis for SMMP Programming

This section provides background analysis relevant to developing an integrated SMMP programming model. The section begins with a discussion on memory access side-effects and how these determine when/where message passing operations can be used in an SDSM program. Following this discussion, point-to-point communication is classified according to where message buffers reside in memory. This classification is necessary as using message passing in a shared memory context gives rise to the possibility that both send and receive message buffers logically refer to the same part of shared memory. Issues that arise from this form of communication are described and a correct, but inefficient, solution is given. Finally, observations about the semantics of message passing programming and its interaction with shared memory semantics are presented. It is shown that these observations allow for a more efficient integration of the SDSM and message passing programming environment.

5.1.1 Memory Access Side-effects

Message passing operations have memory access side-effects. Sending data from a message buffer, for example, requires the sender to read from the message buffer. Conversely, receiving data into a message buffer requires the receiver to write to that message buffer.

The period during which memory access side-effects may occur is determined by whether blocking or non-blocking communication calls are used. When using a blocking call, memory access side-effects occur only during the duration of the call; but when using a non-blocking call, memory access side-effects may occur anytime from when the call is made until some future time when the underlying communication initiated by the call completes.

The potential for memory access side-effects to occur while outside of a communication call leads to the possibility that such side-effects may occur during SDSM flush and barrier operations. This is problematic because flush and barrier operations assume that no memory accesses can occur during these operations.

For the implementation presented in this work, the following restriction applies:

Message passing memory access side-effects must not overlap with SDSM flush and barrier operations.

Figure 5.1 illustrates four cases of message passing usage supported by the SMMP programming model developed here. Details are as follows:

- Case I shows a blocking send from P0 that is matched by a blocking receive at P1.
- Case II shows a non-blocking send from P0 that is matched by a blocking receive at P1. As the send call at P0 is non-blocking, its memory read side-effect may extend beyond the return from the call (this is indicated by the elongated box with dotted lines). This lasts until P0 returns from its wait call.
- Case III shows a non-blocking send from P1 where completion of the operation is implied by the completion of a blocking receive at P1. Other than this, it is similar to case II.
- Case IV shows a non-blocking receive that is posted before its matching send. Locally (at P0), memory write side-effects is assumed to begin when the non-blocking receive is posted and not when the matching send is posted.

Figure 5.1: Message passing use cases and their respective memory access side-effect durations. Case I: Blocking send (S) at P0 matching a blocking receive (R) at P1. Case II: Non-blocking send (iS) at P0 matching a blocking receive at P1. The non-blocking send at P0 is guaranteed to have completed by a wait operation (W). Case III: Non-blocking send at P1 matching a blocking receive at P0 followed by a blocking send at P0 matching a blocking receive at P1. The non-blocking send at P1 is guaranteed to have completed when its blocking receive has completed. Case IV: Non-blocking receive (iR) at P0 is matched by a blocking send at P1 occurring some time later. The non-blocking receive at P0 is guaranteed to have completed by a wait operation.

5.1.2 Classification of Point-to-point Communication

The Danui SDSM manages a part of a process' address space as shared memory. Any region outside of shared memory is not managed and is private to the process. In this section point-to-point communications are classified according to where send and receive message buffers reside in memory.

Although only point-to-point communications are classified, the discussion here is relevant to most collective operations. As long as participants of a collective operation can be organized into individual sender-receiver relationships.

Given two different memory types, private and shared, four point-to-point communication classes are possible. These are shown in figure 5.2 and expanded below:

- Case I: Private-to-private communication. This does not involve any shared memory and is not of concern to the SDSM.
- Case II: Private-to-shared communication. This involves shared memory only at the receiver process. The memory access side-effect for this case is a write access performed by the receiver to shared memory. The source of this data is external with respect to shared memory and the Danui SDSM takes this to be modifications to shared memory.
- Case III: Shared-to-private communication. This involves shared memory only at the sender process. The memory access side-effect for this case is a read access performed by the sender from shared memory. The Danui SDSM ensures that the data read is valid before allowing the communication to proceed.
- Case IV: Shared-to-shared communication. This case can be further classified into symmetric or asymmetric sub-cases depending on whether the receive buffer refers to the same or different part of shared memory respectively:
 - Symmetric Shared-to-shared: The memory access side-effects for this case are a read access at the sender and a write access at the receiver to the same logical part of shared memory.
 - Asymmetric Shared-to-shared: The memory access side-effects for this case are a read access at the sender and a write access at the receiver to different parts of shared memory.

The asymmetric shared-to-shared case can be rationalized as a data copy operation from one part of shared memory to another part of shared memory, and the Danui SDSM can treat them as such.

On the other hand, the symmetric shared-to-shared case is problematic as it involves two processes, sender and receiver, that read and write from the same logical part of

Figure 5.2: Point-to-point communication. Case I: Private-to-private communication; both buffers A and B are in private memory. Case II: Private-to-shared communication; buffer A is in private memory while buffer B is in shared memory. Case III: Shared-to-private communication: buffer A is in shared memory while buffer B is in private memory. Case IV: Shared-to-shared communication: buffer A is in shared memory and buffer B is also in shared memory. Two sub-cases arise depending on whether P1 receives into buffer A (symmetric) or buffer B (asymmetric).

Producer

$A[\dots] \leftarrow \dots$

$\text{Send}(A)$

Barrier

Consumer

$\text{Recv}(A)$

$A[\dots] \leftarrow A[\dots] + 1$

Barrier

Figure 5.3: Producer-consumer example involving symmetric shared-to-shared communication of shared memory region A. This region is modified also modified by the consumer after completing the receive operation. A barrier operation synchronizes the processes at the end.

shared memory concurrently. While this appears harmless, since the receiver writes the exact same data that the sender reads, serious problems concerning the SDSM's memory coherency mechanisms arise. These issues are discussed in the following sections.

5.1.3 Issues Arising from Symmetric Shared-to-shared Communication

This section describes issues arising from symmetric shared-to-shared message passing communication and presents two requirements for implementing symmetric shared-to-shared communication correctly.

Consider the producer-consumer example in figure 5.3 where symmetrical shared-

Producer	Consumer
$A[\dots] \leftarrow \dots \triangleright W1$	
$\text{Send}(A)$	$\text{Recv}(A) \triangleright W3$
	$\dots \leftarrow A[\dots] \triangleright R1$
$\text{Recv}(ack)$	$\text{Send}(ack)$
$A[\dots] \leftarrow \dots \triangleright W2$	
Barrier	Barrier

Figure 5.4: Alternate producer-consumer example involving a symmetric shared-to-shared communication and an acknowledgement message. Shared region A is modified twice by the producer, once at $W1$ and again at $W2$. The consumer only reads from A ; the acknowledgement message from consumer to producer ensures that this read is done before the producer makes further modifications to A . A barrier operation synchronizes the processes at the end.

to-shared message passing communication is used to transfer shared memory region A from the producer to the consumer. Problems arise when the consumer proceeds to modify the received data because such modifications would be in conflict with the modifications that the producer has made prior to sending the data to the consumer, i.e. the modifications at the producer have not been made globally visible via an SDSM flush or barrier operation. As a consequence, diffs from the producer will enter a data race with those from the consumer at the next barrier operation.

The remedy for this data race is to have the producer make its changes globally visible before sending the shared memory region A to the consumer. From the perspective of the SDSM, this ensures that the producer's changes *happen before* the consumer's modifications and any additional modifications to the region that may follow.

Changes can be made globally visible by having the SMMP implementation invoke the flush operation prior to the send operation. Unfortunately, such an implementation would not be efficient because the flushes performed by the producer would involve sending diffs back to various home processes (a shared memory region may span several pages). In addition, should the standard SDSM flush operation be used, the flushed pages would also be invalidated at the producer, forcing the read access associated with the subsequent send operation to fetch fresh copies of the just flushed pages. This would negate the benefit of using message passing since the consumer could have fetched the pages directly from the home processes. In effect, the data transfer between the producer and consumer processes would have been rendered pointless. The only benefit from using message passing in this instance would be the pair-wise synchronization of producer and consumer memory accesses; in which case a small acknowledgement message would have sufficed.

Assume for the moment that the producer makes its changes globally visible prior to sending messages using a flush operation. A second issue arises from the memory write side-effect of the message receive operation at consumer.

Consider the alternate producer-consumer example shown in figure 5.4. In this example, the producer modifies shared region *A* twice, once at W1 and again at W2 (as labeled in the figure). Between these writes, the producer sends *A* to the consumer, the consumer reads from *A* at R1, and an acknowledgement message is sent back to the producer; so that the second write at W2 will not interfere with the read at R1. However, receive operations also have the side-effect of writing to the receive buffer. As such, the receive operation at W3 is a write operation to *A*. Assuming that the consumer begins the example with a valid copy of *A*, the effect of the receive operation at W3 will be the creation of a twin based on the original copy of *A*. This copy would however be stale because the producer would have sent diffs prior to the send operation to make changes to *A* at W1 globally visible. Thus, at the next barrier, the producer would make diffs for modifications at W2, while the consumer would make diffs for “modifications” at W3. These diffs from the producer and consumer processes would therefore result in a race condition.

The solution for the race condition above is to recognize that the write effect at W3 of figure 5.4 is for updating the consumer’s view of *A*, and that the consumer has no actual intention of modifying shared region *A*. This situation is similar to write effects to shared memory when an invalid shared page is validated via a page fetch operation that transitions the newly validated page into the read-only state; not read-write. Thus, the same approach can be taken when receiving shared regions in symmetric shared-to-shared communication. Using this approach will ensure that the receive operation at W3 will not result in an erroneous twin creation, as well as eliminating the race condition at the barrier operation.

The solutions for the issues discussed above are summarized by the following requirements for symmetric shared-to-shared communication:

1. Prior to sending a shared region, modifications to the region must be made globally visible.
2. Data received should be treated as updates to shared memory, and not write accesses.

While the second rule can be accommodated by making received data read-only, the first rule presents a bigger challenge. Using the flush operation was described above as one way to satisfy the first rule. However, it was also noted that using the flush operation would result in substantial overheads that negate the benefits of integrating message passing into the SDSM programming environment.

5.1.4 Leveraging the Semantics of Message Passing Send and SDSM Operations

Modifications to a send buffer must be made globally visible prior to proceeding with a message passing send operation. However, using the SDSM flush operation to make such modifications globally visible is expensive. This section describes how the semantics of message passing send operations can be leveraged to allow for more efficient solutions. Some observations about message passing send operations are listed as follows:

- *Message Buffer:* The location of the region that will be accessed by the process as a result of the message passing operation is explicitly provided as a parameter to the message passing function. In contrast, the page fault mechanism that is used by the SDSM to detect accesses obtains only a fault address. The difference is that in the former, a *region* is provided; whereas in the latter, only an *address* is provided.
- *Read Access:* The memory region that is defined by the message buffer will be read by the sending process. In contrast, in an SDSM read accesses to valid pages are not noticed by the SDSM as no page faults are triggered. The fundamental difference is that in the former, notice of *read access* is given by the message passing interface; whereas in the latter notice of *access violations* are trapped/detected by the SDSM's memory coherency mechanisms.
- *Access Duration:* Message passing send operations have clearly defined durations when memory read accesses from the message buffer may occur. Blocking sends, for example, may read from the message buffer anytime from the start of the operation till it returns. Non-blocking sends may continue reading from the message buffer until appropriate events have occurred that indicate or imply that the send has completed. For example, completing a wait call or receiving an acknowledgement. (There is no contrasting SDSM case, since as noted above, SDSM coherency mechanisms detect access violations and not actual memory accesses.)

The information above leads to two observations:

1. The shared memory region define by a message buffer will be read; and
2. The duration of the read access can be determined from the message passing operation.

The significance of the first observation is that it indicates to a calling process that a region of shared memory will be read by the process. Since it is the programmer's responsibility to ensure that read accesses do not conflict with write accesses through

some form of synchronization (e.g. SDSM barrier operation¹), the SDSM can assume that no other process has modified the region since the last synchronization point.

The significance of the second observation is that the condition identified by the first observation, i.e. that no other process has modified the message buffer region, persists for the duration of the read access as defined by the message passing operation. Thus, it is safe for the SDSM to perform coherency tasks on the message buffer so long as the message passing operation does not return.

Leveraging this window of opportunity, the SDSM can check if the message buffer has been modified by the sender process; and if so, these modifications need to be made globally visible prior to sending the data (as per rule 1 described earlier in section 5.1.3). However, rather than depending on the flush operation, observe that the sender process is a writer to the message buffer, and that the first key observation above allows the SDSM to assume that no other process has modified the message buffer. Thus, the sender process is the only writer to the message buffer. We categorize this as a single-writer situation.

Single-writer situations have been used to justify home-migration in home-based SDSM systems [10, 27, 39, 34] (and this SDSM) as an alternative to sending diffs. The premise is that the state of a modified page at a single-writer process would be equivalent to the master copy with modifications applied. Therefore, costly communication from diffing in SDSM flush operations can be avoided by migrating the homes of the modified pages in the message buffer to the sender process.

The next section describes how these observations are used to implement extensions to the Danui SDSM to support the SMMP programming model.

5.2 The Danui SMMP Extension

The Danui SDSM library is extended to support the SMMP programming model for a subset of MPI-1 functions. These are classified into blocking point-to-point, non-blocking point-to-point, synchronization, and collective communications as follows:

1. *Blocking*: `MPI_Send` and `MPI_Recv` functions.
2. *Non-blocking*: `MPI_Isend`, and `MPI_Irecv` functions.
3. *Synchronization*: `MPI_Test`, `MPI_Wait`, `MPI_Waitall`, `MPI_Request_free`, and `MPI_Barrier` functions.
4. *Collective*: `MPI_Bcast` function.

¹The synchronization must also ensure that prior modifications made to a region are globally visible.

A crucial piece of information that MPI functions do not convey is the address of the remote message buffer. Understandably, this is not normally relevant for the distributed memory systems on which message passing is used; but for an SDSM system that supports an integrated programming environment, being able to tell if source and destination buffers are symmetric or asymmetric is necessary for determining the correct coherency action that the SDSM has to undertake. At present, the SDSM relies on a minor modification to MPI function names that programmers must use to indicate if message buffers are symmetric or asymmetric. For symmetric communication, an “s” prefix is used. For example, `sMPI_Send`, `sMPI_Recv`, `sMPI_Bcast`, etc. For asymmetric communication, an “a” prefix is used. For example, `aMPI_Send`, `aMPI_Recv`, `aMPI_Bcast`, etc. MPI functions that do not involve message buffers do not require prefixes; `MPI_Wait` for example. Using this interface, the producer-consumer example shown in figure 5.3 (p.132) can be written using symmetric blocking point-to-point communication as follows:

```
myid = DSM_Get_pe();
mpicomm = DSM_Get_comm();
A = (int*)DSM_Alloc_collective(sizeof(int)*N);
if (myid == PRODUCER)
{
    for (i=0; i<N; i++) A[i] = ...;
    sMPI_Send(A, N, MPI_INT, CONSUMER, TAG, mpicomm);
}
else (myid == CONSUMER)
{
    sMPI_Recv(A, N, MPI_INT, PRODUCER, TAG, mpicomm, MPI_STATUS_IGNORE);
    for (i=0; i<N; i++) A[i] = A[i] + 1;
}
DSM_Barrier();
```

The remainder of this section details the infrastructure extensions that were implemented to support blocking and non-blocking communication, and then describes the bookkeeping actions necessary for maintaining memory consistency. Details on how home-migration is performed in a one-sided manner is also given.

5.2.1 Infrastructure for Integrating MPI Calls

The general strategy used for supporting MPI is to wrap MPI calls between actions that modify the bookkeeping information maintained by the SDSM so that memory consistency is maintained. The structure for this is as follows:

1. Bookkeeping action,
2. MPI function call, and
3. Bookkeeping action.

For non-blocking MPI functions, returning from a call does not imply that the data transfer is complete. In such cases, bookkeeping actions that need to be executed when the data transfer is complete have to be delayed until the operation has completed. To support this case, a call-back mechanism is used to associate bookkeeping actions to `MPI_Request` handles. Then, when an `MPI_Request` has completed, as indicated by `MPI_Wait`, `MPI_Test`, or `MPI_Waitall`, the associated bookkeeping action is performed. (These MPI operations are overloaded to support this extra functionality.) Using the call-back mechanism, the general strategy for non-blocking MPI functions is as follows:

1. Bookkeeping action,
2. Non-blocking MPI function call, and
3. Register call-back function and parameters.

In some cases, it is not necessary to rely on `MPI_Request` handles to determine if a non-blocking communication call has completed. For example, for case III in figure 5.1 (p.130) completion of a non-blocking send can be implied by completion of the subsequent receive operation. For such cases, MPI provides the programmer with the `MPI_Request_free` function that frees an `MPI_Request` handle without waiting for its associated communication to complete. Once an `MPI_Request` handle has been freed, it is illegal to use it together with functions that check for completion of MPI operations (e.g. `MPI_Wait`). Hence it is not possible to rely on `MPI_Wait`, etc. to check for completion of non-blocking operations and execute its associated call-back function. To support use of the `MPI_Request_free` function, Danui redefines the `MPI_Request_free` function to move the freed handle (and its associated call-back information) into a set of freed handles. Handles in this set are then polled for completion using `MPI_Test` internally at the following points:

- after `MPI_Recv`,
- after `MPI_Wait`,
- after an `MPI_Test` that indicates completion of some handle,
- after `MPI_Bcast`, and
- after `MPI_Barrier`.

Polling is also performed before SDSM flush and barrier operations as message passing operations must not overlap with SDSM flush and barrier operations.

Although polling is performed only at the points listed above, it is still possible for the amount of overhead to become significant. For example, freeing many `MPI_Request` handles will increase the cost of polling; since many handles would have to be tested for completion. Another example is when only a few `MPI_Request` handles are freed, but many `MPI_Recv` calls are made. As each call performs polling, the costs can add up. It should however be noted that these examples assume that the non-blocking communication associated with freed `MPI_Request` handles never complete for several polling rounds; which may not be the case in an actual application.

5.2.2 Taxonomy of Bookkeeping Tasks

Danui is a home-based SDSM where shared memory is partitioned into pages, each of which has a home process. Five page states are used that combine locality (local or remote) and memory protection (unmapped, read-only, or read-write) attributes: Local Read-only (LRO), Local Read-write (LRW), Remote Unmapped (RUM), Remote Read-only (RRO), and Remote Read-write (RRW). Message buffers that are several pages large must undertake bookkeeping work for each page individually as each page can be in a different state.

Message buffers may not be aligned to page boundaries, and do not have to have lengths that are multiples of the page size. Thus, message buffers may include one or several pages either fully or partially. Figure 5.5 illustrates three example message buffers:

- Buffer *A* is a small buffer that fits entirely within a page and so only partially includes the page;
- Buffer *B* is aligned and fully includes several pages (3); and
- Buffer *C* partially includes the first and last pages, but fully includes all the pages in between.

The extent to which a page is included in the message buffer affects the bookkeeping work that is necessary.

The bookkeeping work necessary also depends on whether a message is being sent, or received; and if received, whether it is into a symmetric or asymmetric message buffer.

Bookkeeping work for each combination of page state and extent of inclusion by the message buffer is discussed below for message sends, receives into symmetric message buffers, and receives into asymmetric buffers.

Figure 5.5: Messages buffers (shaded areas) may include multiple pages (boxes) and do not have to be aligned with SDSM pages. Buffer *A* is a small buffer that fits entirely within the page, thus the buffer only includes the page partially. Buffer *B* is aligned and occupies exactly three pages, thus it includes three pages fully. Buffer *C* is a large buffer that partially includes the first and last pages and fully includes all the pages in between.

Table 5.1: Bookkeeping for Sending Messages.

State	Extent	Pre-bookkeeping	Post-bookkeeping
LRO	Full	-	-
	Partial	-	-
LRW	Full	-	-
	Partial	-	-
RUM	Full	Invoke fault handler.	-
	Partial	Invoke fault handler.	-
RRO	Full	-	-
	Partial	-	-
RRW	Full	Migrate page.	-
	Partial	Flush partial page if there are modifications.	-

5.2.2.1 Bookkeeping for Sending Messages

Table 5.1 lists the bookkeeping work that applies when messages are sent. The first two columns of the table lists the page state and the extent to which a page is included in the message buffer. These are followed by columns listing the necessary pre and post-bookkeeping work if any; where pre-bookkeeping work pertains to actions performed before a message passing operation and post-bookkeeping work pertains to actions performed when the message passing operation completes.

Pre-bookkeeping work is required for RUM pages as invalid pages cannot be read. These pages are validated by invoking the fault handler supplied by the Memory Consistency Manager (MCM) being used (see section 3.3). In other words, the bookkeeping work involved is the same as the SDSM read fault. If this occurs for many pages, then the advantages of integrating message passing will be significantly reduced. This, however, is not a huge concern as it is likely that a process that has been tasked to send

data will already have valid copies of that data.

Pre-bookkeeping work is also required for RRW pages as these may contain modifications. Recall that modifications made to the send buffer have to be globally visible before the send operation may proceed. The work required for RRW pages depends on the extent to which the pages are included in the message buffer.

RRW pages that are fully included in the message buffer may be migrated to the sending process as it is the only process that has modified the page (single-writer). Details of how home-migration is performed are presented in section 5.2.3.

RRW pages that are only partially included in the message buffer must be checked to determine if modifications were made to the region that is included in the message buffer. Checking a region involves searching for differences between the bytes of that region and its corresponding region in the twin copy (RRW pages have twin copies). No action needs to be taken if there are no modifications to the region. However, if there were modifications, then the included region is sent to the home process in full. Diffing is not required as this process is a single-writer with respect to that region. The page is also not invalidated as the purpose is to ensure that modifications are globally visible; moreover, as the process is a single-writer for the region there are no remote changes that need to be made locally visible. Finally, since the modifications have already been sent to the home process, the included region is copied to the corresponding part of the twin so that it reflects the correct state of the region at the home process.

5.2.2.2 Bookkeeping for Receiving Messages into Symmetric Buffers

Table 5.2 lists the bookkeeping work that applies when messages are received into symmetric message buffers. (Recall from section 5.1.3 that received data should be regarded as shared memory updates when the receive buffer refers to the same part of shared memory as the send buffer.)

Pre-bookkeeping work is required for partially included RUM pages as the receive operation updates part of the page. The full page, however, needs to be updated before the page can transition out of the RUM state. Otherwise future accesses to portions of the page that are not included in the receive buffer may be inconsistent. This is achieved by invoking the page fault handler for the RUM pages (this transitions the page into the RRO state).

Post-bookkeeping work is required for fully included RUM pages. These are transitioned into the RRO state after the receive buffer has been updated by the message passing operation. Post-bookkeeping work is also required for RRW pages. Those that are fully included in the message buffer will transition into the RRO state after the receive buffer has been updated by the message passing operation. Those that are only partially included in the message buffer will have their twins updated by copying the

Table 5.2: Bookkeeping for Message Receives into Symmetric Buffers

State	Extent	Pre-bookkeeping	Post-bookkeeping
LRO	Full	-	-
	Partial	-	-
LRW	Full	-	-
	Partial	-	-
RUM	Full	-	Transition to RRO.
	Partial	Invoke page fault handler.	-
RRO	Full	-	-
	Partial	-	-
RRW	Full	-	Transition to RRO.
	Partial	-	Update twin.

Table 5.3: Bookkeeping for Receiving Messages into Asymmetric Buffers

State	Extent	Pre-bookkeeping	Post-bookkeeping
LRO	Full	Invoke fault handler.	-
	Partial	Invoke fault handler.	-
LRW	Full	-	-
	Partial	-	-
RUM	Full	-	Migrate page.
	Partial	Invoke fault handler twice.	-
RRO	Full	-	Migrate page.
	Partial	Invoke fault handler.	-
RRW	Full	-	Migrate page.
	Partial	-	-

updated region of the page to the corresponding region in the twin.

5.2.2.3 Bookkeeping for Receiving Messages into Asymmetric Buffers

Table 5.3 lists the bookkeeping work that applies when messages are received into asymmetric message buffers. The general idea is to get pages into a read-writable state after the receive operation as receives into asymmetrical message buffers are write accesses.

Pre-bookkeeping work is required for LRO pages. For both full and partially included pages, the page fault handler for LRO pages is used to bring the pages into the LRW state. Pre-bookkeeping work is also required for RUM and RRO pages which are only partially included in the message buffer. For RUM pages, the fault handler is invoked once to bring the page into the RRO state (requires a page-fetch), and a second time to bring it into the RRW state. For RRO pages, the page fault handler only needs

to be invoked once to bring it into the RRW state. In both cases, transitioning from RRO to RRW will involve twin creation.

Post-bookkeeping work is required for all remote pages — RUM, RRO, and RRW — when they are fully included in the message buffer. For these page states, the page is migrated to the receiver process since it will have modified the full page and will therefore have the most up-to-date copy.

5.2.3 One-sided Home-migration

Home-migration is used in the SMMP extension as a technique for making modifications to shared memory globally visible when a single-writer scenario arises. In addition, home-migration also improves data locality and reduces the amount of data communication necessary in parallel applications by avoiding diff communication. However, unlike the barrier-based home-migration implementations presented in section 3.3 (p.55), these migrations are performed in a one-sided fashion.

The Forwarding Pointer technique described by Fang *et al.* [27] for one-sided home-migration is used. Each SDSM process maintains a page table that contains, amongst other details, home-pointers to the process ID of the home process for each page. Figure 5.6 illustrates how one-sided home-migration is performed. In part I, the initial structure of a home-tree formed by home-pointers at each process is shown. Process P0 is the root of the tree, indicating that P0 is the home process for the page. In part II, P1 claims the page from P0 by sending a “claim” request to P0. A response “ack” message is sent back in reply. Finally part III shows the result of this exchange, where home-pointers at P0 and P1 have been updated to point to P1. Other processes are not aware of this migration and continue to have their home-pointers pointing at P0. However, as home-pointers either point at the home process or one of the former home processes, messages bound for the current home process can be forwarded to the home process.

The state of the migrated page transitions into the LRW state at the new home process and into the RRO state at the former home process. The LRW state is used at the new home process so that other processes with remote copies of the page will invalidate those copies at the next barrier. The RRO state is used at the former home process because the page will have been in the LRO state before migration. Until a flush operation is called at that process, it is not necessary to prematurely invalidate the page by using the RUM state.

In some circumstances, it is possible for home-trees to extend such that they resemble linked-lists. When this occurs, messages bound for a home process may have to be forwarded by several intermediate processes; as such, the total communication latency for these messages will increase. This issue is addressed using heuristics for

Figure 5.6: One-sided home-migration for a page. Part I: Home pointers for the page at each process form a home-tree for that is rooted at P0, the home process. Part II: P1 claims the page from P0 by sending a “claim” request, to which P0 responds with an “ack” message. Part III: P0 and P1 have updated their home-pointers to point at P1 (the new home process), while P2 and P3 still maintain home-pointers that point at P0.

reducing the number of intermediate processes where possible. This is described in the subsection labelled “Path Compression” below.

Another way to reduce the cost of communicating messages within a home-tree is to increase the amount of information that is included in each message. This technique is used to implement a “claim” and “ack” handshake protocol, such that multiple pages can be claimed and acknowledged in the same message. This is described in the subsection labelled “Bulk Claim” below.

Finally, home-trees can be collapsed at BARRIER operations; such that home-pointers for a given home-tree all point directly to the root of the tree. This is described in the subsection labelled “Collapsing Home-trees” below.

5.2.3.1 Path Compression

The height of the home-tree determines the cost of sending messages to the home process. For instance, if P2 in figure 5.6 part III wishes to claim the page, the claim request would have to be sent to P0 first and then forwarded to P1. A heuristic is used in this case to have P0 update its home-pointer to P2 after it has forwarded the request to P1. This is possible because P2 will be the next home process. The result of this is to have P2 at the root of the tree, P0 and P1 pointing to P2, and P3 still pointing at P0.

Figure 5.7: Path compression is unable to shorten the height of the home-tree for this sequence of claims. P0 is initially the home, claims are then made by processes in the following order: P1, P2, P3, and P4.

This is only a heuristic and in some circumstances, a list of home-pointers may still be formed. Figure 5.7 illustrates an example where P0 is initially the home process. A linked-list sequence is formed when processes P1, P2, P3, and P4 claim the page in that order. This happens because each time a process in the set of “many processes” (the triangle in the figure) makes a claim, its request is forwarded up through P0 and the present root process. Path compression therefore only applies to these processes.

The chance of this scenario occurring is reduced if an application exhibits good data access locality (such that pages do not constantly migrate). Also, home-trees are collapsed at each BARRIER operation so the chances of them growing very large is remote. The subsection labelled “Collapsing Home-trees” below describes how this is performed.

5.2.3.2 Bulk Claim

When the message buffer of a message passing communication is large and spans many pages, then multiple pages may need to be migrated to the sender process. In this case, a direct implementation of the claim protocol would be costly as at least two message latencies per page are required; more if it involves forwarding of the request message through intermediate processes. Such an implementation would not be much better than performing a flush operation.

Unlike flush operations, however, request messages for the claim protocol may be aggregated as each claim is simply a message containing the process ID of the claiming process and a page number. Thus, a bulk claim can be made by including multiple page numbers in the same message. This reduces the number of messages required as requests for multiple pages can be sent in the same message.

When an aggregated request is received, the requests is acknowledged by sending

an “ack” message that contains the number of pages acknowledged. If this process is no longer home to some of the requested pages only those resident at this process are acknowledged. The remainder of the pages are re-aggregated and forwarded; possibly into smaller aggregations since home-trees for the pages may be different. (Although this is unlikely as all the pages are related in that they have been included in the same message passing buffer.)

Finally, the bulk claim completes once the sum of the “ack” page counts received at the claiming process equals the number of pages claimed.

5.2.3.3 Collapsing Home-trees

Home-trees are collapsed at the start of barrier operations to reduce the height of all home-trees back to one. Two arrays of home-pointers are maintained. The first array H is used to store the home-pointers that form the home-trees. The second array G is formatted as follows:

$$G[i] = \begin{cases} 0 & \text{if } H[i] \neq \text{this-process, or} \\ H[i] & \text{if } H[i] = \text{this-process.} \end{cases}$$

Collapsing is performed using `MPI_Allreduce` to sum G into H (overwriting the old H).

5.3 Evaluation of the Danui SMMP Extension

Bulk data transfer is one area where the use of message passing in SDSM programs is expected to be beneficial for performance, as whole data regions can be transferred rather than fetching and/or diffing one page at a time (as would be the case if relying on the SDSM coherency protocols). Scalable collective communication operations is another area where the use of the message passing model in SDSM programs is expected to be beneficial for performance, as these often scale logarithmically with the number of processes rather than linearly as would be the case if relying on the SDSM coherency protocols. Thus, a benchmark for measuring bulk data transfer latencies and an application that can benefit from broadcast communication (namely Dense Blocked LU) have been used to assess the effectiveness of the Danui integrated SMMP implementation.

5.3.1 Bulk Data Transfer SMMP Benchmark

The single-threaded latency benchmark from the Ohio State University MPI Benchmarks 3.1.1 (OMB-3.1.1) [2] described previously in section 4.1.1 (p.81) uses a “ping-pong” style of communication that bounces a message of some size between two pro-

Process P0 loop:	Process P1 loop:
1. $A[\dots] \leftarrow B'[\dots] + 1$	1. $B[\dots] \leftarrow A'[\dots] + 1$
2. <code>DSM_Barrier</code>	2. <code>DSM_Barrier</code>
3. Swap buffer pointers; A with A' and B with B' .	3. Swap buffer pointers; B with B' and A with A' .

Figure 5.8: Locality-aware, SDSM-only implementation. Four buffers A, A', B, B' are grouped as the “ A -buffers” and the “ B -buffers.” Process P0 modifies only the A -buffers, while process P1 modifies only the B -buffers. Pointer swapping is used so that processes never read from buffers that are being modified.

cesses repeatedly. This is adapted for measuring bulk data transfer costs in a shared memory environment by injecting data dependencies using a combination of memory write and read accesses.

Two contrasting memory access patterns, representing locality-aware and locality-unaware cases, were implemented. The *locality-aware* case uses multiple shared memory regions that are each modified by only one process, while the *locality-unaware* case uses one shared memory region that is modified by both processes in turns. For both cases, data dependencies are formed when one process reads what the other process has written. SDSM-only, SMMP, and MPI-only versions of both cases were implemented.

5.3.1.1 Locality-aware Bulk Data Transfer Benchmark

Steps for the SDSM-only implementation of the locality-aware bulk data transfer benchmark are shown in figure 5.8. Four shared memory buffers are used in total that are arranged as two pairs which are referred to as the “ A -buffers” and the “ B -buffers.” The A -buffers are modified only by process P0, while the B -buffers are modified only by process P1. In each iteration, a process will modify one buffer from its assigned pair using values from the buffer which its peer modified during the previous iteration. For example, process P0 writes into A using values from B' . This is followed by a `DSM_Barrier` synchronization and a buffer pointer swapping step; thus for each iteration, buffers A and B are the buffers that will be modified, and buffers A' and B' are the buffers that will be read.

Steps for the SMMP implementation are shown in figure 5.9. This is largely the same as the SDSM-only version except that the `DSM_Barrier` calls have been replaced by symmetrical shared-to-shared `sMPI_Send` and `sMPI_Recv` calls. These are used to send A from process P0 to process P1, and then to send B from process P1 to process P0. The effect of this is to update the views of A and B at processes P1 and P0

Process P0 loop:

1. $A[\dots] \leftarrow B'[\dots] + 1$
2. `sMPI_Send(A)`
3. `sMPI_Recv(B)`
4. Swap buffer pointers; A with A' and B with B' .

Process P1 loop:

1. $B[\dots] \leftarrow A'[\dots] + 1$
2. `sMPI_Recv(A)`
3. `sMPI_Send(B)`
4. Swap buffer pointers; B with B' and A with A' .

Figure 5.9: Locality-aware, SMMP implementation. This is same as the SDSM-only implementation with the exception that `DSM_Barrier` calls have been replaced by `sMPI_Send` and `sMPI_Recv` calls that serve to transfer data and synchronize processes. (The MPI-only version would be the same except that `MPI_Send` and `MPI_Recv` calls would be made instead.)

respectively.

Steps for the MPI-only implementation are the same as the SMMP implementation except that the SDSM library is not used, and the `MPI_Send` and `MPI_Recv` functions are used in place of `sMPI_Send` and `sMPI_Recv`.

For all three versions, the memory write and read operations stride through the buffers at SDSM page sized intervals (4KB), accessing only 1 byte from each page. This is to trivialize the impact of memory access times on the measured latency.

The benchmarks were executed on the evaluation platform detailed in table 4.1 (p.81). The home-migration capable Scalable Multiple-writers MCM described in section 3.3.5 (p.69) was used when executing the SDSM-only and SMMP implementations. Memory allocated for the buffers was arranged such that no SDSM page holds the contents of more than one buffer. This facilitates migration of each modified page to its respective writer process. Thus, all write accesses were performed on local pages (warm up iterations were used). Read accesses, however, are performed on remote pages and require page fetches. These fetches represent the generated data transfer which the benchmark seeks to measure.

Results for the locality-aware bulk data transfer benchmarks are shown in figure 5.10 for buffer sizes up to 4MB. Bulk data transfer costs are similar for the SDSM-only and SMMP versions when buffer sizes are smaller than the SDSM page size of 4KB. For buffer sizes of 4KB and larger, the SMMP version performs as well as the MPI-only version.

Results for the SDSM-only implementation are explained by noting the cost of page fetch operations. The number of pages fetched in each iteration is given by:

$$\left\lceil \frac{BufferSize}{PageSize} \right\rceil.$$

Figure 5.10: Results for the locality-aware bulk data transfer benchmarks. The Scalable Multiple-writers MCM was used to obtain results for the SDSM-only and SMMP versions.

Thus, latency for all buffer sizes up to the page size remains the same and only begins to rise when the buffer size exceeds the SDSM page size.

Results for the SMMP implementation can be explained in two parts: buffer sizes smaller than the SDSM page size, and buffer sizes of or larger than the SDSM page size.

When the buffer size is smaller than the SDSM page size, the partial extent bookkeeping work listed in table 5.1 applies to the `smPI_Send` calls, and the partial extent bookkeeping work listed in table 5.2 applies for the `smPI_Recv` calls. None of the bookkeeping work for partial extents involve home-migration. This implies that for buffer sizes smaller than the page size, no home-migration is performed for the SMMP implementation of the locality-aware benchmark. (No `DSM_Barriers` are used as well.) The default page placement is a round-robin distribution of page homes. Since 4MB buffers have been allocated, and because there are only two processes, the home process for the first page of all four buffers will be process P0. Thus, for `smPI_Send` operations at process P1, bookkeeping work for partial RRW pages has to be undertaken — “flush partial page if there are modifications” (table 5.1). This cost scales with the buffer size, hence it can be observed that the data transfer costs for sub-4KB buffer sizes increases slightly.

When the buffer size is of or larger than the SDSM page size, the full extent book-

Process P0 loop:	Process P1 loop:
1. $A[\dots] \leftarrow A[\dots] + 1$	1. -
2. <code>DSM_Barrier</code>	2. <code>DSM_Barrier</code>
3. -	3. $A[\dots] \leftarrow A[\dots] + 1$
4. <code>DSM_Barrier</code>	4. <code>DSM_Barrier</code>

Figure 5.11: Locality-unaware, SDSM-only implementation. A single shared buffer is used that is modified by both processes in turns.

Process 0 loop:	Process 1 loop:
1. $A[\dots] \leftarrow A[\dots] + 1$	1. -
2. <code>sMPI_Send(A)</code>	2. <code>sMPI_Recv(A)</code>
3. -	3. $A[\dots] \leftarrow A[\dots] + 1$
4. <code>sMPI_Recv(A)</code>	4. <code>sMPI_Send(A)</code>

Figure 5.12: Locality-unaware, SMMP implementation. This is same as the SDSM-only implementation with the exception that `DSM_Barrier` calls have been replaced by `sMPI_Send` and `sMPI_Recv` calls that serve to transfer data and synchronize processes. (The MPI-only version would be the same except that `MPI_Send` and `MPI_Recv` calls would be made instead.)

keeping work listed in table 5.1 applies for the `sMPI_Send` calls, and the full extent bookkeeping work listed in table 5.2 applies for the `sMPI_Recv` calls. Home-migration occurs and so `sMPI_Send` only operates on LRW pages and `sMPI_Recv` only operates on RRO pages. The tables indicate that no bookkeeping work is required for these cases. This explains why the SMMP implementation is able to match the performance of the MPI-only implementation.

5.3.1.2 Locality-unaware Bulk Data Transfer Benchmark

Steps for the SDSM-only implementation of the locality-unaware bulk data transfer benchmark are shown in figure 5.11. A single shared buffer is used that is modified by both processes in turn. `DSM_Barrier` calls are used to synchronize accesses on the shared buffer.

Steps for the SMMP implementation of the locality-unaware bulk data transfer benchmark are shown in figure 5.12. This is largely the same as the SDSM-only version except that `DSM_Barrier` calls have been replaced by symmetrical shared-to-shared `sMPI_Send` and `sMPI_Recv` calls.

Figure 5.13: Results for the locality-unaware bulk data transfer benchmarks. The Scalable Multiple-writers MCM was used to obtain results for the SDSM-only and SMMP versions.

Steps for the MPI-only implementation of the locality-unaware bulk data transfer benchmark are similar to the SMMP implementation except that the SDSM library is not used, and the `MPI_Send` and `MPI_Recv` functions are used in place of `sMPI_Send` and `sMPI_Recv`.

Results for the locality-unaware bulk data transfer benchmarks are shown in figure 5.13 for buffer sizes up to 4MB. Bulk data transfer latencies are highest for the SDSM-only implementation, and lowest for the MPI-only implementation. Latencies for the SMMP implementation lie in between latencies for the SDSM-only and MPI-only implementations, tending towards MPI-only latencies.

Results for the SDSM-only implementation show that latencies are steady for results below the 4KB buffer size, but increase beyond 4KB. This is explained by the cost of page fetch operations, as it was for the locality-aware benchmark previously. The slightly higher latencies in this case are because of an additional `DSM_Barrier` call per iteration.

Results for the SMMP implementation before the 4KB buffer size are flat and can be explained by the bookkeeping overheads for the “partial extent” case, as previously described. For buffer sizes 4KB and beyond, home-migration is possible. However, as the benchmark is locality-unaware, the pages are migrated at *every* `sMPI_Send` call. These migration overheads explain the gap between results for the SMMP implementation

Figure 5.14: Tiles are used to map blocks to processes. Tiling patterns for one to ten processes are shown as *a–j* respectively.

and those for the MPI-only implementation.

5.3.2 Parallel Dense Blocked LU Factorization

The SPLASH-2 [73] benchmark suite includes an implementation of a Parallel Dense Blocked LU Factorization algorithm. The defining characteristic of this LU algorithm is that the data layout and the computation are structured to maximize locality of memory accesses.

The algorithm divides an input matrix into a two-dimensional grid of blocks [74]. Blocks are assigned processes according to a tiling algorithm that tries to generate the most “square-like” block-to-process mappings. Figure 5.14 illustrates the tile shapes that would be used for one to ten processes. For example, four processes would use a 2x2 block tiling, and six processes would use a 3x2 tiling.

The tiling pattern used is purely a function of the number of processes and is not influenced in any way by the size of the matrix or the size of each block. As illustrated in figure 5.15, only a partial portion of the 2x2 tiles at the right column and bottom row is included in the input matrix. This is also true for blocks, as can be observed in the same figure.

Computation progresses by iterating through the diagonal blocks; beginning at the top left and ending at the bottom right diagonal blocks. Figure 5.15 shows the state when computation has progressed to the fourth diagonal block, which is labeled as the

Figure 5.15: A 2x2 tiling for four processes with partial blocks at the right and bottom edges, the smallest of which is the partial block in the bottom right corner. Computation iterates down the diagonal blocks. At each iteration, the perimeter blocks are blocks to the right and bottom of the diagonal. Blocks below and to the right of the perimeter blocks are interior blocks. The example is partially solved and the current diagonal, perimeter and interior blocks are indicated.

“current diagonal block.” Blocks to the right and bottom of the current diagonal block are called “perimeter blocks.” Blocks below and to the right of perimeter blocks (along the relevant dimensions) are called “interior blocks.”

Computation is structured along diagonal, perimeter, and interior blocks. For each diagonal block, iterating from top to bottom:

1. (Compute diagonal block.) The process owning the diagonal block performs computation. The remainder of the processes wait for computation to be completed in the synchronization below.
2. (Synchronize processes.) This synchronization is necessary as computation on perimeter blocks requires the results from the diagonal block computation.
3. (Compute perimeter blocks.) Processes with perimeter blocks perform computation. The remainder of the processes wait for computation to be completed in the synchronization below.
4. (Synchronize processes.) This synchronization is necessary as computation on interior blocks requires the results computed by perimeter blocks.
5. (Compute interior blocks.) Processes with interior blocks perform computation. The remainder of the processes proceed to the next iteration.

Note that no synchronization is required between the last step of one iteration and first step of the next iteration, even though computation for a “current diagonal block” requires the results of past interior block computation on that block. This is because each block is only modified by its owner process.

Given that processes only modify blocks which they have been assigned, data layout is optimized by allocating the memory required for blocks belonging to a process in one contiguous region. Address pointers to individual blocks in these regions are recorded in a block matrix. This is illustrated in figure 5.16. In this way, memory write access characteristics for each process are localized to their allocated region.

SDSM-only and SMMP implementations of the LU algorithm were implemented. These implementations are detailed below followed by SDSM profiling results showing the memory access characteristics of the algorithm and performance comparisons for the two implementations.

5.3.2.1 SDSM-only Implementation

The algorithm is implemented using the Danui SDSM in the manner described above. The contiguous memory region for blocks are allocated using the `DSM_Alloc_reserved` function, which ensures that the SDSM pages containing the allocated region are not

Figure 5.16: Blocks belonging to a process are stored together in a single contiguous shared memory region. A block matrix holds pointers to individual blocks (not all pointers are shown in the figure). Depending on the tiling pattern, block size, and problem matrix size, some blocks may be partial blocks.

used for any other allocation. Synchronization is performed using the `DSM_Barrier` function.

5.3.2.2 SMMP Implementation

The SDSM-only implementation is converted into an SMMP program by replacing synchronization steps with appropriate message passing operations:

- The first synchronization point is replaced by a `sMPI_Bcast` call on the diagonal block. For the processes that have perimeter blocks to compute, the `sMPI_Bcast` call provides synchronization that ensures that the write accesses at the block owner have completed before the diagonal block is read for computations on perimeter blocks.
- The second synchronization point is replaced by `sMPI_Isend` and `sMPI_Irecv` calls (non-blocking). Each process that has computed a perimeter block uses `sMPI_Isend` to initiate a send to the processes that require the block. The number of such processes depends on the shape of the tile. For example, using the tiling pattern for eight processes as shown in figure 5.14(h), and assuming that process P6 has computed a perimeter block, then:
 - `sMPI_Isend` would be used to send the block to P2 if the perimeter in question is the horizontal perimeter, or
 - `sMPI_Isend` would be used to send the block to P7, P4, P5 if the perimeter in question is the vertical perimeter.

Receiver processes issue matching `sMPI_Irecv` calls. Before using a perimeter block for the first time, `MPI_Wait` is used to ensure that the data has arrived. Each process needs to receive each perimeter block once. (Completion checks for `MPI_Request` handles associated with the `sMPI_Isend` calls are deferred to after the next `sMPI_Bcast`; all should have completed by that point.)

5.3.2.3 Memory Access Characteristics

The effect of optimizing data layout and computation can be observed by counting the number of write faults that occur to a shared page. LU was executed using a 1024×1024 matrix with 64×64 blocks and four processes. Write fault counts for this run are shown in figure 5.17. A separation of write faults by process is observed across all the pages. The triangular peaks are due to the way computation progresses down the diagonal blocks; which results in blocks towards the bottom and right of the input matrix being modified more than those towards the top and left.

Figure 5.17: Write fault counts for LU on a 1024×1024 matrix using 64×64 blocks and four processes.

5.3.2.4 Performance Comparisons

The SDSM-only and SMMP implementations of LU were executed for a 2048×2048 matrix using two to eight processes. Figure 5.18 shows the execution times for both implementations using 32×32 , 64×64 , 128×128 , and 256×256 blocks. It is observed that both implementations perform best for 64×64 blocks. The execution time for the SMMP implementation is also lower than the SDSM-only implementation, for the same number of processes and block size.

Figure 5.19 plots the speedup achieved by the SMMP implementation over the SDSM-only implementation. Speedups of about 15 percent are observed when there are two processes, increasing to about 25–30 percent when there are four processes, 30–35 percent for six processes, and 35–45 percent speedup when there are eight processes.

Table 5.4 lists a break down of the timing measurements for the case involving eight processes and using 64×64 blocks. The first column lists the steps in the algorithm. The next two columns list the cumulative time spent by the SDSM-only and SMMP implementations at each step respectively. The fourth column shows the speedup attained by each step individually. The fifth column shows the difference between the SDSM-only and SMMP timing measurements (SDSM-only time minus SMMP time). The last column shows the difference in percentage terms.

Comparing the cumulated computation times for Diagonal, Perimeter and Interior

Figure 5.18: Execution times for Dense Blocked LU Factorization using the SDSM-only and SMMP implementations for a 2048×2048 matrix. Block dimensions were 32×32 , 64×64 , 128×128 , and 256×256 .

Figure 5.19: Speedup of SMMP implementation over SDSM-only implementation for Dense Blocked LU Factorization. (Calculated as time taken by SDSM-only implementation divided by time taken by SMMP implementation.)

Table 5.4: Break down of timing measurements for the case that used eight processes and 64×64 blocks. Timings reported here were measured at process P0 and are cumulative for each step (Diagonal, Synchronization-1, etc.). For SMMP, Synchronization-1 is the time for `smPI_Bcast` and Synchronization-2 is the time for `smPI_Isend` calls. The biggest contributor to speedup is performance improvement for interior block computation. (Times are in microseconds.)

	SDSM-only	SMMP	Speedup	Difference	Percent
Diagonal	4808	4899	1.0	-91	0
Synchronization-1	657635	368783	1.8	288852	33
Perimeter	107126	94526	1.1	12600	1
Synchronization-2	219592	21171	10.4	198421	22
Interior	1921384	1532545	1.3	388839	44
Total	2910545	2021924	1.4	888621	100

steps (both SDSM-only and SMMP), it is seen that Interior is higher than Diagonal and Perimeter by several orders of magnitude. This is expected since each block is a diagonal or perimeter block once, but may be an interior block multiple times.

Comparing each step of the SDSM-only and SMMP implementations, *Diagonal block computation* takes the same amount of time as there are no differences in implementation. *Synchronization-1* is costly for both implementations, at around 23 percent of total time for the SDSM-only and 18 percent for the SMMP implementations. For the SDSM-only implementation, this time is spent performing memory consistency work for the interior blocks modified in the previous iteration. The Scalable Multiple-writers MCM is used. However, there will be zero write-notices since all processes write only to their assigned blocks, and only a few migration-notices that serve to invalidate copies of perimeter block pages. Most of the cost is instead attributed to the reduction (2 calls) and gather (2 calls) operations used by the MCM to coordinate propagation of write and migration notices. For the SMMP implementation, a single broadcast call is used to transmit computation results of the previous Diagonal step. Due to this payload (diagonal block data), the speedup observed is only 1.8. (For the 32 bytes block size case, speedup attained for this step is 3.0 as less data needs to be communicated.)

Perimeter block computation is slightly faster for the SMMP implementation mainly because the data for the diagonal block has already been made available. In contrast, the SDSM-only implementation fetches pages of the diagonal block as they are encountered. However, this does not result in much of a difference as there is only one diagonal block to fetch.

Synchronization-2 for the SMMP implementation is faster than the SDSM-only implementation by 10 times as non-blocking sends are used to share results of perimeter

blocks. Non-blocking receive calls for perimeter blocks required by the next step are also issued.

Interior block computation is about 1.25 times faster for the SMMP implementation compared to the SDSM-only implementation. This difference can be attributed to how results of the perimeter blocks are obtained. In the SDSM-only implementation, results are obtained by page fetch operations. In the SMMP implementation, results are obtained by calling `MPI_Wait` on the non-blocking requests issued previously. These calls are only made before using a perimeter block for the first time (during each Interior block computation step). Some overlap of communication and computation is achieved in this way.

In terms of contribution to overall speedup, the Interior step is the biggest contributor (44 percent), followed by Synchronization-1 (33 percent), and Synchronization-2 (22 percent).

5.4 Summary

This chapter describes how the Danui SDSM library was extended to support an integrated SDSM and MPI programming environment.

An analysis was given that outlined how to properly use message passing in the context of SDSM flush and barrier operations. Following this, a taxonomy of point-to-point communication identified four different communication cases depending on whether message buffers resided in private or shared memory. These cases were: I) private-to-private, II) private-to-shared, III) shared-to-private, IV) shared-to-shared. The shared-to-shared case was further classified into symmetric shared-to-shared and asymmetric shared-to-shared sub-cases depending on whether message buffers referred to the same or different logical shared memory regions.

The symmetric shared-to-shared case represents a new concept that is foreign to message passing programming as both message buffers in a point-to-point communication refer to the same logical shared memory region. This case was shown to be problematic when memory access side-effects from message passing operations were handled by SDSM coherency protocols. Paradoxically, it was shown that relying on SDSM coherency protocols could result in incoherent memory. The correct treatment of these cases required two conditions: 1) modifications to the send buffer must be made globally visible prior to a send operations, and 2) data received into a symmetrical receive buffer must be treated as memory *updates*. It was also noted that satisfying the first condition can be highly inefficient as it relies on the SDSM flush operation.

This inefficiency was addressed by leveraging key observations about message passing send operations that allow an SDSM system to use home-migration in place of flush operations. The benefits of this are two fold: Firstly, home-migration is a more efficient

alternative to flush when it is applicable; and secondly, it adapts an application's page placement to improve data locality.

Details of how the extended Danui SDSM handles each of the message send and receive cases, in combination with the different page states used by the SDSM, were given. As the ability to perform home-migration in a one-sided fashion was required, the details of how to perform this operation using the forwarding pointer technique described by Fang *et al.* [27] were discussed. A call-back infrastructure for supporting non-blocking operations was also described.

Timing measurements for the extended SDSM were made for a bulk data transfer benchmark and an implementation of the Parallel Dense Blocked LU Factorization algorithm.

The first benchmark represents a tractable direct evaluation of the bulk data transfer capabilities of the extended SDSM. Two variations of the benchmark were introduced; one was locality-aware, while the other was locality-unaware. Comparisons were made using SDSM-only, SMMP, and MPI-only implementations of both variations of the benchmark. It was found that the SMMP implementations outperformed the SDSM-only implementations in both benchmark variations. For the locality-aware version of the benchmark, the SMMP implementation catches up and eventually matches the performance of the MPI-only implementation.

The second benchmark explored the use of collective and non-blocking communication. Comparisons were made using SDSM-only and SMMP implementations. It was found that the SMMP implementation outperforms the SDSM-only implementation. However, the improvements were not as large as with the previous benchmark. This is understandable since integrating message passing into the program does nothing to improve the computation overhead. That is, the computation to communication ratio is important.

Chapter 6

Conclusions

This thesis sought to investigate the benefits of integrating shared memory and message passing programming. Towards this goal, a new Software Distributed Shared Memory (SDSM) library called Danui was developed from scratch that was subsequently extended to support an integrated shared memory and message passing (SMMP) programming environment. The Message Passing Interface (MPI) standard was selected to implement this SDSM for portability between platforms. Additionally, it made sense to expose an MPI interface when investigating SMMP programming models as MPI is the dominant message passing programming standard.

Danui was implemented using a home-based protocol and also uses SDSM technologies such as twinning-and-diffing [13, 22] and home-migration [39, 34, 27]. The home-based protocol was chosen because the type of applications that can benefit from integration with message passing are also the type of applications that can benefit from home-migration. Even so, the option to support a homeless protocol alongside the present implementation is available.

Cost models were developed for Danui SDSM operations. This allowed comparison between alternative implementations of the SDSM modules; specifically, the modules that provide mutual exclusion (Lock module) and maintain memory consistency (Heap module). For the case of memory consistency, all of the benchmark results and parametrised cost models point toward one conclusion: that is, data transfer costs from page fetches and page modifications dominate overheads. Case in point, figure 4.15 (p.110) shows that the difference in synchronization overhead between the Write-set N-Homes and Invalidate All MCMs is $\sim 430 \mu\text{s}$ when the SDSM has 20000 shared pages. In contrast, a single page fetch operation on the same system for a page size of 4KB costs $\sim 183 \mu\text{s}$ (see cost of RUM fault in equation 4.2, p.93). Thus, the extra $\sim 430 \mu\text{s}$ incurred by the Write-set N-Homes MCM during each barrier synchronization for the purpose of migrating pages to improve data locality is in general a net gain.

Danui was extended to support an integrated SMMP programming environment.

In doing so, an analysis of memory access side-effects from message passing operations and a taxonomy of point-to-point communication cases were given. The requirements for correct implementation were listed. It was noted that using SDSM flush operations to satisfy these requirements would be so costly as to render use of message passing pointless. It was then shown that leveraging the semantics of message passing operations in relation to an SDSM context made it possible to avoid use of costly memory flush operations. Instead, home-migration was used. Performance evaluation showed that the Danui SMMP implementation significantly reduces data transfer latency in SDSM programs.

The following sections describe future work for the Danui SDSM and its SMMP implementation.

6.1 Future Work for the Danui Software Distributed Shared Memory Library

Danui is a new MPI-based SDSM that was implemented to explore integrated SMMP programming. This section provides a critique of the Software Distributed Shared Memory part of Danui and lists future work.

6.1.1 Page Pre-fetching

Pre-fetching [70, 49, 15, 45] can be used to improve the Danui SDSM implementation by preemptively fetching a remote page before a page fault for that page has occurred. The objective is to minimize the time spent executing page fetch operations. However, the challenge of pre-fetching lies in how to determine what to fetch, and also when to fetch. Fetching the wrong page, or the right page too soon (e.g., before data values have been stored to the page), may increase SDSM overheads.

For the moment, an explicit form of pre-fetching is possible in Danui by using non-blocking `smPI_Send` and `smPI_Recv` calls. For example, this was used in the Dense Blocked LU benchmark described in section 5.3.2 to distribute data for perimeter blocks.

6.1.2 Multiple Computation Threads per SDSM Process

Multi-core processors are commonplace in computer systems and the number of cores in multi-core processors continues to increase. Therefore, it is likely to be beneficial for an SDSM to execute multiple computation threads per process. Danui is presently lacking in this area. One issue that would need to be addressed is how multiple computation

threads per SDSM process would complicate the SMMP programming model, since all the threads in a process share the same MPI process ID (rank).

6.1.3 Critical Path Performance Model and Locks

The extended critical path performance model presented in section 4.7 does not consider lock operations. Lock usage is more complicated to model as it involves synchronization of arbitrary pairs of processes.

A possible approach would be to extend Modeled Time to send the modeled time of a releasing process along with the lock. At the acquiring process, the modeled time can be adjusted to the release time plus the lock transfer cost based on the lock cost model.

6.1.4 Use of MPI_THREAD_MULTIPLE

The MPI_THREAD_MULTIPLE thread run-time mode is used to support the Event daemon thread. This has reduced the portability of the SDSM somewhat as not all MPI implementations support this mode (e.g. Open MPI 1.4 does not support this mode for InfiniBand). This situation is expected to change in coming years as multi-core processors become more widespread, and core counts increase. Then, MPI implementors will have more incentive to provide support for MPI_THREAD_MULTIPLE.

6.1.5 Cluster of Sub-clusters

The Danui SDSM can be initialized with respect to a given MPI communicator. Thus, by dividing MPI processes into sub-communicators, multiple SDSM instances can be initialized. This may be useful when the cluster has a hierarchical network where latencies are lower within certain groups of processes but higher when communicating across groups. Here, SDSM instances can be started for processes within a group to take advantage of the lower latencies; leaving inter-group communication to be solved using message passing.

Alternatively, the approach can be used to group processes according to some pipelined program consisting of multiple stages. Here, separate SDSM instances may be used by some stages depending on suitability.

While the Danui SDSM can provide the above features, a proper study is required to investigate the viability and usefulness sub-clusters.

6.2 Future Work for the Danui SMMP Implementation

In this section, future directions for integrated shared memory and message passing research using Danui.

6.2.1 Migrate Page to Receiver

Consider the symmetric shared-to-shared communication case and assume that a full page is transferred using matching `sMPI_Send` and `sMPI_Recv` calls. In such a case, the page will migrate to the sender in order to make any changes to the page globally visible. Observe that the receiver process will have an identical copy of that page after the transfer. Thus, an alternative to migrating the page to the sender process is to migrate the page to the receiver. This alternative will be useful if receiver processes modify the contents of the buffer after transfer. Danui can be extended to support this case.

Note that this can only apply to one receiver. For example, it is incorrect to migrate a page to all receivers in an `sMPI_Bcast` communication.

6.2.2 MPI-2 Remote Memory Access

The message passing operations considered by the SMMP model thus far have been limited to MPI-1 operations. MPI-2 operations include one-sided communication operations that enable remote memory access (RMA). Usability of these are however more constrained when compared to the shared memory view provided by the SDSM. However, constraints can be sources for optimization. For example, if `MPI_Put` is not guaranteed to have completed until an `MPI_Win_fence`, then that implies that the process that called `MPI_Put` is a single-writer for the whole duration between `MPI_Put` and `MPI_Win_fence`. As such, home-migration can be used. The difficulty in supporting RMA operations is that there are no matching calls that can be used to perform the bookkeeping work. Additional effort will be required to coordinate bookkeeping activity; perhaps through use of the Event module.

6.2.3 Collective Operations

Collective operations are attractive as they offer scalable communication to the SDSM. So far, only `sMPI_Bcast` and `aMPI_Bcast` have been implemented. `sMPI_Bcast` was also used in the SMMP version of the LU benchmark. However `aMPI_Bcast` has never been used. It exhibits an interesting behaviour that it broadcasts the contents of a shared memory buffer from one process into different regions of shared memory at each destination process. The result is to have the contents of one region in shared memory duplicated to other regions.

Other collective operations are more complicated. The `MPI_Reduce`, for example, has two message buffers and involves partial values that have not been reduced. (Does this rule out symmetrical buffers?) More study will be required to characterize these collective operations to determine how they would fit within a SMMP programming model.

6.2.4 Homeless Memory Consistency Manager and SMMP Extension

A Homeless Memory Consistency Manager (MCM) can be implemented in Danui alongside the Home-based MCM. Extending such an implementation to support SMMP programming will involve determining how to efficiently make modifications globally visible in a homeless SDSM; as home-migration is not possible.

6.2.5 SMMP Application Examples

In this work, we studied two SMMP benchmarks; a modified latency benchmark, and the Dense Blocked LU algorithm. More examples are required to determine the merits of the SMMP programming model; both in terms of programming complexity and performance. The difficulty is that the benchmarks available are either for shared memory or message passing models. Moreover, as there is no SDSM equivalent of an MPI standard, every application has to be individually written/porting into the SDSM interface.

Bibliography

- [1] InfiniBand trade association. <http://www.infinibandta.org>.
- [2] Ohio state university mpi benchmarks. <http://mvapich.cse.ohio-state.edu/benchmarks/>.
- [3] Open MPI: Open source high performance computing. <http://www.open-mpi.org/>.
- [4] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.
- [5] A. Agarwal. Retrospective: The MIT alewife machine: Architecture and performance. In *25 Years ISCA: Retrospectives and Reprints*, pages 103–110, 1998.
- [6] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiawicz, B.-J. Lim, K. Mackenzie, and D. Yeung. The MIT alewife machine: Architecture and performance. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 2–13, 1995.
- [7] A. Agarwal, D. Chaiken, G. D’Souza, K. Johnson, D. Kranz, J. Kubiawicz, K. Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT alewife machine: A large-scale distributed-memory multiprocessor. In M. Dubois and S. S. Thakkar, editors, *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, pages 239–261, Norwell, Massachusetts, 1991. Kluwer Academic Publishers.
- [8] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *SJCC*, 1967.
- [9] C. Amza, A. L. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29:18–28, 1996.
- [10] C. Amza, A. L. Cox, S. Dwarkadas, L.-J. Jin, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. In *Proceedings of*

- IEEE, Special Issue on Distributed Shared Memory*, volume 87, pages 467–475, Mar. 1999.
- [11] C. Amza, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Software DSM protocols that adapt between single writer and multiple writer. In *HPCA*, pages 261–271, 1997.
- [12] D. Bailey. FFTs in external or hierarchical memory. *J. Supercomputing*, 4:23–35, 1990.
- [13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. pages 168–176, 1990.
- [14] 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, CMU, Sept. 1991.
- [15] R. Bianchini, R. Pinto, and C. L. de Amorim. Data prefetching for software DSMs. In *Proceedings of the 1998 International Conference on Supercomputing (12th ICS'98)*, pages 385–392, Melbourne, Australia, July 1998. ACM.
- [16] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulwik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second, local area network. *IEEE Micro*, 15(1):29–36, Feb. 1995.
- [17] R. Brightwell and K. D. Underwood. Evaluation of an eager protocol optimization for MPI. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 327–334. Springer, 2003.
- [18] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *Proceedings of the European Workshop on OpenMP (EWOMP'99)*, 1999.
- [19] J. M. Bull and D. O'Neill. A microbenchmark suite for OpenMP 2.0. *ACM SIGARCH Computer Architecture News*, 29(5):41–48, Dec. 2001.
- [20] J. Cai, A. P. Rendell, P. E. Strazdins, and H. J. Wong. Performance model for cluster-enabled OpenMP implementations. In *Proceeding of 13th IEEE Asia-Pacific Computer Systems Architecture Conference*, pages 1–8, 2008.
- [21] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.
- [22] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, 1991.

- [23] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In R. E. Johnson and R. P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 519–538. ACM, 2005.
- [24] Cray, Inc. *Chapel Language Specification 0.785*.
- [25] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *ISCA*, pages 434–442, 1986.
- [26] T. A. El-Ghazawi, W. W. Carlson, and J. M. Draper. UPC language specification v1.1.1, Oct. 12 2003.
- [27] W. Fang, C. li Wang, W. Zhu, and F. C. M. Lau. A novel adaptive home migration protocol in home-based dsm. In *Proceedings of the 2004 IEEE Intl. Conf. on Cluster Computing*, pages 215–224, 2004.
- [28] S. S. Fu and N. feng Tzeng. Lock improvement technique for release consistency in distributed shared memory systems. In *FMPSC: Frontiers of Massively Parallel Scientific Computation*, pages 255–262. National Aeronautics and Space Administration (NASA), IEEE Computer Society Press, 1996.
- [29] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and Engineering Computation. MIT Pres, 1994.
- [30] 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 *ISCA*, pages 15–26, 1990.
- [31] W. Gropp. Learning from the success of MPI. In B. Monien, V. K. Prasanna, and S. Vajapeyam, editors, *HiPC*, volume 2228 of *Lecture Notes in Computer Science*, pages 81–94. Springer, 2001.
- [32] W. Gropp and E. Lusk. Why are PVM and MPI so different? *Lecture Notes in Computer Science*, 1332:3–10, 1997.
- [33] P. Guedes and M. Castro. Distributed shared object memory. In *Proceedings of the 4th Workshop on Workstation Operating Systems*, pages 142–149. IEEE Computer Society Press, 1993.

- [34] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory system. In *IEEE 4th HPC ASIA 2000*, pages 158–163. IEEE, May 2000.
- [35] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, T. Takahashi, and Y. Ishikawa. SCASH: Software DSM using high performance network on commodity hardware and software. In *Eighth Workshop on Scalable Shared-memory Multiprocessors*, pages 26–27. ACM, May 1999.
- [36] R. J. Harrison. Moving beyond message passing: Experiments with a distributed-data model. Technical report, Argonne National Laboratory, 1991.
- [37] J. Heinlein, K. Gharachorloo, S. A. Dresser, and A. Gupta. Integration of message passing and shared memory in the Stanford FLASH multiprocessor. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 38–50, San Jose, CA, Oct. 1994.
- [38] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, PDS-9(10):1029–1040, Oct. 1998.
- [39] W. Hu, W. Shi, and Z. Tang. Home migration in home-based software dsms. In *First Workshop on Software Distributed Shared Memory*, pages 21–26, 1999.
- [40] L. Iftode, J. P. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. Technical Report TR-509-96, Department of Computer Science, Princeton University, January 1996, 11 Pages.
- [41] A. Itzkovitz and A. Schuster. Multiview and millipage: Fine-grain sharing in page-based DSMs. In *Proceedings of the 3rd Symp. on Operating Systems Design and Implementation (OSDI'99)*, pages 215–228, Feb. 1999.
- [42] P. Keleher, A. L. Cox, H. Dwarkadas, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the 1994 Winter Usenix Conference*, pages 115–131, 1994.
- [43] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, 1992.
- [44] D. Kranz, K. L. Johnson, A. Agarwal, J. Kubiawicz, and B.-H. Lim. Integrating message-passing and shared-memory: Early experience. *ACM SIGPLAN Notices*, 28(7):54–63, July 1993.

- [45] A. I.-C. Lai and C.-L. Lei. Data prefetching for distributed shared memory systems. In *HICSS (1)*, pages 102–110, 1996.
- [46] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.
- [47] K. Li. IVY: A shared virtual memory system for parallel computing. In *ICPP (2)*, pages 94–101, 1988.
- [48] W.-Y. Liang, C.-T. King, and F. Lai. Adsmith: An efficient object-based distributed shared memory system on pvm. In *ISPAN*, pages 173–179. IEEE Computer Society, 1996.
- [49] H. Liu and W. Hu. A comparison of two strategies of dynamic data prefetching in software DSM. In *IPDPS*, page 62. IEEE Computer Society, 2001.
- [50] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
- [51] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard version 1.3*, Sept. 2008.
- [52] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard version 2.2*, Sept. 2009.
- [53] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Proceedings of Supercomputing'94*, pages 763–772, Washington DC, Nov. 1994. IEEE.
- [54] J. Nieplocha, R. Harrison, M. K. Kumar, B. Palmer, V. Tipparaju, and H. Trease. Combining distributed and shared memory models: Approach and evolution of the global arrays toolkit. In *Proceedings of the Workshop on Performance Optimization via High-Level Languages and Libraries (POHLL-02)*, 2002.
- [55] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A portable “shared-memory” programming model for distributed memory computers. In *Proceedings of Supercomputing'94*, pages 340–349, Washington DC, Nov. 1994. IEEE.
- [56] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. A nonuniform memory access programming model for high-performance computers. *Journal of Supercomputing*, 10(2):197–220, 1996.
- [57] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. *ACM SIGPLAN FORTRAN Forum*, 17(2):1–31, Aug. 1998.

- [58] Y. Ojima, M. Sato, T. Boku, and D. Takahashi. Design of software distributed shared memory system using MPI communication layer. In *Proceedings The 4th International Workshop on OpenMP: Experiences and Implementations WOMPEI 2005*, pages 18–25, Tsukuba, Japan, Jan. 2005.
- [59] OpenMP Architecture Review Board. *OpenMP Application Program Interface version 3.0*, May 2008.
- [60] I. Park, C. liang Ooi, and T. N. Vijaykumar. Reducing design complexity of the load/store queue. In *MICRO*, pages 411–422. ACM/IEEE, 2003.
- [61] F. Petrini, W. chun Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics network: High-performance clustering technology. *IEEE Micro*, 22(1):46–57, 2002.
- [62] M. Schulz. Extracting critical path graphs from MPI applications. In *CLUSTER*, pages 1–10. IEEE, 2005.
- [63] L. M. Silva, J. G. Silva, and S. Chapple. Implementing distributed shared memory on top of MPI: the DSMPI library. In *Proceedings of the 4th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'96)*, pages 50–57, Jan. 1996.
- [64] W. E. Speight, H. Abdel-Shafi, and J. K. Bennett. An integrated shared-memory/message passing API for cluster-based computing. Technical Report ECE TR-98-06, Electrical and Computer Engineering Department, Rice University, 1998.
- [65] W. E. Speight and J. K. Bennett. Brazos: A third generation DSM system. In *Proc. of the USENIX Windows NT Workshop*, Aug. 1997.
- [66] T. Sterling, T. Cwik, D. Becker, J. Salmon, M. Warren, and B. Nitzberg. An assessment of beowulf-class computing for NASA requirements: Initial findings from the first NASA workshop on beowulf-class clustered computing. In *Proceedings, IEEE Aerospace Conference*, Aspen, CO, Mar. 1998.
- [67] R. W. Stevens. *UNIX Network Programming*. Software Series. Prentice Hall PTR, 1990.
- [68] Sun Microsystems, Inc. *The Fortress Language Specification version 1.0*, Mar. 2008.
- [69] H. Tezuka, A. Hori, and Y. Ishikawa. PM: A high-performance communication library for multi-user parallel environments. Tech. Rpt. TR-96015, RWC, Nov. 1996.

- [70] H.-H. Wang, K.-C. Li, S.-H. Lu, and C.-C. Yang. Towards implementation of a novel scheme for data prefetching on distributed shared memory systems. *The Journal of Supercomputing*, 47(2):111–126, 2009.
- [71] H. J. Wong, J. Cai, A. P. Rendell, and P. E. Strazdins. Micro-benchmarks for cluster OpenMP implementations: Memory consistency costs. In R. Eigenmann and B. R. de Supinski, editors, *IWOMP*, volume 5004 of *Lecture Notes in Computer Science*, pages 60–70. Springer, 2008.
- [72] H. J. Wong and A. P. Rendell. Optimizing performance for SCore cluster enabled OpenMP applications: Solving the Laplace equation and performing Fourier transforms. In *Proceedings The 4th International Workshop on OpenMP: Experiences and Implementations WOMPEI 2005*, pages 10–17, Tsukuba, Japan, Jan. 2005.
- [73] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characteriation and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–37, New York, June 22–24 1995. ACM Press.
- [74] S. C. Woo, J. P. Singh, and J. L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 219–229, San Jose, California, 4–7 Oct. 1994. ACM Press.
- [75] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, Sept. 1998. Special Issue: Java for High-performance Network Computing.
- [76] B.-H. Yu, S. Cranefield, Z. Huang, and M. Purvis. Homeless and home-based lazy release consistency protocols on distributed shared memory. In V. Estivill-Castro, editor, *Twenty-Seventh Australasian Computer Science Conference (ACSC2004)*, volume 26 of *CRPIT*, pages 117–123, Dunedin, New Zealand, 2004. ACS.
- [77] Y. Zhou, L. Iftode, and K. Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *OSDI*, pages 75–88, 1996.
- [78] J. Zhu, J. Hoeflinger, and D. A. Padua. Compiling for a hybrid programming model using the LMAD representation. In H. G. Dietz, editor, *LCPC*, volume 2624 of *Lecture Notes in Computer Science*, pages 321–335. Springer, 2001.

- [79] J. Zhu, J. Hoeflinger, and D. A. Padua. A synthesis of memory mechanisms for distributed architectures. In *ICS*, pages 13–22, 2001.