Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems

Bharath Ramesh

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

Srinidhi Varadrajan, Co-Chair
Calvin J. Ribbens, Co-Chair
Dennis G. Kafura
Naren Ramakrishnan
Mark T. Jones

July 3, 2013
Blacksburg, Virginia

Keywords: Distributed Shared Memory, Virtual Shared Memory, Memory Consistency
Copyright 2013, Bharath Ramesh

Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems

Bharath Ramesh

(ABSTRACT)

Among the key challenges of computing today are the emergence of many-core architectures and the resulting need to effectively exploit explicit parallelism. Indeed, programmers are striving to exploit parallelism across virtually all platforms and application domains. The shared memory programming model effectively addresses the parallelism needs of mainstream computing (e.g., portable devices, laptops, desktop, servers), giving rise to a growing ecosystem of shared memory parallel techniques, tools, and design practices. However, to meet the extreme demands for processing and memory of critical problem domains, including scientific computation and data intensive computing, computing researchers continue to innovate in the high-end distributed memory architecture space to create cost-effective and scalable solutions. The emerging distributed memory architectures are both highly parallel and increasingly heterogeneous. As a result, they do not present the programmer with a cache-coherent view of shared memory, either across the entire system or even at the level of an individual node. Furthermore, it remains an open research question which programming model is best for the heterogeneous platforms that feature multiple traditional processors along with accelerators or co-processors. Hence, we have two contradicting trends. On the one hand, programming convenience and the presence of shared memory call for a shared memory programming model across the entire heterogeneous system. On the other hand, increasingly parallel and heterogeneous nodes lacking cache-coherent shared memory call for a message passing model. In this dissertation, we present the architecture of Samhita, a distributed shared memory (DSM) system that addresses the challenge of providing shared memory for non-cache-coherent systems. We define regional consistency (RegC), the memory consistency model implemented by Samhita. We present performance results for Samhita on several computational kernels and benchmarks, on both cluster supercomputers and heterogeneous systems. The results demonstrate the promising potential of Samhita and the RegC model, and include the largest scale evaluation by a significant margin for any DSM system reported to date.

"Live as if you were to die tomorrow. Learn as if you were to live forever."

Mahatma Ghandhi

My parents, without whom I could not have accomplished this
My brother Girish, who has always been there for me
My dearest Denitsa, who encouraged me to complete this work

Acknowledgements

I take this opportunity to thank the people who helped me in innumerable ways throughout the duration of this work:

- My advisor, Dr. Srinidhi Varadarajan, who has been a friend, collaborator, philosopher and a guide. He helped me through all stages of my work, from conceptualization to implementation.
- My co-advisor, Dr. Cal Ribbens, for his constant support, guidance and encouragement. This work would not have been possible without his input.
- My Ph.D. committee—Dr. Denis Kafura, Dr. Naren Ramakrishnan, and Dr. Mark Jones—for their suggestions, support and insightful questions.
- Dr. Godmar Back, for spending hours with me on technical discussions.
- Dr. Eli Tilevich, for his comments, suggestions and mirthful demeanor, which made working late tolerable.
- Dr. Kevin Shinpaugh and Byoung-Do Kim for providing me with resources and support during this work.
- The *glibc* and OFED mailing lists participants.
- The staff at the Department of Computer Science. In particular, Ginger Clayton, Melanie Darden, and Jessie Eaves.
- My friends who made my stay entertaining: Joy Mukherjee, Omprakash Seresta, Anil Bazaz, Jon Bernard, Karthik Channakeshava, Vijay Keswani, Shekhar Agarwal, Kripa Iyer, Apoorva Shende, Saurabh Bisht, Rakesh Phatak, Vedavyas Duggirala.
- My colleagues and friends at the Computing Systems Research Lab: Joy Mukherjee, Vedavyas Duggiralla, Hari Pyla, Shankha Banerjee, Craig Bergstrom, Mike Heffner, Pilsung Kang, Lee Smith, Joe Ruscio.
- My fellow squash players: John Harris, Apoorva Shende, Kathleen Meehan, Pankaj Joshi.
- My family for their unconditional love and support, without which this dissertation would not have been possible.
- Last but not least, my dearest Denitsa, for her patience and support during the toughest stages of this endeavor.

Contents

1 Introduction **1**
1.1 Is it time to rethink DSM systems? . . . 3
1.2 A new consistency model . . . 5
1.3 Organization of the dissertation . . . 7

2 Related Work **8**
2.1 Distributed memory and heterogeneous architecture programming models . . . 8
2.2 Shared memory programming models . . . 10
2.3 Shared memory for non-cache-coherent systems . . . 10
2.3.1 Survey of DSM systems . . . 11
2.3.2 Virtual shared memory for heterogeneous architectures . . . 14
2.4 Memory consistency models . . . 15

3 Regional Consistency (RegC) Model **19**
3.1 Regional consistency . . . 20
3.2 Formalizing RegC . . . 23

4 Samhita Architecture **26**

5 Implementation of Samhita **31**
5.1 Samhita communication layer (SCL) . . . 32
5.2 Samhita cache management . . . 34
5.3 Samhita global address space management . . . 37
5.4 Memory allocation strategies . . . 38
5.5 Barrier synchronization in Samhita . . . 40
5.6 Store instrumentation for fine grain updates . . . 44
5.7 Samhita API . . . 46

6 Performance Evaluation **49**
6.1 Benchmarks . . . 50
6.1.1 Memory bandwidth micro-benchmark . . . 50
6.1.2 Micro-benchmark . . . 51
6.1.3 STREAM TRIAD . . . 53
6.1.4 LU factorization . . . 53
6.1.5 Black-Scholes model . . . 53

6.1.6 Jacobi sweep . . . 53
6.1.7 Molecular dynamics application . . . 54
6.2 Cluster implementation performance evaluation . . . 54
6.2.1 Memory bandwidth micro-benchmarks . . . 55
6.2.2 Micro-benchmark . . . 56
6.2.3 STREAM TRIAD . . . 63
6.2.4 LU factorization . . . 67
6.2.5 Black-Scholes . . . 71
6.2.6 Jacobi sweep . . . 71
6.2.7 Molecular dynamics . . . 73
6.3 Heterogeneous system performance evaluation . . . 75
6.3.1 Memory bandwidth micro-benchmarks . . . 75
6.3.2 STREAM TRIAD . . . 76
6.3.3 Jacobi sweep . . . 78
6.3.4 Molecular dynamics application . . . 80

7 Conclusion and Future Work **81**
7.1 Conclusion . . . 81
7.2 Future work . . . 82

Bibliography **84**

List of Figures

3.1 Pseudo code describing regional consistency. 20
3.2 Comparison of RegC, RC, LRC and ScC 22

4.1 Architectural view of Samhita DSM system consisting of memory servers and resource managers, which provide a consistent shared address space for the compute servers. . 27
4.2 Similarities between the implementation of Samhita for a cluster and heterogeneous system. 28
4.3 Sequence of actions between compute servers, memory servers and resource manager for (a) memory allocation, (b) job startup and (c) synchronization operations. 29

5.1 Remapping of physical pages from Samhita cache region to an application's virtual address space during page admit, and remapping physical pages from an application's virtual address space to the Samhita cache region on page evict. 35
5.2 Example of store instrumentation in Samhita (a) source code, (b) source compiled to LLVM IR and (c) instrumented LLVM IR 45

6.1 Computational kernel of the micro-benchmark code used in the experiments, reflecting local allocation. For global allocation, array indices are a function of thread id. For global strided allocation, the k loop take strides of num_threads. 51
6.2 Memcpy bandwidth achieved for varying cache size for Ithaca and SystemG. 55
6.3 Memset bandwidth achieved for varying cache size for Ithaca and SystemG. 56
6.4 Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated locally. 57
6.5 Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally. 58
6.6 Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally, but access using strides. 59
6.7 Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated locally. 59
6.8 Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally. 60

6.9 Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally, but accesses using strides. 60
6.10 Compute time vs. number of rows of doubles allocated. Compute time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$. 61
6.11 Synchronization time vs. number of rows of doubles allocated. Synchronization time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$. 62
6.12 Synchronization time (log scale) vs. number of cores. Synchronization time for Pthreads and Samhita are compared for local, global allocation and global allocation with strided access. $M = 100$, $B = 256$ and $S = 2$ are kept fixed. 63
6.13 Sustained memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Vectors of dimension $n = 16M$. 64
6.14 Sustained memory bandwidth vs. number of cores for Samhita and Pthreads based STREAM TRIAD benchmark. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for $p = 1$ is $n = 16M$; problem size for $p = 256$ is $n = 4G$. 65
6.15 Sustained memory bandwidth vs. number of cores for Samhita for two different problem sizes. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for small problem on $p = 1$ node is $n = 16M$; problem size for large problem on $p = 1$ node is $n = 32M$. 66
6.16 Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 4000$; problem size for $p = 256$ is $n = 64000$. 67
6.17 Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 8000$; problem size for $p = 256$ is $n = 128000$. 68
6.18 Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 10000$; problem size for $p = 256$ is $n = 160000$. 69
6.19 Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Fixed problem size $n = 30000$. 69
6.20 Computational rate vs. problem size for single threaded Samhita based LU with cache size of 1GB. 70
6.21 Parallel speedup vs. number of cores for PARSEC Black-Scholes application. 70
6.22 Parallel speedup vs. number of cores for Jacobi. Speedup is relative to 1-core Pthreads execution time. 72
6.23 Computation rate vs. number of cores for Samhita and Pthreads based Jacobi. Data size $3n^2$ and number of cores p are scaled proportionally, i.e., $3n^2/p$ is a constant. Problem size for $p = 1$ is $n = 4096$; problem size for $p = 256$ is $n = 65536$. 73
6.24 Parallel speedup vs. number of cores for molecular dynamics. Speedup is relative to 1-core Pthreads execution time. 74
6.25 Memcpy bandwidth achieved with varying cache size. 76
6.26 Memset bandwidth achieved with varying cache size. 77

6.27 Achieved memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Vectors of dimension $n = 2M$. 77
6.28 Achieved memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for $p = 1$ is $n = 2M$; problem size for $p = 120$ is $n = 240M$ 78
6.29 Parallel speedup vs. number of cores for Pthread and Samhita based Jacobi kernel. Speedup is relative to 1-core Pthread execution. 79
6.30 Computational rate vs. number of cores for Pthread and Samhita based Jacobi kernel. Data size $3n^2$ and number of cores p are scaled proportionally, i.e., $3n^2/p$ is a constant. Problem size for $p = 1$ is $n = 1344$; problem size for $p = 120$ is $n = 14722$. 79
6.31 Parallel speedup vs. number of cores for Pthread and Samhita based MD application. Speedup is relative to 1-core Pthread execution. 80

List of Tables

1.1 Key performance numbers for representative platforms over the last 20 years. 3
1.2 Comparison of properties of popular consistency model. 5

5.1 Samhita Communication Layer API 33
5.2 Samhita API 47
5.3 Comparison of memory allocation API of C library and Samhita 48
5.4 Comparison of Pthreads and Samhita API for thread management and synchornization 48

Chapter 1

Introduction

The emergence of many-core architectures and the need to exploit parallelism is perhaps the most important challenge in computing today. Programmers are striving to exploit parallelism across virtually all platforms and application areas. On the platforms with the largest market share (e.g., portable devices, laptops, desktop, servers) the traditional shared memory programming model dominates. Consequently, there is a large and growing ecosystem of shared memory parallel programmers, tools, and design practices. However, on high-end platforms distributed architecture still have their place. They provide a cost effective approach for meeting the extreme demands for processing and memory of critical problem domains such as scientific computation and data intensive computing. This popularity of distributed architectures is reflected in the bi-annual rankings of the "Top 500" supercomputers [1]; close to 82% of the current "Top 500" supercomputers are clusters. Unfortunately, the dominant programming model for distributed memory is message passing, a programming model that is widely seen as more difficult than a shared memory model, especially for inexperienced programmers [46, 45].

In the last few years emerging node architectures are both highly parallel and increasingly heterogeneous. Furthermore, these emerging heterogeneous architectures do not present the programmer with a cache-coherent view of shared memory at the level of the individual node. It is not clear what programming model is best for these heterogeneous platforms, which feature multiple traditional processors along with accelerators or coprocessors. One response to the lack of node-wide

shared memory is to use a message passing model within a node, essentially viewing each node as a mini-cluster. Unfortunately, this means giving up the advantage of shared memory that typically is available within some components of the node. Furthermore, the amount of memory per core in coprocessors is typically low. So treating cores of the coprocessor as nodes of a mini-cluster and not fully using the much larger memory associated with the host or general purpose CPU, severely limits the size of the problems that can be solved. Another programming option is to offload computation to the coprocessors. This allows the code running on the coprocessor to take advantage of the programming model that works best for the coprocessor (i.e., shared memory), but still requires the programmer to manage moving data to and from the coprocessor.

Thus, we have two opposing trends. On the one hand, we want to leverage and encourage the shared memory parallel programming community, e.g., Pthread programmers, existing shared memory codes. On the other hand, we want to leverage the cost and scalability advantages of distributed memory architectures and heterogeneous accelerated nodes, neither of which provides cache-coherent shared memory in hardware.

This is not the first time the computing community has faced these kinds of conflicting priorities and trends. Almost since distributed memory clusters first appeared, researchers have explored ways to provide virtual shared memory, or *distributed shared memory* (DSM), over such systems. The goal of DSM systems is to provide shared memory semantics over physically distributed memories. DSM systems achieve this by hiding the necessary underlying data distribution from the programmer, thereby providing a simpler programming model than a typical distributed memory model such as message passing. A critical component of any DSM system is its consistency model, which defines the semantics of memory accesses. To improve performance, consistency models are typically relaxed to enable greater parallelism in data accesses. DSM systems have been implemented as hardware [60, 30, 57], software [58, 37, 76, 26, 52, 71, 62, 35, 47, 69, 64, 66], and hybrid systems [25, 27, 53, 50]. Unfortunately, although much of this work yielded valuable insight, the lasting impact of these systems has not been high.

Table 1.1: Key performance numbers for representative platforms over the last 20 years.

Year	Processor clock rate	Memory latency	Memory bandwidth	Network latency	Network bandwidth
92 - 93	40MHz - 200MHz	80ns - 100ns	1.5Gbps	$> 200\mu s$	10Mbps
94 - 96	60MHz - 500MHz	70ns - 100ns	1.5Gbps	$> 100\mu s$	100Mbps
97 - 99	100MHz - 1GHz	70ns - 100ns	2.1Gbps	$30\mu s - 100\mu s$	1Gbps
00 - 04	500MHz - 2.4GHz	70ns - 120ns	8.5Gbps	$\sim 10\mu s$	10Gbps
05 - 06	1GHz - 3.2GHz	70ns - 100ns	20Gbps	$1.3\mu s$	20Gbps
07 - 08	1.5GHz - 3.2GHz	60ns - 100ns	40Gbps	$1.3\mu s$	20Gbps
09 - 10	1.7GHz - 3.3GHz	50ns - 60ns	128Gbps	$1\mu s$	40Gbps
11 -	1.8GHz - 3.4 GHz	50ns - 60ns	250Gbps	$500nss$	56 Gbps - 100Gbps

1.1 Is it time to rethink DSM systems?

Historically, DSM systems were doomed by unacceptable performance due to relatively slow interconnection networks. Table 1.1 suggests it is time to give DSM systems a new look. The data shows two major trends that have occurred since the early work on DSM systems in the late eighties and nineties. Notice two important trends:

1. Network performance has increased significantly, with network latencies falling by nearly three orders of magnitude and network bandwidth increasing by nearly four orders of magnitude. More importantly, network bandwidth is almost comparable to main memory today, as compared to the early nineties, when it was more than two orders magnitude slower than main memory.

2. Memory latencies have remained largely flat, while processor clock speeds have grown by over two orders of magnitude. This increasing gap (the memory wall) can often be hidden by careful cache management and accurate hardware prefetching.

These two trends fundamentally change the design challenge of a DSM system by casting it as a cache management problem. In effect, if we can design sophisticated cache management, particularly prefetching, we can exploit the similarity in bandwidth between network and memory today and hide the higher latency of remote memory access to provide shared memory performance comparable to current hardware solutions. In the scenario of emerging heterogeneous architecture, the motivation and challenge are very similar to the traditional DSM case: programming ease prefers one model,

but peak performance (by some measure) prefers another. In particular, by treating the processing cores of accelerators or coprocessors as individual cores of a cluster, can we provide effective virtual shared memory for these heterogeneous nodes as well?

One final observation that motivates a new look at DSM is the emergence of Remote Direct Memory Access (RDMA), the technology underlying the impressive interconnect performance numbers in Table 1.1. RDMA provides a mechanism for moving data from the memory of one computer to another without involving either computer's operating system. It is perhaps ironic that RDMA is a more natural fit for the shared memory programming model than the message passing model. While sequential, two-sided message passing paradigms such as MPI have been optimized for execution over RDMA using buffer affinities, large message transfer still involves additional higher layer messaging and added software latency. RDMA requires only one side to have the information to complete a communication, which corresponds nicely to a shared memory programming model, where one thread needs only a pointer in the shared global address space to access any data from any other thread.

The core observation underlying our approach is that although the latency of remote memory access over RDMA is $\sim$1-2μs, the bandwidth, at several gigabytes per second, rivals main memory performance. This observation fundamentally changes the challenge of designing a DSM system by reducing it to a cache management problem. For example, consider the multi-core Intel Nehalem processor, widely used in cluster supercomputers. Access to on-chip L1 cache takes 4 cycles, whereas access to main memory can take over 300 cycles ($\sim$100ns for data on different rows in DRAM bank), even ignoring the overhead of L2 and L3 misses at 11 cycles and 39 cycles respectively. This factor of 300 in the access time between register access and main memory access is largely hidden by careful cache management and accurate prefetching. We take this argument one step further by treating memory local to a cluster node as another layer of the caching hierarchy. In effect, local memory is used to cache data from remote memory, and by effectively managing this cache, we can largely hide the latency difference between $\sim$100ns for local memory access and $\sim$2μs for remote memory access. Furthermore, many scientific applications have fairly regular data access patterns—for instance, row or column major order, strided across—that can be detected and used to drive

Table 1.2: Comparison of properties of popular consistency model.

Model	Shared data & synchronization primitive association	Distinguish critical & non-critical data	Granularity	Non-critical section data consistency guarantees
Entry consistency	explicit	yes	object	no guarantees
Scope consistency	transparent	yes	page	requires barriers
Release consistency	transparent	no	page	consistent
Regional consistency	transparent	yes	page - ordinary region object - consistency region	consistent

intelligent prefetching strategies.

Given the promise of RDMA as provided by emerging high performance interconnects, we believe a usable DSM system is now possible. In this research, we propose *Samhita*, a new user-level software virtual shared memory system designed to enable existing threaded codes that run on homogeneous multicore desktop systems to run on cluster supercomputers and emerging heterogeneous architectures with minimal code modification. The system is architected with memory servers and compute nodes enabling its use for:

1. Multi-threaded codes that require a consistent view of memory.
2. Single-threaded applications that require a large address space.

1.2 A new consistency model

There are two diverging trends—the programmer's desire for a shared memory programming model and the architect's need to sacrifice cache-coherence for scalability and heterogeneity—that motivate a new look at memory consistency models. Memory consistency and coherence are related but distinct concepts. Coherence defines what can be read from a memory location; consistency defines when a write to a memory location is visible to subsequent reads. The consistency model defines the semantics of memory accesses; it determines both the performance and programmability of the virtual shared memory system. In this research we propose *regional consistency* (RegC), a new consistency model that gives programmers the strong shared memory programming model they prefer, and can be implemented efficiently over modern non-cache-coherent systems.

The common approach to providing shared memory semantics over non-cache-coherent architectures is to relax the consistency model to allow greater parallelism in data access, but at the cost of some ease of programming. Table 1.2 compares three popular relaxed consistency models with RegC in terms of four defining properties:

1. Whether shared data must be explicitly associated with synchronization primitives.
2. Whether critical and non-critical section memory accesses are distinguished.
3. The granularity at which consistency updates are typically done.
4. To what extent consistency is maintained for non-critical memory updates.

The entry consistency model [22] requires explicit association between shared data and synchronization primitives; it does not provide any memory consistency guarantees for non-critical-section data. Although entry consistency has performance advantages, the association restriction and its lack of consistency for updates to shared data done outside of critical sections make it difficult to use. Scope consistency [48] removes the explicit association requirement and makes non-critical section updates consistent at barriers. Scope consistency unburdens the programmer from the explicit shared data to synchronization primitive association, but consistency updates are done at the granularity of a page, which has performance implications. Release consistency [41] does not require any explicit association nor does the programmer have to use barriers to make non-critical section data consistent. In this sense, release consistency is easier to program than either entry or scope consistency. However, since release consistency does updates at the granularity of a page, and since it does consistency updates at page granularity for both critical sections and non-critical sections, it can still suffer from performance problems.

Our new RegC memory consistency model is motivated by these observations. It explicitly distinguishes between modifications to memory protected by synchronization primitives and those that are not, allowing for a more performant and scalable implementation. In our implementation of RegC in Samhita we use fine grain (object level) updates for modifications to shared data protected by synchronization primitives and use page invalidations for modifications to data not protected by

synchronization primitives. In essence, RegC is similar to entry consistency for critical-section shared data accesses, associating critical-section updates to mutual exclusion locks transparently similar to scope consistency, and similar to release consistency for non-critical section accesses. RegC provides a sufficiently strong consistency model, in that it allows Samhita's API to correspond directly to the familiar Pthreads API. In addition, Samhita presents the programmer with the familiar fork-join model as is commonly used in threaded codes. Pthreads codes can be ported to RegC/Samhita trivially. In fact, in our performance evaluation, the benchmarks use the same code base, with Pthreads and Samhita calls simply selected by macro substitution.

1.3 Organization of the dissertation

The remainder of this dissertation is divided into six chapters. In Chapter 2 we provide an overview of related work on parallel programming models and approaches to programming clusters and heterogeneous architectures. Chapter 3 describes our new RegC memory consistency model. The architecture of the Samhita virtual shared memory system is presented in Chapter 4. Next we discuss the implementation of Samhita and Samhita's implementation of RegC in Chapter 5. We present the results of a performance evaluation on both traditional distributed memory clusters and heterogeneous (CPU/coprocessor) platforms in Chapter 6. The last chapter focuses on the conclusion and outlines selected future work.

Chapter 2

Related Work

A parallel programming model is used by a programmer to take advantage of the underlying parallel system. The prevalent models used in such systems are either shared memory or message passing models based on the inter-process communication mechanism used. Our research focuses on providing shared memory semantics over both interconnected memories and emerging node architectures by providing a virtual shared memory system across these non-cache-coherent systems. In Section 2.1 we briefly describe the programming models predominantly used in distributed memory architectures and emerging heterogeneous architectures. We present common shared memory programming models in Section 2.2. Section 2.3 presents approaches that provide shared memory semantics over distributed memory. Lastly we present prior work on memory consistency models in Section 2.4.

2.1 Distributed memory and heterogeneous architecture programming models

The predominant programming model used for distributed memory architectures is message passing. Message Passing Interface (MPI) [17] is the standardized language independent protocol. The MPI standard defines the syntax and semantics of the API that the programmer uses. It provides both point-to-point and collective communication primitives. Programmers are responsi-

ble for distributing data among the different processes running on the individual processing cores of the cluster apart from managing communication. Modern cluster supercomputers increasingly use multi-core processors at the individual node level. The MPI standard only defines minimal requirements for thread compliant implementations. Programmers are increasingly developing hybrid applications, which use MPI for inter-node communication and shared memory model for intra-node communication. When developing these codes the programmer is responsible for preventing races between threads of the same application and also must simultaneously ensure that no conflicting communication calls are posted.

For emerging heterogeneous architectures the ideal programming model is still an open question. The lack of any cache-coherent memory between the accelerators and the host processors implies that one cannot take advantage of the shared memory programming model that programmers use in homogeneous multi-core systems. The predominant approach is to offload computation to the accelerators and coprocessors. This requires the programmer to be responsible for moving data to and from the accelerators before and after offloading the computation. These accelerators require hardware specific code, which increases the complexity of programming these systems and reduces code portability. The most common programming environments for general purpose computing on GPUs are CUDA [2] and OpenCL [3]. CUDA and OpenCL APIs are similar in many aspects. However, the CUDA environment is available only for Nvidia GPU devices. OpenCL provides an environment for writing programs that can execute on heterogeneous platforms that consist of both CPUs and GPUs.

To reduce the burden on the programmer modern compilers try to hide some of the complexity of programming these hardware by providing the programmer with offload primitives. OpenACC [18] and the upcoming 4.0 standard of OpenMP [19] present the programmer with compiler directives to offload loops and regions of code to the accelerator. The programmer is still required to identify the data that needs to be copied to and from the accelerator. Furthermore, the load balancing across the different processing elements is challenging because the accelerators and host processors have different computational characteristics.

2.2 Shared memory programming models

Existing shared memory programming models have been implemented as a new programming language, as extensions to existing sequential languages, or as libraries to be used with existing sequential programming languages. The motivating goal for all the implementations has been ease of programming to increase productivity. The most common and widespread shared memory programming model is the threaded programming model. In a threaded programming model, threads are a subset of a process. Threads share the address space of the process but execute independently. Communication is simplified as threads share the process address space. Context switching between threads is faster than between processes. There have been numerous implementations of the threaded programming model. Most of these implementations follow the POSIX Thread model [20].

The other commonly used shared memory programming model is the Open Multi-Processing (OpenMP) [19] programming model, which is an implicit threaded model. OpenMP provides a programmer with a set of function calls and compiler directives to write parallel shared memory applications. OpenMP supports programing in C, C++ and FORTRAN. In OpenMP multiple threads are forked and the tasks are divided amongst them; these threads run concurrently. The run-time behavior of an application, including the number of threads, can be controlled using environment variables.

2.3 Shared memory for non-cache-coherent systems

There have been numerous attempts to present a unified model for both shared and distributed memory architectures. Linda [40] is one of the earliest such models. Linda at times is referred to as a co-ordination language as it provides a model for co-ordination and communication among parallel processes. The Linda programming model aims at separating communication from computation. Each program using Linda is made up of 'tuples' – abstract data objects that can be either executable code or data values. These 'tuples' are stored in a shared abstract execution environment called the 'tuple space'. The individual processes are allowed to perform a fixed set of operations in this 'tuple space' which affects these stored 'tuples'. The semantics of the basic operations or primitives

of Linda, like read or write, can be considered to be analogous to the synchronous receive and send operations of the message passing programming model. In effect, Linda can be considered to resemble the sequential message passing paradigm.

The second programming model that presents a unified model for both shared and distributed memory architectures is the Partitioned Global Address Space (PGAS) [5] model. In this model, the global address space is logically divided into partitions. A part of each partition is local to each processor. The advantage of the PGAS model is the ability to exploit data locality. The PGAS programming model is implemented as extensions to existing sequential programing languages. For example, Unified Parallel C (UPC) [29] is an extension of the C programming language, Co-array FORTRAN [65] is an extension to FORTRAN, Titanium [44] is an extension of Java. Chapel [4] and X-10 [6] are newer variants of the PGAS model – the asynchronous partitioned global address space (APGAS) model. The APGAS model extends the traditional PGAS model by allowing both local and remote asynchronous task execution.

2.3.1 Survey of DSM systems

The area of DSM systems is a well researched area. However, the majority of the DSM systems implemented have been research prototypes. DSM systems have been implemented as hardware systems, e.g., Merlin [60], Memnet [30] and DASH [57]. There have been a few hybrid systems, such as Alewife [27], FLASH [53] and SHRIMP [25]. More details about some of the hardware and hybrid implementations can be found in [68]. Our focus is primarily on software DSM systems because they are portable. DSM systems have been implemented as software systems since Ivy [58], the first software DSM system implementation.

The first requirement for any DSM to gain acceptance is ease of use. Ease of use of a DSM system depends on the programming model presented by that system. The threaded programming model is the most common and widespread model used for shared memory parallel architectures. This is reflected in the choice of the programming model adopted by the majority of software DSM systems. There are a few exceptions, such as Mirage [37] and Momento [73], which have no particular programming model associated with them; they have been implemented by modifying the

operating system kernel. More recent implementations of DSM systems, such as DSM-Threads [62], Orion [63] and Murks [67], support either the POSIX Thread interface or something close to this standard threading model. We follow this approach by supporting a threaded programming model that also resembles POSIX Threads.

The second requirement for any software DSM system to gain acceptance is portability. Portability of a DSM system enables the DSM to run on a wide variety of platforms. Software DSM systems have been implemented at various levels. Some of these systems, such as Mirage, Momento and Murks have been implemented as modifications to the operating system kernel. On the other hand, Ivy, Mermaid [76] and Munin [26] have been implemented predominantly as user-level systems with some modification to the operating system kernel. A system requiring any modifications to the operating system kernel is harder to port to newer operating systems. Midway [22] requires a modified compiler, while Munin requires a modified linker apart from compiler modifications. This dependence on system software modifications is one of the main reasons these systems have remained research prototypes. Additional requirements in the form of compiler support or modified linkers makes a system harder to port and use with commonly available compilers and linkers. Most widely used DSM systems (e.g., TreadMarks [52], DSM-Threads, JIAJIA [47] and C Region Library (CRL) [49]) have been implemented without any modifications to the operating system kernel, nor do they require compiler support or linker modifications; these systems depend only on standard system libraries. This approach is reflected in our choice to implement a DSM system depending only on standard system libraries and requiring no modifications to the operating system kernel.

The third requirement for any DSM system to gain acceptance is efficiency. To improve performance in such a scenario, the standard technique used is data replication using caches. With the introduction of caches comes the issue of maintaining coherence and consistency. The efficiency of the system directly depends on the choice of the coherence protocol and the memory consistency model implemented by a DSM system (details of consistency models are described in Section 2.4). Leslie Lamport proposed the sequential consistency model [54], the first consistency model extending uni-processor semantics to multi-processor systems. Many of the initial DSM systems (e.g. Ivy, Mirage and Mermaid) implement the sequential consistency model. However, the sequential consis-

tency model, though conceptually intuitive, imposes severe restrictions, which affect performance and scalability. There have been other consistency models [34, 42, 41, 22, 48] proposed to alleviate these issues by relaxing or weakening the restrictions imposed by the sequential consistency model. Munin supports the use of multiple consistency models to maintain consistency of shared data. Munin implements both the sequential consistency model and release consistency model [41]. The release consistency model ensures all previous updates to the shared space are visible after a release of a synchronization primitive. Midway introduces entry consistency [22] in its implementation, which requires each shared object to be associated with a unique synchronization primitive. CRL does not use any memory consistency model but requires the application developer to explicitly group access to a region as operations (e.g., read/write) and annotate them using calls to CRL library. This makes the programming model exposed by CRL extremely complicated and prone to errors.

In spite of using relaxed consistency models, these systems are hindered by an unacceptable level of performance imposed by the interconnection network on which these systems are implemented. Most of the earlier systems target earlier generations of interconnection networks such as ATM networks, Ethernet or other interconnection networks that do not provide sufficiently low latency and high bandwidth. To conceal these communication overheads, DSM systems implement a modified form of the above mentioned memory consistency models. TreadMarks implements the lazy release consistency model [51]. In the lazy release consistency model the updates are not sent immediately at the release of any synchronization primitive but delayed till the subsequent acquisition of any synchronization primitive. JIAJIA implements a lock-based cache coherence protocol [75] using scope consistency [48]. None of the above mentioned DSM system implementations tried to solve the problem of network latency presented by an interconnection network. Jump-DP [28] attempts to reduce the communication latency by implementing a new socket interface in the operating system kernel, which has lower overhead. This implementation of a new socket interface greatly crippled the portability of Jump-DP.

As mentioned, earlier DSM systems combine the physical memories of the computers on which the application is running to form the shared address space. In early DSM systems such as Ivy,

Mirage, Mermaid, Munin and TreadMarks, each node maintains an entire copy of the shared address space. This presents a significant limitation since the shared address space is limited to the physical memory available on a single cluster node. In later DSM systems such as JIAJIA, the shared address space is distributed across the physical memories of nodes running the application. However, this design couples the shared memory footprint with the number of threads needed to run the application, preventing a single threaded application running on one node from accessing a large pool of shared memory. In contrast, Samhita separates memory servers from compute resources, enabling cluster designs that use a dedicated bank of memory servers to serve applications independent of the number of threads of execution.

The earliest attempt to use memory-mapped communication (similar to RDMA) to implement a DSM system is Virtual Interface Architecture [69]. Later DSM systems like NGDSM [64] and ViSMI [66] are implemented using InfiniBand switched fabric [9], but continued to maintain complete copies of the address space similar to early DSM systems, significantly limiting the size of the shared address space.

2.3.2 Virtual shared memory for heterogeneous architectures

Over the last few years there have been numerous attempts to extend the shared memory programming model for heterogeneous architectures. Global Memory for ACcelerators (GMAC) [39] is an asymmetric distributed shared memory system for heterogeneous architectures that use GPUs. GMAC provides set of APIs to allocate memory on the accelerator and allows the host to access the allocated memory on demand. The data-centric programming model presented by GMAC allows the CPU to access memory on the accelerator but not vice versa. All coherence and consistency operations are performed on the CPU due to the asymmetry in memory access.

Saha et al. [70] present a programming model for heterogeneous architectures. The system does not share the entire application address space; instead, it only shares a part of it between the different processing components. Language annotations are provided to demarcate code that executes on the different processing elements and data that is shared between them. The system uses the release consistency model using implicit acquire/release semantics whenever annotated code

is executed. The programmer is also provided with explicit memory allocation, dynamic ownership attributes and consistency operation APIs.

2.4 Memory consistency models

For a programmer to write correct concurrent applications, results of memory operations need to be predictable. Memory consistency models describe the rules, which when followed guarantee that memory accesses will be predictable. There are several memory consistency models that have been proposed in the literature. In this section we briefly describe sequential consistency (SC) [54], weak consistency (WC) [34], processor consistency (PC) [42], release consistency (RC) [41], entry consistency (EC) [22], and scope consistency (ScC) [48] which are some of the widely used consistency models in distributed shared memory systems.

Lamport defines sequential consistency as "...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". Sequential consistency is a widely used consistency model in earlier implementations of DSM systems. The definition of the model brings out two aspects: (1) program order is maintained at each processor (2) global order is an interleaving of all the sequential orders at each processor. The model, though conceptually simple, imposes restrictions that negatively affect the performance of a system implementing sequential consistency. To alleviate the effect of sequential consistency other consistency models have been proposed that relax or weaken the restrictions.

The weak consistency model is one of the earliest weak models proposed. The weak consistency model differentiates shared data into two categories, one that has no effect on concurrent execution of the algorithm and the other that includes synchronization variables to protect access to shared data or provide synchronization operations. The weak consistency model is associated with three properties: (1) access to all synchronization variables is sequentially consistent, (2) no operation on synchronization variables is permitted till all previous accesses to shared data are performed and (3) no access to shared data is allowed till all previous operations on a synchronization variable have

been performed. An important distinction between weak consistency and sequential consistency is that consistency is enforced on a set of accesses rather than individual accesses.

The one advantage of weak consistency that improves performance is that all writes from a single processor can be overlapped. This imposes a burden on the programmer to use synchronization variables as multiple writes can occur to the same location. Processor consistency follows a middle approach between weak consistency and sequential consistency. Goodman defines processor consistency as "...the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program [42]". Processor consistency allows writes from two processors, observed by themselves or a third processor, not to be identical. However, writes from any processor are observed sequentially.

One of the biggest drawbacks in weak consistency is the fact that when a synchronization variable is accessed, the processor has no knowledge if the access to the shared data is complete or about to start. This requires the processor to perform a memory consistency operation every time a synchronization variable is accessed. Release consistency extends weak consistency by categorizing accesses to the shared data as *ordinary* or *special* accesses, which are equivalent to access to data and synchronization variables, respectively, in the weak consistency model. Release consistency further categorizes *special* accesses as *sync* and *nsync* accesses. Finally, *sync* access are further categorized as either *release* or *acquire* accesses (analogous to mutex lock operation). For a system to be release consistent it obeys the following rules: (1) before any ordinary read or write access is performed, all previous *acquire* accesses must be performed, (2) before a *release* access is performed all previous reads and writes done by the processors must be performed and (3) all accesses to synchronization variables are processor consistent. At every release the processor propagates its modification to shared data to all other processors. This entails a lot of data transfer overhead. To reduce the amount of data transfer, propagation of modified data is postponed in lazy release consistency (LRC) [51]. In LRC the acquiring processor determines the modification it requires to meet the requirements of RC.

Both weak consistency and release consistency models use synchronization primitives to ensure ordering of shared data variables. Entry consistency (EC) uses this relationship between synchro-

nization primitives and access to shared data. EC requires all shared data to be explicitly associated with at least one synchronization primitive. Whenever a synchronization primitive is acquired all updates to the shared data associated with that synchronization primitive are performed. In EC each synchronization primitive has a current owner that last acquired the primitive. When the ownership changes because another processor acquires the synchronization primitive, all updates to the shared data associated with the primitive are sent to the acquiring processor. To reduce performance impact, synchronization primitives can exist in two modes – *exclusive* and *non-exclusive*. In the *non-exclusive* mode, though the synchronization primitive is owned by one processor it can be replicated at others. Only a single processor is allowed to acquire a synchronization primitive in *exclusive* mode. To modify the shared data associated with a synchronization primitive a processor must own the synchronization primitive in *exclusive* mode.

Though association of shared data with synchronization primitives reduces the overhead associated with data transfer among processors, EC is hindered by the increased complexity of explicitly associating shared data with synchronization primitives. Programming using EC is complicated and can be error prone. Scope consistency (ScC) alleviates the explicit association of shared data with synchronization primitives. Scope consistency detects the association dynamically at the granularity of pages, thus providing a simpler programming model. The implicit association of memory accesses to synchronization primitives is termed *consistency scope*. For a system to be scope consistent it obeys the following rules: (1) before a new session of a consistency scope is allowed to be open at a processor, all previous writes performed with respect to the scope need to be performed at the processor and (2) access to memory is allowed at a processor only after all consistency scopes have been successfully opened. Though scope consistency presents a relaxed consistency model, the programming model exposed to the user is complex when compared to RC or LRC. The authors of scope consistency mention that precautions need to be take to ensure that a program runs correctly under ScC [48].

All of the previously discussed consistency model are sequentially consistent for data race free codes. The authors of location consistency (LC) [38] present a model that is not sequentially consistent, i.e., writes to the same location are not serialized and not necessarily observed in the

same order by any processor. LC represents the state of a memory location as a partially ordered multiset of write and synchronization operations. For the LC model to be able to provide this partial ordering of writes and synchronization operations it requires an accompanying cache consistency model which is not provided by traditional multi-processor systems. Because writes to the same location are not serialized the programming model associated with using LC is complicated and adds a significant burden on the programmer.

To summarize, programmability and performance are two ends of a spectrum. The traditional approach in the past to enable performance on parallel platforms was to use a relaxed consistency model. However, weaker consistency models achieve performance by sacrificing programmability. In our approach, to support the familiar memory consistency model expected by today's shared memory programmers, we provide a strong consistency model. However, we believe most of the performance can be recovered by a consistency model that enables one to develop intelligent runtime systems that support it and by providing programmers with extensions to the programming model that can leverage intrinsic information available only at runtime.

Chapter 3

Regional Consistency (RegC) Model

Virtual shared memory systems present shared memory semantics over non-cache-coherent memory. The scalability and overhead of a virtual shared memory system largely depends on its memory consistency model. The shared global address space provided by a virtual shared memory system is actually distributed across independent memories. To improve performance in such a scenario, the standard technique used is data replication. However, we require a sophisticated memory consistency model to maintain consistent copies of these distributed data.

There are two fundamental aspects of memory required for writing correct concurrent applications. The first aspect, *coherence*, defines what values can be returned by a read to a memory location. The second aspect, *consistency*, determines when a write to a memory location will be returned by subsequent reads to the same memory location. In the rest of this section P_i refers to processor i.

Memory is said to be coherent if the following conditions hold:

- A read by processor P_i to a memory location M following a write by P_i to M, with no other writes to M by any processor between the write and the read, always returns the value written by P_i.
- A read by a processor P_i to a memory location M that follows a write by another processor P_j to M returns the written value if the read and write are sufficiently separated and no other

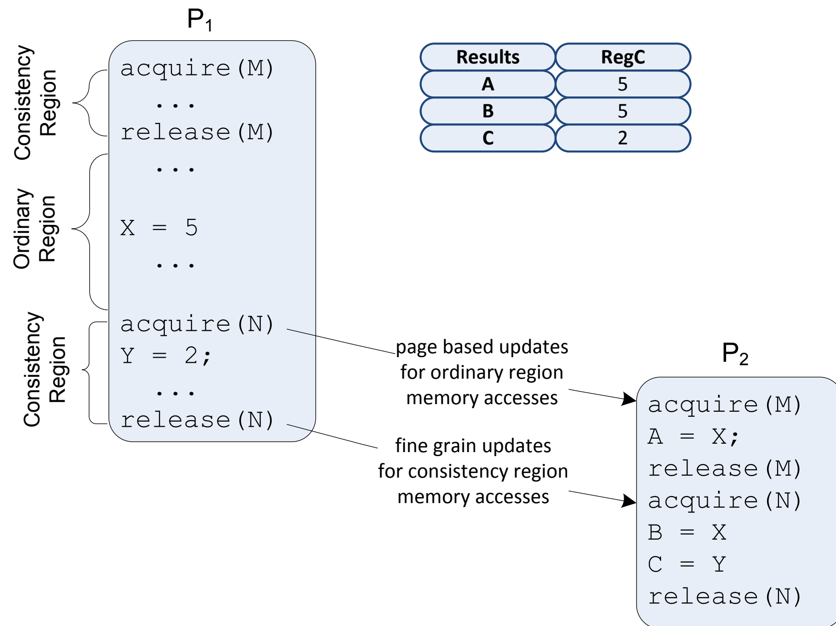


Figure 3.1: Pseudo code describing regional consistency.

writes to M occur between the two accesses.

- Writes to the same memory location are serialized, and every processor sees the same order.

This knowledge of memory coherence alone is insufficient for a programmer to write correct concurrent applications. To write correct concurrent applications, a programmer needs to have knowledge of the memory consistency model as well. A memory consistency model determines when a written value to a memory location is made available to be read. Hence, the memory consistency model presents a programmer with predictable behavior of memory accesses. This behavior is determined by formal rules, which if followed, guarantee memory to be consistent.

3.1 Regional consistency

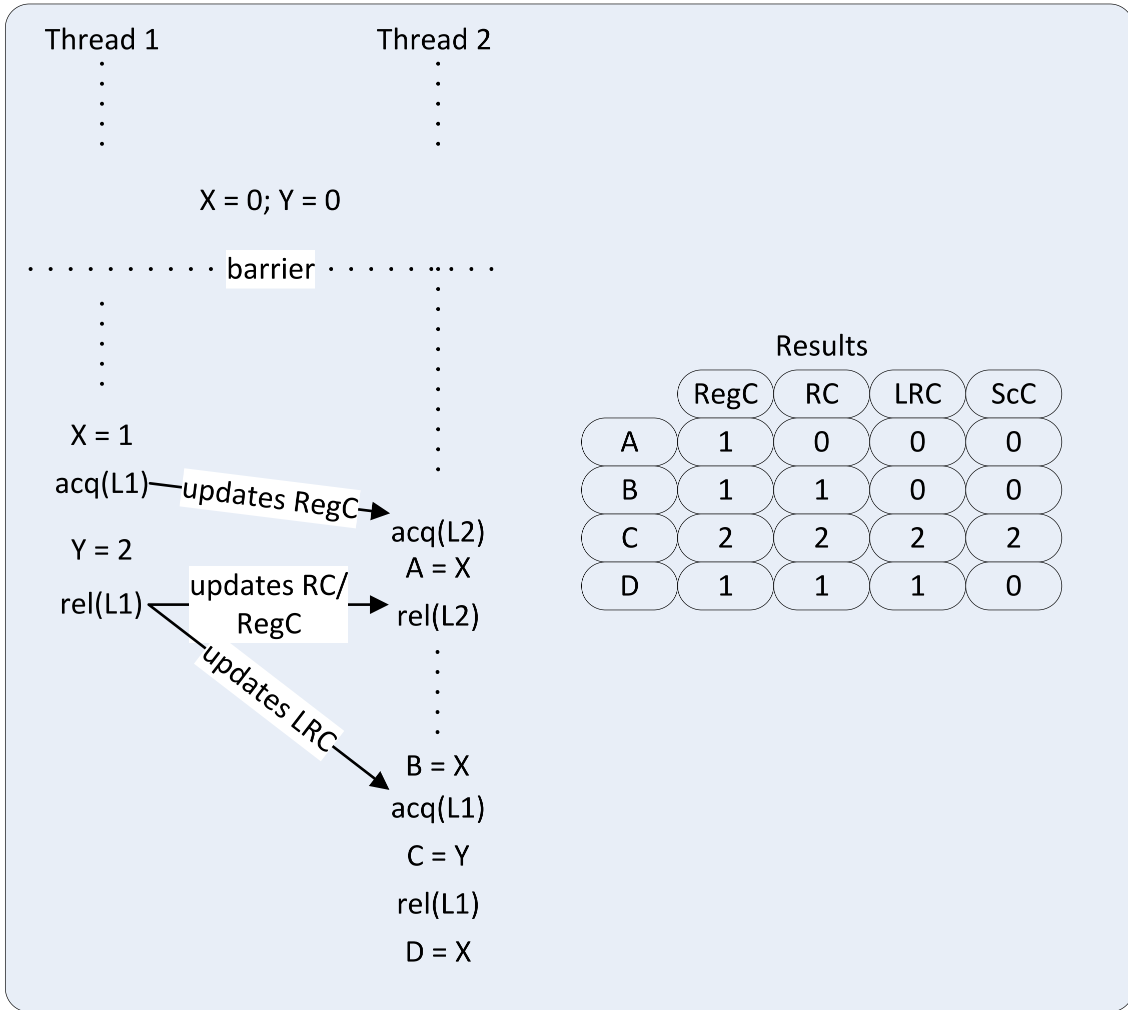
Before giving a formal definition of our new consistency model, we describe the basic idea and how it compares to similar models. The idea behind *regional consistency* (RegC) is to divide an application's memory accesses into two kinds of regions—*consistency regions* and *ordinary regions*—as depicted in Figure 3.1. These regions are demarcated by synchronization primitives utilizing mutual exclusion (mutex) locks and barriers. More specifically, a consistency region is demarcated

by a mutex lock acquire and release. All memory accesses outside of a consistency region occur in an ordinary region. When a barrier occurs in an ordinary region, it splits that region in two, i.e., one ordinary region ends at the barrier and a new one begins after the barrier.

The RegC rule for barriers is simple: all modifications made in the preceding ordinary region are made consistent for the processors participating in that barrier. To describe the RegC rules for consistency regions, we first define a *span* as one instance of a consistency region that executes at a given processor. A span starts at the acquire of a mutex lock and ends on the successful release of that lock. Any modification to data made in a span will be visible to processors that subsequently enter spans corresponding to the same mutex lock. Note that spans corresponding to different locks are independent, i.e., they can execute concurrently. Different spans can also be nested, corresponding to nested critical sections. Finally, modifications made in the preceding ordinary region are propagated on the start of a span. RegC guarantees that these updates will be visible at other processors before the start of *any* span corresponding to *any* consistency region.

Regional consistency can be viewed as an amalgamation of release consistency (RC) and scope consistency (ScC). Similar to ScC, we transparently detect data modification within a consistency region and implicitly associate it with corresponding locks, thereby creating the dichotomy of ordinary and consistency accesses. Similar to RC, we ensure that updates from ordinary regions are propagated on lock acquisition/release, not just on explicit barrier operations. We believe that performing updates from ordinary regions only on explicit barriers is unduly restrictive, i.e., it limits parallel problem decomposition to block synchronous codes. For other common parallel decompositions (e.g., producer/consumer, pipeline) superimposing barrier semantics creates unnecessary synchronization between unrelated threads and increases false sharing.

The general view is that relaxing consistency models improves performance but at the cost of programmability. Since our goal with RegC is to maintain the familiarity of the strong consistency model expected by thread-based programs, the challenge is to allow for a performant implementation of the consistency model. Both RegC and RC provide a sufficiently strong model for writing correct threaded code, compared to ScC. The differences between RegC and RC allow significant performance opportunities for RegC. Explicitly distinguishing between memory modifications made



	RegC	RC	LRC	ScC
A	1	0	0	0
B	1	1	0	0
C	2	2	2	2
D	1	1	1	0

Figure 3.2: Comparison of RegC, RC, LRC and ScC

inside a consistency region and those made outside allows an implementation of RegC to delay updates made in ordinary regions, which RC cannot (LRC, which makes a similar optimization, is less intuitive to programmers than RegC.) Furthermore, the distinction allows a RegC implementation to use different update policies to propagate the modifications in ordinary and consistency regions, i.e., page-based invalidation policy for ordinary regions and fine grained updates for consistency regions.

Figure 3.2 shows a comparison between regional consistency (RegC), release consistency (RC), lazy release consistency (LRC), and scope consistency (ScC). In this example, we assume that the code shown is in program execution order for one particular execution. ScC will result in stale

values for A, B and D because X is updated in an ordinary region. LRC results in stale values for A and B because the updates related to $rel(L1)$ are delayed till the subsequent $acq(L1)$. The value for A is stale for execution under RC because the updates are propagated at $rel(L1)$ and this occurs after the execution of $A = X$ in *Thread* 2. RegC is stronger than RC in that the write to X is propagated on $acq(L1)$ and is guaranteed to be visible on the next acquire of any lock, in this case $acq(L2)$ in *Thread* 2. By propagating updates on both acquisition and release, RegC ensures that all updates sent by RC are also sent by RegC. However, because the granularity for updates sent for modification within a consistency region are at the granularity of individual objects, RegC has lower overhead than RC which propagates updates at the granularity of a page.

3.2 Formalizing RegC

To define RegC formally we use the formal definitions for the memory access transitions presented in [72]. For the purpose of completeness we include these definitions here:

Definition 1. Performing with respect to a processor. A *LOAD* by processor P_i is considered *performed* with respect to P_k at a point in time when the issuing of a *STORE* to the same address by P_k cannot affect the value returned by the *LOAD*. A *STORE* by P_i is considered *performed* with respect to P_k at a point in time when an issued *LOAD* to the same address by P_k returns the valued defined by this *STORE* (or a subsequent *STORE* to the same location).

Definition 2. Performing an access globally. A *STORE* is *globally performed* when it is *performed* with respect to all processors. A *LOAD* is *globally performed* if it is *performed* with respect to all processors and if the *STORE* that is the source of the returned value has been *globally performed.*

In addition to the above two standard definitions we propose the following new definition.

Definition 3. Subsequently after. A *span* for any *consistency region* at P_j is said to start *subsequently after* a *span* for any *consistency region* at P_i if and only if the *span* has successfully started at P_i before the *span* at P_j successfully starts. Note that a span only successfully starts when the corresponding lock acquisition succeeds.

Before we define the RegC model formally, we distinguish a *STORE performed* with respect to the regions of memory accesses as follows:

- A *STORE performed* within a *consistency region* is defined as a *consistent STORE*.
- A *STORE performed* outside of a *consistency region* is defined as an *ordinary STORE*.

Furthermore, we distinguish a *consistent STORE* being *performed* with respect to a *consistency region* from a *STORE* being *performed* with respect to a processor as follows:

- A *consistent STORE* is *performed* with respect to a *consistency region* when the current *span* of that *consistency region* ends.
- A *STORE* is *performed* with respect to P_i if a subsequent *LOAD* issued by P_i returns the value defined by this *STORE* (or a subsequent *STORE* to the same memory location).

Finally, given these definitions, the rules for *regional consistency* are as follows:

1. Before a new *span* of a *consistency region* is allowed to successfully start on P_j subsequently after a *span* of any *consistency region* on P_i, any ordinary *STORE* performed at P_i before that *span* on P_i must be performed with respect to P_j.
2. Before a new *span* of a *consistency region* is allowed to successfully start at P_i, any *consistent STORE* previously *performed* with respect to that *consistency region* must be *performed* with respect to P_i.
3. A *STORE performed* at P_i must be *performed* with respect to P_j, for all P_i and P_j participating in a barrier.

The first rule determines when an *ordinary STORE* is performed with respect to a processor. An *ordinary STORE* is performed with respect to a processor P_j before any *span* is allowed to start at P_j, provided this *span* starts *subsequently after* the *span* immediately following the *STORE* at processor P_i. The second rule ensures that when a new *span* starts at a processor, any *consistent STORE performed* previously with respect to that *consistency region* is guaranteed to be *performed*

with respect to the processor. The third rule guarantees that any *STORE* performed before the start of a barrier is performed with respect to all processes participating in that barrier.

Chapter 4

Samhita Architecture

The main motivation for the architecture of Samhita is

1. To provide multi-threaded applications with a consistent view of a global shared address space.

2. To provide single-threaded applications access to a large address space.

These goals are achieved by a novel approach wherein we separate the notion of serving memory from the notion of consuming memory for computation. Figure 4.1 depicts Samhita's architecture, which consists of three main components – compute servers, memory servers and resource managers. These components run on the nodes of a cluster, which are interconnected using a high performance interconnection network, e.g., InfiniBand switched fabric, Myrinet, 10 Gigabit Ethernet. Figure 4.2 brings out the similarity in how Samhita would be implemented for a cluster and a heterogeneous system. In Figure 4.2a we can observe that compute servers, manager and memory servers execute on individual nodes of a cluster supercomputer. In the scenario of a heterogeneous system we execute the manager and memory server on the host CPUs and the compute servers execute on the coprocessors, as presented in Figure 4.2b. The role of each component in Samhita is as follows:

Compute servers execute one or more threads of control from one or more applications and access memory served by the memory servers. Multiple threads of an application may execute on different compute servers and have access to a consistent global address space – from the application's perspective processing cores on a distributed memory compute server appear as individual cores of

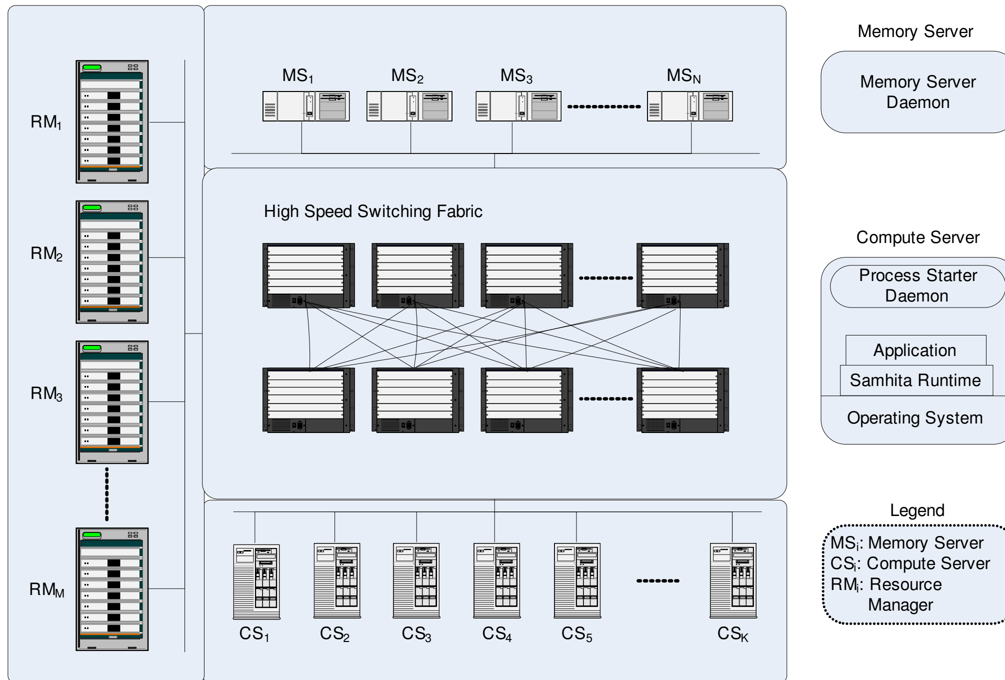
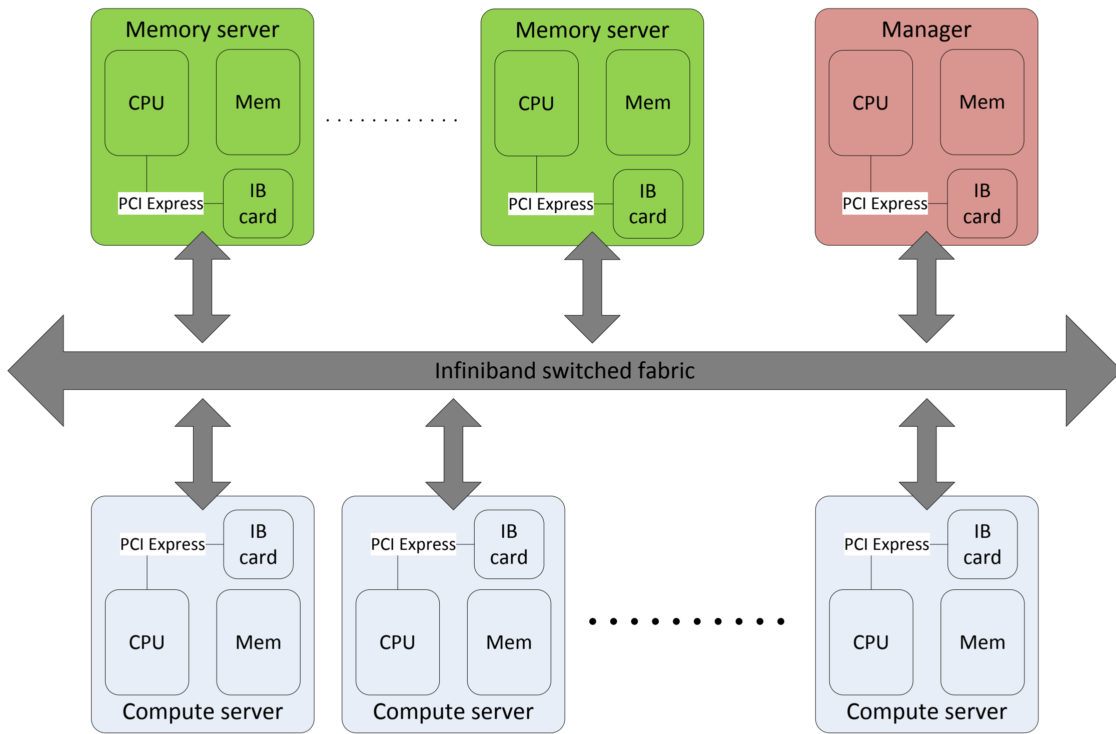


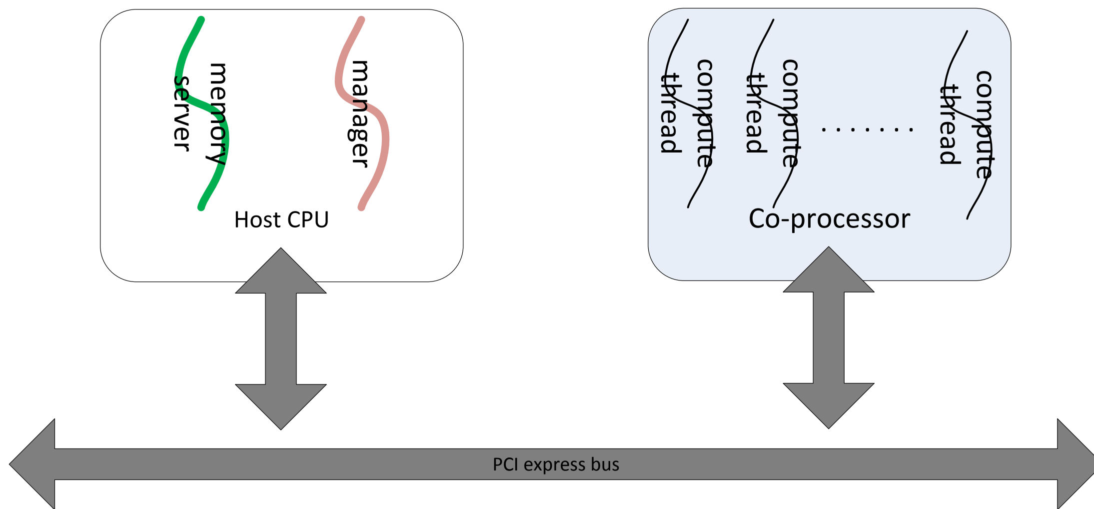
Figure 4.1: Architectural view of Samhita DSM system consisting of memory servers and resource managers, which provide a consistent shared address space for the compute servers.

a shared memory system. Samhita exposes a *fork-join* execution model similar to POSIX threads. An application starts with one thread of execution, which then spawns other threads. Placement of threads onto compute servers is handled by the resource manager.

Memory servers act as a pool to provide the global address space required by an application, with each memory server responsible for one or more allocations made in the global address space. The memory servers at any instant serve memory for one or more applications which are executing on the compute servers. In Samhita, each page of the global address space is served by a single memory server, creating a "home based" protocol. The maximum size of the global address space is equal to the amount of memory exported by the pool of memory servers. Samhita's global address space is partitioned among the memory servers. The resource manager is responsible for co-ordinating memory allocation requests from compute servers and allocating a set of memory servers to satisfy each allocation. To mitigate the impact of hot spots, large memory allocations are strided across multiple memory servers. This operation is depicted in Figure 4.3a.



(a) Architecture of Samhita for a cluster.



(b) Architecture of Samhita for a heterogeneous system.

Figure 4.2: Similarities between the implementation of Samhita for a cluster and heterogeneous system.

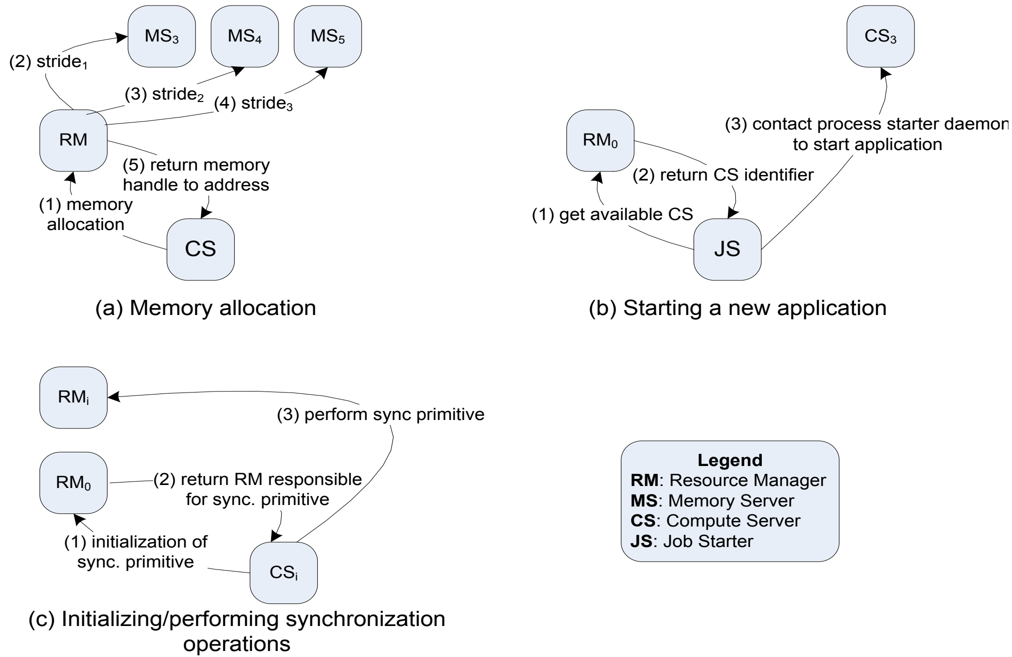


Figure 4.3: Sequence of actions between compute servers, memory servers and resource manager for (a) memory allocation, (b) job startup and (c) synchronization operations.

Resource managers co-ordinate the actions of compute servers and memory servers and provide the core functionality for implementing synchronization primitives. The resource manager is primarily responsible for

Job startup: A well known resource manager is responsible for co-ordinating the startup of a new application and allocating an available compute server. The application starter first contacts the resource manager to retrieve a reference to a compute server and then contacts the process starter component running on each compute server to start the application. This action is depicted in Figure 4.3b.

Thread placement: The resource manager is responsible for allocating processing cores on compute servers to new threads. When a compute server starts a new thread, the Samhita run-time system contacts the resource manager to get the compute server on which to start the new thread and uses the process starter component to instantiate a new copy of the application. The Samhita run-time system maps the data section (*.data* and *.bss* sections) of the application binary into the global address space to make it a thread of the application instead of a

complete new instantiation.

Memory allocation: The resource manager is also involved in allocating memory in the shared global address space. When a compute server wants to allocate memory in the shared address space it contacts the resource manager. The resource manager then ensures that the allocation is strided across the memory servers to reduce hot spots. This operation is depicted in Figure 4.3a.

Synchronization operations: These operations include lock acquisition and release, barrier synchronization and condition variable signaling. Since these operations are relatively frequent and can easily become a bottleneck, Samhita uses a scale-out architecture with multiple resource managers to improve the performance of synchronization operations. During the initialization of a synchronization variable, a well known resource manager returns a reference to a particular resource manager that is responsible for handling all synchronization operations on that variable. The sequence of actions for this is depicted in Figure 4.3c.

Chapter 5

Implementation of Samhita

The implementation of Samhtia consists of three components: the memory server, the resource manager and a run-time system that runs on each compute node. The memory servers and resource managers are implemented as daemons. They run on the memory server and resource manager nodes respectively in our cluster implementation, and on the host CPUs in our heterogeneous architecture implementation. The run-time system is a shared library that programmers link with the application. The implementation of the components is driven by two concerns – portability and low overhead. For a software system to be portable it should not require any modification to the operating system kernel and should depend only on standard system libraries. The current implementation of Samhita runs on the GNU/Linux family of operating systems. Samhita depends only on standard system libraries, namely *glibc* (GNU C library implementation of the C standard library [8]) and *libelf* (part of elfutils [7]). Samhita requires that *glibc* support the Native POSIX Thread library [33] implementation of POSIX Threads. These requirements are available in all modern GNU/Linux distributions. To reduce overheads, Samhita interfaces directly with the interconnection network hardware. To support multiple interconnection networks, we abstract the interconnection network layer to expose a common interface, the Samhita Communication Layer (SCL). Section 5.1 describes the implementation of SCL.

In the rest of this chapter we describe the management of local cache associated with each compute thread in Section 5.2. Section 5.3 describes the global address management in Samhita.

False sharing is concern for any virtual shared memory system, we describe the memory allocation strategies used in the implementation of Samhita to reduce its impact in Section 5.4. To maintain consistency of the global address space, Samhita provides barrier synchronization, condition variable signaling and mutual exclusion locks as synchronization primitives. We describe Samhita's barrier synchronization protocol implementation in Section 5.5. We describe the store instrumentation technique used to support fine grain updates for consistency regions in Section 5.6. The Samhita API is presented in Section 5.7.

5.1 Samhita communication layer (SCL)

The Samhita Communication Layer (SCL) provides a minimal set of APIs required for Samhita to interact with any interconnection network. In principle SCL is similar to the ADI layer of MPICH [43]. The primary design decision in SCL is the fundamental abstraction for communication – either a serial protocol similar to MPI or a new direct memory access based protocol. The key observation here is that implementing a serial line abstraction over memory based hardware such as RDMA imposes additional protocol overhead in exchanging information on registered memory. In contrast, a memory based communication layer can be implemented efficiently over physical serial line hardware such as TCP/IP 10 Gigabit Ethernet. This reasoning led us to choose a communication abstraction based on direct memory access for SCL.

Table 5.1 lists the APIs provided by SCL, which can be categorized as follows:

Connection management. APIs to set up and destroy point-to-point connections and create and destroy multicast groups.

Send/receive. APIs for sending small messages of fixed size. SCL implements both blocking and non-blocking versions of the send primitive. The size of the message is determined by the interconnection network. These messages are useful for control messages.

Get/put. APIs for transferring data of arbitrary length exploiting direct memory transfer. SCL implements both blocking and non-blocking versions of the get and put primitive.

Test/wait. APIs for testing the completion of any non-blocking primitive.

Table 5.1: Samhita Communication Layer API

Description	API
Initialize SCL	scl_initialize(scl_init_attr_t *attr, scl_status_t *status)
Terminate SCL	scl_finalize(void)
Create a point-to-point connection	scl_connect(enum SCL_CONNECTION_TYPE type, const char *host, scl_cx_attr_t *attr, scl_status_t *status)
Close a connection	scl_disconnect(void *cx_id)
Poll for incoming connections	scl_poll_connections(scl_status_t *status)
Select for incoming connections	scl_select(enum SCL_SELECT_RESULT *result, uint64_t timeout, scl_status_t *status)
Create a multicast group	scl_create_mcast_grp(void **cx_ids, int count, scl_status_t *status)
Destroy multicast group	scl_destroy_mcast_grp(void *mc_id)
Receive a message (non-blocking)	scl_recv(void *data, scl_status_t *status)
Send a message (blocking)	scl_send(void *cx_id, uint64_t *dest, void *data, scl_status_t *status)
Send a message (non-blocking)	scl_send_nb(void *cx_id, uint64_t *dest, void *data, scl_status_t *status)
Get a memory region (blocking)	scl_get(void *cx_id, uint64_t *src, uint8_t buf, uint64_t *dest, size_t len, scl_status_t *status)
Get a memory region (non-blocking)	scl_get_nb(void *cx_id, uint64_t *src, uint8_t buf, uint64_t *dest, size_t len, scl_status_t *status)
Put a memory region (blocking)	scl_put(void *cx_id, uint64_t **dest, uint8_t buf, uint64_t *src, size_t len, scl_status_t *status)
Put a memory region (non-blocking)	scl_put_nb(void *cx_id, uint64_t **dest, uint8_t buf, uint64_t *src, size_t len, scl_status_t *status)
Test for non-blocking work request completion	scl_test(uint64_t wr_id, scl_status_t *status)
Wait for non-blocking work request completion	scl_wait(uint64_t wr_id, scl_status_t *status)
Allocate from SCL atomic data	scl_atomic_alloc(scl_status_t *status)
Free SCL atomic data	scl_atomic_free(scl_atomic_t *ptr)
Allocate from SCL memory	scl_malloc(enum SCL_MALLOC_TYPE type, scl_status_t *status)
Free SCL memory region	scl_free(scl_malloc_t *ptr)
Register memory with SCL	scl_reg_memory(void *ptr, size_t size, scl_status_t *status)
De-register registered memory region	scl_dereg_memory(void *mr, scl_status_t *status)
Create a memory handle	scl_create_mh(void *addr, void *mr, uint64_t *mh, scl_status_t *status)
Get buffer associated with memory handle	scl_get_mh_buffer(void *mr, scl_status_t *status)
Perform a SCL atomic fetch and add	scl_fetch_add(void *cx_id, uint64_t *dest, uint64_t *src, uint64_t val, scl_status_t *status)

Memory allocation. APIs to allocate registered memory from a pre-allocated pool. The size of the pool can be configured at the initialization of SCL.

Memory registration. APIs to allow the application programmer to register arbitrary memory regions with the interconnection network hardware for communication operation.

Atomic operations. API for atomic fetch and add operation is currently supported. SCL uses the underlying interconnection network operation when available.

Since SCL exposes memory based semantics for communication, the sending side needs to know the memory address at the destination. Ordinarily, this would require additional communication to exchange this information. To avoid this scenario, SCL associates a set of default buffers with each point-to-point connection. These buffers can be used for communication if the memory handle of the destination location is unknown. Buffer management (flow control) of these default buffers is performed at the higher layer.

The current implementation of SCL supports the Quad Data Rate (QDR) InfiniBand hardware. Support for InfiniBand is provided using the OpenFabrics Enterprise Distribution (OFED) [10] software stack version 1.3 or higher. InfiniBand provides four types of transport services: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). Our SCL implementation uses the RC transport service, which enables us to take advantage of reliable RDMA operations to perform memory-mapped communication. To establish an RC transport service, each end-point of the connection must create a RC Queue Pair (QP). SCL abstracts the complexity of setting up connections, shared receive queues and associated buffers into a single simplified connection setup call.

5.2 Samhita cache management

The global address space in Samhita is divided into pages. The page size is currently set to the operating system virtual memory page size of $4K$. Each page of the global address space is served by a single memory server. Since Samhita operates at the granularity of pages, the impact of false sharing is exaggerated. To reduce the impact of false sharing Samhita utilizes a *multiple-*

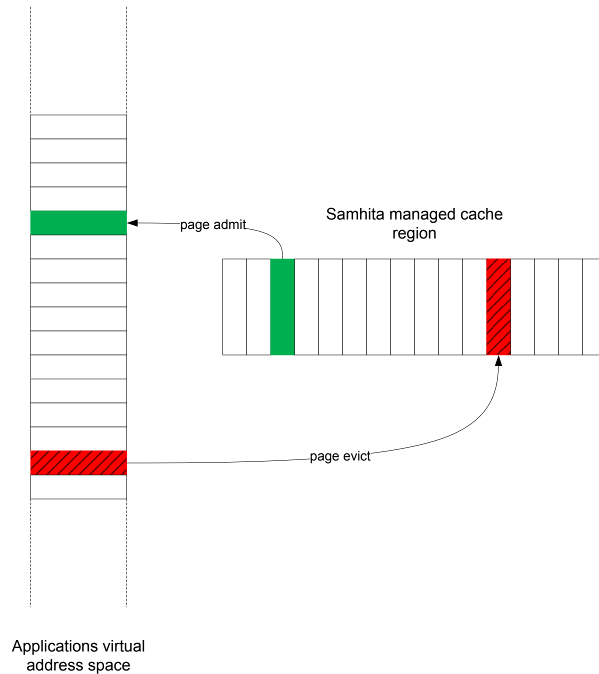


Figure 5.1: Remapping of physical pages from Samhita cache region to an application's virtual address space during page admit, and remapping physical pages from an application's virtual address space to the Samhita cache region on page evict.

writer protocol. Whenever an application thread[1] accesses a memory location in the global address space, the access is through its local cache. Each application thread has its own cache. If the page corresponding to the memory location is not present in the local cache, the run-time system gets a copy of the page from the corresponding memory server and stores it in the local cache. To reduce the number of misses for applications that exhibit spatial locality, we use cache lines consisting of multiple pages. In essence the Samhita run-time uses the concept of demand paging to populate the local cache.

We also use anticipatory paging, or prefetching, to exploit spatial locality. On encountering a cache miss, Samhita places a request for the missing cache line and an asynchronous request for the adjacent cache line. In principle, this design is similar to modern microprocessors, where the missing cache word is brought in synchronously, with the rest of the cache line being populated asynchronously by the cache controller.

[1]Each Samhita compute thread is actually a process, but we use the term thread throughout

There are two relatively straightforward approaches for implementing data retrieval on a cache miss:

1. The run-time system could first fetch a copy of the page from the memory server into a well known buffer and then copy the data from the buffer into the application address space.

2. The run-time system could allocate the missing page in the application address space, register it with SCL and fetch the page data from the remote memory server directly into the application address space, thereby avoiding the cost of the copy operation.

Interestingly, both approaches take tens of microseconds for a cache line, an unacceptably high overhead when the actual data transfer takes almost the same time. Instead, Samhita implements a novel approach based on changing the mapping between virtual addresses and the underlying physical memory address using the `mremap` [15] system call. This technique is depicted in Figure 5.1. In this approach, the run-time uses a pre-allocated cache region, which is registered with SCL. When the run-time system encounters a cache miss it fetches a copy of the page into the pre-allocated region. After the copy is fetched we use the `mremap` system call to modify the mapping of the physical pages from the cache region into the faulting address in the application's virtual address space. This greatly reduces the overhead of fetching pages from the memory server from tens of microseconds to about 4μs/cache line. When we evict a page due to a capacity miss, the run-time re-maps the physical pages from the application's virtual address space back into the cache region managed by the Samhita run-time system.

In Samhita every page in the local cache can exist in one of two modes – *read-only* or *read-write*. The granularity of the local cache in Samhita is a cache line. Similar to a mode associated with any page, every cache line belongs either to a *read* list or a *write* list. A cache line belongs to the *read* list if every page of the cache line is cached in *read-only* mode. A cache line belongs to the *write* list if one or more pages of the cache line are cached in *read-write* mode.

In Samhita the cache eviction policy is biased against pages that exist in *read-only* mode in the cache. The victim for page eviction is a cache line from the *read* list, selected using the first-in, first-out (FIFO) algorithm. However, when the number of cache lines in the *read* list falls below a

threshold the victim is then selected from the *write* list. The victim for eviction is again selected using the FIFO algorithm, i.e., the cache line that was added first to the *write* list is the victim chosen for eviction.

5.3 Samhita global address space management

Management of the shared address space in Samhita involves detecting cache misses and transitions from *read-only* to *read-write* modes. This is achieved by using the *mprotect* system call in conjunction with a *SIGSEGV* handler. The *SIGSEGV* signal is raised by the operating system for any invalid memory access. We install a signal handler for the *SIGSEGV* signal, which is called every time the signal is generated. We describe the following cause-effect scenarios for a *SIGSEGV* signal to be generated in our run-time system.

1. The application tries to read a memory location in the Samhita global address space, but the page associated with this location is not present in the local cache. When the signal handler is called the Samhita run-time system requests a copy of the page from its corresponding memory server and a copy is cached in *read-only* mode. The faulting instruction is restarted and the second access is successful.

2. The application tries to write to a memory location in the Samhita global address space, but the page associated with this location is not present in the local cache. When the signal handler is called the run-time system requests a copy of the page from its corresponding memory server and a copy is cached in *read-only* mode. The faulting instruction is restarted.

3. The application tries to write to a memory location in the Samhita global address space, and the page associated with this location exists in the local cache but in *read-only* mode. The signal handler creates a twin of this page and the copy is upgraded to *read-write* mode. The faulting instruction is restarted and the next access is successful.

If a page in the local cache is in *read-write* mode, updates made to the page need to be propagated to the corresponding memory server when the page is evicted from the cache. Recall that Samhita

uses a *multiple-writer* protocol to reduce the impact of false sharing, which requires a mechanism to accurately detect the write set within a page. To compute the write set of a page we use the *exclusive disjunction* (XOR) difference between the current copy of the page and the copy of the page on the memory server. This XOR difference is called the *page diff* of a page.

To compute a *page diff* we need a copy of the original page as fetched from the memory server. The run-time system makes a copy of the page in its memory whenever a page transitions from *read-only* mode to *read-write* mode. This copy of the page is called the *page twin*. The *page diff* is computed as an XOR difference between the *page twin* and the modified copy of the page being evicted. This *page diff* is then applied by the memory server to update its copy. The updated copy at the memory server is generated by using the XOR operation on the copy of the page at the memory server and the *page diff* received from the compute thread. Using the XOR operation only those bits that have been modified are changed on the memory server.

When an application using Samhita is executed, the run-time system maps the data segment of the application binary to the shared global address space transparently. When a new compute thread is created the run-time system removes the existing data segment of the new instantiation using the `munmap` [16] system call. Subsequent access to a memory location in the data segment by the new thread will result in a *SIGSEGV* signal, which causes the Samhita signal handler to be called. The run-time system treats this access as any other access to the shared global address space. In this way, the processing cores of individual compute servers appear as individual cores of a shared memory system.

5.4 Memory allocation strategies

False sharing affects the performance of any system that uses cache. In the scenario of virtual shared memory systems the impact can be greatly magnified due the granularity of sharing. Memory allocation strategies have an effect on false sharing. This is particularly noticed in applications that allocate memory and modify it in one thread and read data from these memory regions in other threads. If multiple threads behave in this manner it can cause increased false sharing as memory

allocated by different threads can reside in the same page, the granularity of sharing in Samhita. This kind of allocation pattern is common in applications that use master-worker queues or pipelined stages in their processing.

Samhita allocates memory using three strategies to reduce both false sharing and the cost of memory allocation. The strategy used to satisfy a memory allocation request is based on the size of the allocation request. The strategies used are as follows:

Small allocations: For small allocations, the allocation is satisfied from *arenas*. These arenas are associated with each thread of the application. These allocations are handled locally by the compute thread instead of contacting the manager for each allocation request. When a compute thread requests for a memory allocation, the run-time system allocates it from the arena associated with the thread. If the memory allocation request cannot be satisfied using the arenas currently available at a compute thread, the run-time system contacts the manager for a new arena. Once the run-time system finds an arena that can satisfy the request it allocates the memory from that arena. This strategy not only reduces false sharing but also reduces the cost to satisfy small allocation requests. The manager strides the memory associated with these arenas across multiple memory servers. In our current implementation any allocation smaller than 4KB is satisfied using local arenas.

Medium allocations: For medium size memory allocations, the allocation is satisfied from *zones* that are shared across all threads. The run-time system contacts the manager for these allocations. When the manager gets an allocation request it allocates it from one of the available zones. If there are no zones that can satisfy this request, the manager creates a new zone and satisfies the request from the newly created zone. This approach reduces both memory fragmentation and reduces the cost of memory allocation by not having to contact the memory servers for each individual memory allocation request. The memory associated with the zone is strided across multiple memory servers. This approach is used when the memory allocation requested is greater than 4KB but less than 1MB in size.

Large allocations: For large memory allocations, the allocation is once again satisfied by contacting the manager. The manager strides the memory allocation across multiple memory servers. Whenever an application frees the memory associated with a large allocation, if it is sufficiently large to satisfy an arena or zone memory region, then the manager does not free this allocation at the memory servers. Instead, it adds the allocation to a free list, which it uses later to satisfy the memory required for a new arena or zone. This strategy is used when memory allocation request size is greater than 1MB.

5.5 Barrier synchronization in Samhita

Barrier synchronization is a common primitive used by Bulk Synchronous Parallel (BSP) [74] applications. Most BSP applications implemented in message passing iterate over the following three stages:

1. Each compute thread performs a concurrent independent computation. The data used in the computation are stored in the local memory, or in the case of Samhita, the local cache.
2. Each thread then exchanges data with potentially every other thread.
3. All threads synchronize at a barrier to ensure completion of the prior two stages.

In Samhita synchronization barriers are also memory barriers, i.e., the run-time system enforces ordering of memory accesses before and after the barrier. This results in updates to the global address space during the synchronization operation. In a shared memory system since the entire global address space is visible to every thread, step 2 above transforms a barrier into a memory synchronization operation.

In Samhita when a barrier synchronization variable is created the programmer specifies the number of threads that will participate in the synchronization operation. Since we implement a memory barrier, the barrier synchronization protocol in Samhita involves the following stages:

1. Every participating compute thread sends the changes made to the pages in its local cache to

the appropriate memory servers to update the global address space. In essence, this is similar to the data exchange stage of a BSP algorithm.

2. Wait until all the participating compute threads have finished updating the global address space. This is essentially the last stage of a BSP algorithm.

The first stage of the barrier synchronization protocol can be very expensive because the run-time system on each compute node has no global knowledge of the changes made to pages on other nodes. This lack of knowledge can significantly impact the performance of any application using Samhita. Samhita implements a novel barrier synchronization protocol that achieves the lower bound for synchronization traffic, which is the set of pages that have been modified on more than one compute node.

To achieve this, every page of the global address space is associated with an *epoch*, where an *epoch* is a monotonically increasing counter with the following properties:

- When the run-time system fetches a page from a memory server because of a cache miss, the server sends its current *epoch* along with the copy of the page.
- The *epoch* associated with a page is incremented at the compute thread every time the mode of the page transitions from *read-only* to *read-write*.
- The *epoch* associated with a page is incremented at the memory server every time it receives a *page diff* for the page.

The barrier synchronization protocol in Samhita is implemented in two parts. One part of the protocol involves the compute thread and the other part involves the memory servers. Algorithm 1 presents the pseudo code of the implementation at the compute thread while Algorithm 2 presents the pseudo code of the implementation at the memory servers.

When an application enters the barrier synchronization call, every compute thread first sends the lists of pages in its cache that are in *read-write* mode to the respective memory servers. When a memory server receives this list from a compute thread it increments the *epoch* for the pages in the list. The memory server then generates a list of pages to be invalidated and sends it to the compute

Algorithm 1 Pseudo code of barrier synchronization protocol implemented at the compute thread.

1: Wait for all threads involved in the barrier to check in with the manager.
2: **for** $i = 1$ to memservs **do** {memservs is the total number of memory servers}
3: Send the list of pages in *read-write* mode that are served by memory server i to memory server i.
4: **end for**
5: Wait for all threads to finish sending list of *read-write* pages to respective memory servers.
6: **for** $i = 1$ to memservs **do**
7: Send the list of pages in *read-write* mode along with their *epoch* that are served by memory server i to memory server i.
8: Receive the list of pages to invalidate. Invalidate the pages and send the *page diffs* for the invalidated pages to memory server i.
9: Send the list of pages in *read-only* mode along with their *epoch* that are served by memory server i to memory server i.
10: Receive the list of pages to invalidate. Invalidate the pages.
11: **end for**
12: Wait for all threads to finish invalidating pages.

Algorithm 2 Pseudo code of barrier synchronization protocol implemented at the memory server, for a give compute thread c.

1: **for all** *page*s in the list of *read-write* pages received from a compute thread c **do**
2: Increment the *epoch* of the *page* if the page is not *invalid* at the memory server.
3: **end for**
4: **for all** *page*s in the list of *read-write* pages received from a compute thread **do**
5: **if** *epoch* for *page* at memory server $>$ *epoch* at compute thread **then**
6: Add *page* to invalidate list.
7: **else if** *epoch* for *page* at memory server $=$ *epoch* at compute thread **then**
8: Mark page at memory server as *invalid*. Store the compute thread that has the current copy of *page*.
9: **end if**
10: **end for**
11: Send the invalidate list to compute thread c.
12: **for all** *page*s in the list of *read-only* pages received from a compute thread **do**
13: **if** *epoch* for *page* at memory server $>$ *epoch* at compute thread **then**
14: Add *page* to invalidate list.
15: **end if**
16: **end for**
17: Send the invalidate list to compute thread c.

thread. This list is generated by comparing the *epoch* of the page at the memory server with the *epoch* of the page at the compute thread. If the *epoch* at the memory server is greater, the page is added to the invalidation list. However, if the *epochs* are equal then the memory server marks its copy of the page as *invalid* and stores the compute thread which has the current copy in the data structure associated with the page. The invalidation list is then sent to the compute thread, which invalidates all the pages in the list and sends the *page diffs* for the invalidated pages to the corresponding memory server to update the copy of the page. This mechanism invalidates only the pages that exist in the cache in *read-write* mode on *more than one node*. To invalidate the pages that exist in *read-only* mode (this is needed because pages in read-only mode on one node may have been modified on another node), the compute server sends another list of pages that exist in *read-only* mode along with their *epochs* to the respective memory servers. When a memory server receives this list from a compute thread, it generates another invalidation list. To select the pages to invalidate, the memory server again compares the *epoch* of a page at the memory server with the *epoch* at the compute thread. If the *epoch* at the memory server is greater, the page is added to the invalidation list. The invalidation list is then sent to the compute thread which invalidates all the pages in the list. At the end of this stage every memory server either has a current copy of the page or knows the compute thread that has the current copy in its local cache.

When a compute thread encounters a cache miss after the barrier synchronization operation and requests a copy of a page that has been marked *invalid* at the memory server, then the memory server sends a request to the compute thread that caches the current copy of the page for a *page diff* to update the memory server copy. The compute server on receiving this request stops the execution of the compute thread and creates the *page diff* from the current copy of the page in its local cache and sends it back to the memory server. The compute thread also updates the copy of its *page twin* to the current copy in the local cache, increments the *epoch* of the page and resumes computation. The memory server when it receives the *page diff* updates its copy and marks the page as *current*. It then sends a copy of the updated page to the requesting compute thread along with the new *epoch*.

The compute thread does not invalidate the page or downgrade the protection of the page from *read-write* to *read-only* when it receives the request for the *page diff*. This barrier synchronization protocol reduces the set of invalidated pages and associated traffic to only those pages that have been modified on more than one compute node, which is the lower bound for communication overhead.

The time complexity of the barrier synchronization protocol depends on the memory access pattern of the application. For example the time complexity of step 2 of Algorithm 1 depends on the number of pages modified by the compute thread in its local cache. The number of pages in the local cache is small when compared to the total number of pages that the application accesses from the shared address space. Similarly the time complexity of step 4 of Algorithm 2 depends on the number of pages modified by two or more compute threads. The maximum number of pages in this case is at most the number of pages in the local cache,which is typically much smaller than the total number of pages that the applications accesses.

5.6 Store instrumentation for fine grain updates

One of the main motivations of the regional consistency (RegC) model is to allow for a performant implementation of the consistency model. In Samhita's implementation of RegC, ordinary region updates are propagated at the granularity of a page using an invalidate protocol. Consistency region updates are propagated at the granularity of objects using an update protocol. For Samhita to use fine grain updates, each store made in a consistency region must be tracked by the run-time system. For us to track individual consistency region stores we use static analysis to determine which stores are performed within a consistency region. We achieve this by using the LLVM compiler framework [55]. Our store instrumentation tool analyzes the application source code compiled to LLVM's intermediate representation (IR). We insert a function call to our run-time system passing the address of the store and size of the data type before each store performed in a consistency region.

Figure 5.2 presents an example of the store instrumentation performed. Figure 5.2a presents a code fragment in where a global variable *var* is incremented in a consistency region. Figure 5.2b shows the code compiled to LLVM's IR. Figure 5.2c shows the instrumented source code. The

```
/* Global variables */
int var;
smh_mutex_t m;

int main(int argc, char *argv)
{
      ...
      ...
      ...
      smh_mutex_lock(m, NULL);
      ++var;
      smh_mutex_unlock(m, NULL);
      ...
      ...
      ...
      return EXIT_SUCCESS;
}
```

(a) Source code

```
      ...
      ...
      ...
%1 = tail call i32 @smh_mutex_lock(i64* @m, i32* null) nounwind
%2 = load i32* @var, align 4, !tbaa !0
%3 = add nsw i32 %2, 1









store i32 %3, i32* @var, align 4, !tbaa !0
%4 = tail call i32 @smh_mutex_unlock(i64* @m, i32* null) nounwind

      ...
      ...
      ...
```

(b) Source compiled to LLVM IR

```
      ...
      ...
      ...
%1 = tail call i32 @smh_mutex_lock(i64* @m, i32* null) nounwind
%2 = load i32* @var, align 4, !tbaa !0
%3 = add nsw i32 %2, 1
%flag = load i32* @smh_in_lock_ctx, align 4
%ifcond = icmp eq i32 %flag, 1
br i1 %ifcond, label %then, label %4

then:                                        ; preds = %0
call void @smh_track_store(i8* bitcast (i32* @var to i8*), i8 4) nounwind
br label %4

; <label>:4                                  ; preds = %0, %then
store i32 %3, i32* @var, align 4, !tbaa !0
%5 = tail call i32 @smh_mutex_unlock(i64* @m, i32* null) nounwind
      ...
      ...
      ...
```

(c) Instrumented LLVM IR

Figure 5.2: Example of store instrumentation in Samhita (a) source code, (b) source compiled to LLVM IR and (c) instrumented LLVM IR

instrumentation tool analyzes all stores that are performed after the acquisition of a lock till the lock is released. It traverses all the basic blocks that occur between lock acquisition and lock release for stores that are performed. The instrumentation tool inserts a run-time check to verify that the store occurs in a lock context, and if so calls the instrumentation function before the store is performed. The instrumented code calls the instrumentation function with the address of the global variable *var* and the size of the variable, which is 4 bytes as the variable is of type integer.

The instrumentation tool only instruments stores that are performed on a global variable or a pointer. The instrumentation tool ignores any store performed to local stack variables. This ensures that only stores to global variables or variables that can alias global variables are instrumented. The run-time system associates the store performed with the mutex locks that are currently acquired. The run-time system does not track any stores that do not occur in the global address space that is managed by Samhita.

5.7 Samhita API

Samhita provides a minimal set of APIs that an application programmer can use to write threaded code that can take advantage of Samhita's virtual shared memory system. Table 5.2 lists the memory management, synchronization and thread management APIs provided by Samhita. Samhita provides APIs for dynamic memory allocation; these APIs are very similar to the memory allocation API provided by the standard C library. A comparison between the API provided by the standard C library and that provided by Samhita is presented in Table 5.3. The Regional consistency (RegC) model provides a sufficiently strong consistency model that makes it very easy to port existing threaded shared memory code to use Samhita. Samhita provides APIs for thread creation and management and also commonly used synchronization primitives, i.e., barrier synchronization, mutual exclusion locks and condition variable signalling. The Samhita APIs are very similar to the standard Pthread API [20] making porting most codes nothing but a "find and replace" operation. A comparison between the Pthread and Samhita APIs is presented in Table 5.4.

Table 5.2: Samhita API

Description	API
Memory allocation	smh_malloc(size_t size, smh_status_t *status)
Change the size of an existing allocation	smh_realloc(void *ptr, size_t size, smh_status_t *status)
Free memory allocation	smh_free(void *ptr)
Map an existing memory region into Samhita	smh_map(void *addr, size_t size, smh_status_t *status)
Create a Samhita thread	smh_thread_create(smh_thread_t *thread, void *(*start_routine) (void *), void *arg, smh_status_t *status)
Wait for thread termination	smh_thread_join(smh_thread_t thread, void **value_ptr, smh_status_t *status)
Get executing thread id	smh_gettid(void)
Create thread specific key	smh_key_create(smh_key_t *key, void (*destructor)(void *), smh_status_t *status)
Store value in key	smh_setspecific(smh_key_t key, const void *value, smh_status_t *status)
Get value from key	smh_getspecific(smh_key_t key, smh_status_t *status)
Create barrier synchronization variable	smh_barrier_create(smh_barrier_t *barrier, uint32_t nthreads, smh_status_t *status)
Barrier synchronize	smh_barrier_wait(smh_barrier_t *barrier, smh_status_t *status)
Create a mutex lock variable	smh_mutex_create(smh_mutex_t *mutex, smh_status_t *status)
Lock a mutex variable	smh_mutex_lock(smh_mutex_t *mutex, smh_status_t *status)
Unlock a mutex variable	smh_mutex_lock(smh_mutex_t *mutex, smh_status_t *status)
Create a condition variable	smh_cond_create(smh_cond_t *cond, smh_status_t *status)
Wait on condition variable	smh_cond_create(smh_cond_t *cond, smh_mutex_t *mutex, smh_status_t *status)
Signal a condition variable	smh_cond_signal(smh_cond_t *cond, smh_status_t *status)
Broadcast on a condition variable	smh_cond_broadcast(smh_cond_t *cond, smh_status_t *status)
Create a reduction operation variable	smh_redvar_create(smh_redvar_t *redvar, enum SMH_REDVAR_OP op, smh_status_t *status)
Perform a reduction operation	smh_reduce(smh_redvar_t *redvar, enum SMH_TYPE type, void *inval, void *outval, smh_status_t *status)

Table 5.3: Comparison of memory allocation API of C library and Samhita

Description	C library	Samhita
Allocate memory	malloc(size_t size)	smh_malloc(size_t size, smh_status_t *status)
Change allocation size	realloc(void *ptr, size_t size)	smh_realloc(void *ptr, size_t size, smh_status_t *status)
Free memory allocation	free(void *ptr)	smh_free(void *ptr)

Table 5.4: Comparison of Pthreads and Samhita API for thread management and synchornization

Description	Pthread	Samhita
Thread create	pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)	smh_thread_create(smh_thread_t *thread, void *(*start_routine) (void *), void *arg, smh_status_t *status)
Join with thread	pthread_join(pthread_t thread, void **retval)	smh_thread_join(smh_thread_t thread, void **value_ptr, smh_status_t status)
Initialize barrier	pthread_barrier_init(pthread_barrier_t *restrict barrier, const pthread_barrierattr_t *restrict attr, unsigned count)	smh_barrier_create(smh_barrier_t *barrier, uint32_t nthreads, smh_status_t *status)
Barrier synchronize	pthread_barrier_wait(pthread_barrier_t *barrier)	smh_barrier_wait(smh_barrier_t *barrier, smh_status_t *status)
Initialize mutex	pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr)	smh_mutex_create(smh_mutex_t *mutex, smh_status_t *status)
Lock mutex	pthread_mutex_lock pthread_mutex_t *mutex)	smh_mutex_lock(smh_mutex_t *mutex smh_status_t *status)
Unlock mutex	pthread_mutex_unlock(pthread_mutex_t *mutex)	smh_mutex_unlock(smh_mutex_t *mutex smh_status_t *status)
Initialize condition	pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr)	smh_cond_create(smh_condvar_t *cond, smh_status_t *status)
Condition wait	pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex)	smh_cond_wait(smh_condvar_t *cond, smh_mutex_t *mutex, smh_status_t *status)
Condition signal	pthread_cond_signal(pthread_cond_t *cond)	smh_cond_signal(smh_condvar_t *cond, smh_status_t *status)
Condition Broadcast	pthread_cond_broadcast(pthread_cond_t *cond)	smh_cond_broadcast(smh_condvar_t *cond smh_status_t *status)

Chapter 6

Performance Evaluation

In this chapter we describe performance studies that demonstrate that Samhita and RegC provide a programmable, scalable, and efficient shared memory programming model. For the cluster implementation we present scalability results on up to 256 cores, which to our knowledge is the largest scale test by a significant margin for any DSM system reported to date. For the heterogeneous system implementation we present scalability results on up to 120 cores. To the best of our knowledge we are the first thread based virtual shared memory system on such systems. The results demonstrate that for scalable algorithms, our Samhita implementation achieves good weak scaling on large core counts. Strong scaling results for Samhita are very similar to equivalent Pthreads implementations on a single node. For less scalable algorithms, scalability is limited by synchronization overhead. We identify extensions to the programming model that transparently leverage information about data placement and consistency requirements to improve performance.

The performance evaluation was carried out on the following system configurations:

SystemG: A 2600 core (325 nodes) cluster supercomputer. Each node contains two quad-core 2.8GHz Intel Xeon (Penryn Harpertown) processors with 8GB of main memory. The cluster is interconnected using quad data rate (QDR) InfiniBand switched fabric.

Ithaca: A 672 core (84 node) cluster supercomputer. Each node contains two quad-core 2.26GHz Intel Xeon (Nehalem Gainestown) processors with 24GB of main memory. The cluster is interconnected using QDR InfiniBand switched fabric.

Heterogeneous system: The work station contains two octa-core 2.60GHz (Sandy Bridge EP) processors with 64GB of main memory. The node also contains two Intel Xeon Phi 5110P (Knights Corner) coprocessors, pre-production version.

The performance evaluation includes strong scaling results where problem size is fixed as the number of cores grows, and weak scaling experiments where problem size grows with the number of cores. We describe the benchmarks used in our evaluation in Section 6.1. Section 6.2 presents the results of our evaluation using these benchmarks for our cluster implementation. The results of our evaluation for the heterogeneous system is presented in Section 6.3.

6.1 Benchmarks

In our evaluation we use two synthetic micro-benchmarks, one synthetic memory bandwidth benchmark and four application kernels. The first micro-benchmark measures the achievable bandwidth in terms of read and write operations. The second micro-benchmark measures the impact of false sharing and ordinary region size on important factors of an application runtime – computation time and synchronization time. We use the STREAM TRIAD [61] as the synthetic memory bandwidth benchmark. We use LU factorization, a Black-Scholes solver, a Jacobi sweep and a molecular dynamics application as the application kernel benchmarks. The API provided by Samhita is very similar to that of Pthreads [20]. In fact, all our benchmarks share the same code base, with memory allocation, synchronization and thread creation expressed as macros. These macros are processed using the `m4` [11] macro processor. This illustrates how existing shared memory code can run using Samhita/RegC with trivial code modification.

6.1.1 Memory bandwidth micro-benchmark

Our synthetic memory micro-benchmarks measure the memory bandwidth that is achievable by an application in terms of read and write operations. We use the `memcpy` [13] operation to measure the achievable read bandwidth by copying data from a memory region allocated in the Samhita address space to a memory region allocated locally on a computer server of the same size.

```
for (i = 0; i < N; ++i) {
    sum = 0;
    for (j = 0; j < M; ++j) {
        for (k = 0; k < S; ++k) {
            rsum = 0;
            for (l = 0; l < B; ++l) {
                *am(k,l) = r * (*am(k,l));
                rsum += *am(k,l);
            }
            sum += M_PI * rsum;
        }
    }
    LOCK(lock);
    gsum += sum;
    UNLOCK(lock);
    BARRIER_WAIT(barrier);
}
```

Figure 6.1: Computational kernel of the micro-benchmark code used in the experiments, reflecting local allocation. For global allocation, array indices are a function of thread id. For global strided allocation, the k loop take strides of num_threads.

To measure the achievable write bandwidth we use the `memset` [14] operation on a memory region allocated in the Samhita address space. For both the `memcpy` and `memset` memory bandwidth test we use an allocation size of 1GB and we measure the bandwidth by varying the cache size associated with each Samhita compute thread. The cache sizes used in the performance study are 256MB, 512MB, 768MB, 1GB and 2GB. For the heterogeneous evaluation we run two sets of experiments to measure the bandwidth. First, we run the memory server on `cpu0` on the host and the benchmarks on `mic0`. Second, we run the benchmark on `mic1` with the memory server continuing to run on `cpu0`.

6.1.2 Micro-benchmark

The micro-benchmark (see Figure 6.1) allocates a fixed amount of data (S rows of doubles, each of length B) per compute thread. An inner compute loop executes M times and does two floating point operations per data element per iteration. At the end of this inner loop we update a global sum protected by a mutex variable, which is followed by a barrier synchronization. In this way, the

amount of data per thread can be varied via S and B; and the amount of computation per data element (relative to the frequency of synchronization operations) can be varied using M. An outer iteration repeats the computation N times. We use $N = 10,000$ and $B = 256$ for all experiments reported in this chapter.

Since data layout and the potential for false sharing are important to the performance of any cache-based shared memory system, the micro-benchmark also allows us to vary the memory allocation and work distribution strategy. The allocation is performed either *locally* or *globally*. Local allocation means that each thread allocates the memory that will hold its data. Note that this memory is still drawn from the global address space, is served by the memory servers, etc. The Samhita memory allocator ensures that there will be no false sharing among compute threads who do their own local allocation in this way. Global allocation means that only one thread does a single large shared allocation, with each compute thread then working on its own share of that data. Hence, global allocation has a greater risk of false sharing (within a page or within a cache line) among compute threads than local allocation. There are two variations of work distribution and data access for the global allocation case. In the first variation, during the inner compute loop all the data that is accessed in each iteration is contiguous, i.e., S rows of length B, all stored contiguously. We refer to this variation as simply "global" allocation. In the second access pattern variation, the data in each iteration of the inner loop data is stored contiguously for a block of length B, but the next block accessed is strided based on the processor id. In other words, each compute thread accesses S rows of length B, but the rows assigned to the threads are interleaved. These two global allocation variations correspond to block or round-robin allocation of rows of a matrix, for example. The potential for false sharing in the "global strided" case is the highest of the three approaches. In our performance evaluation we evaluate how false sharing amongst threads affects compute time and synchronization time We also evaluate how the amount of data accessed in the ordinary region affects both compute and synchronization time.

6.1.3 STREAM TRIAD

The STREAM benchmark [61] is a synthetic benchmark that measures sustained memory bandwidth for a set of simple vector kernels. We implemented a thread based version of the TRIAD operation. This operation is a simple vector update (or DAXPY), a level 1 operation from the BLAS [56] package. The TRIAD kernel computes $A = B + \alpha C$, where A, B and C are vectors of dimension n and α is a scalar. Each run of the benchmark consists of 400 iterations of the TRIAD operation with a barrier between each iteration.

6.1.4 LU factorization

A standard HPC benchmark is the LU factorization of a dense $n \times n$ matrix. Using the Samhita API, we implemented a simple shared-memory parallel version of the right-looking Level 3 BLAS [31] version of the LU algorithm, patterned after the LAPACK routine `dgetrf` [21]. Our algorithm is parallelized over block-columns of the matrix, with block size set to 128. We measured performance for various problem sizes and numbers of threads. For comparison purposes, we also measured the performance of the equivalent MPI based ScaLAPACK routine, namely `pdgetrf` [24], run with a $1 \times p$ processor topology, where p is the number of processor cores used.

6.1.5 Black-Scholes model

The Black-Scholes application is from the PARSEC benchmark suite [23]. This application calculates the theoretical estimate of the prices for a portfolio of European style options using the Black-Scholes partial differential equation (PDE). We used the large input data set from the PARSEC benchmark suite which has 10 million entries.

6.1.6 Jacobi sweep

The Jacobi algorithm application kernel is based on the code from the OmpSCR [32] repository. It corresponds to a simple Jacobi sweep as commonly used in multigrid solvers, for example. This kernel corresponds to the Jacobi iteration for solving the linear system corresponding to a discrete laplacian. The memory access pattern for this kernel is representative of many computations with a

nearest neighbor communication pattern, i.e., the update at a given grid point depends on previous values at some small number of near neighbors. We use two implementation of the Jacobi application kernel. In the first Pthreads and Samhita implementation we use a mutex variable to protect a global variable and require three barrier synchronization operations in each outer iteration. The second implementation replaces work done done using the mutex locks with a reduction operation implemented by the Samhita run-time system. In our cluster evaluation we use both the lock-based and reduction operation based implementation. For the heterogeneous system evaluation we use the reduction operation based implementation as we are yet to port our store instrumentation technique to support the Intel Xeon Phi's K1OM ABI [59].

6.1.7 Molecular dynamics application

The molecular dynamics application benchmark is a simple n-body simulation using the velocity Verlet time integration method. The particles interact with a central pair potential. The molecular dynamics application is based on the code from the OmpSCR repository. Our threaded implementations use a mutex variable to protect the global kinetic and potential energy variables. Barrier synchronization is used during various stages of the computation for synchronization. We also evaluate using a second implementation that uses the reduction operation provided by Samhita. We use both the lock-based and reduction operation based implementation for evaluation in our cluster implementation. For the heterogeneous system evaluation we only use the reduction operation based implementation.

6.2 Cluster implementation performance evaluation

In this section we present our evaluation of the cluster implementation of Samhita. We present some of the results from our evaluation on Ithaca and SystemG. We chose these systems to show the performance differences of the cc-NUMA architecture in the Nehalem processors and the shared bus design of the Penryn class processor. MPI based parallel applications on Ithaca use OpenMPI-1.3.3. Unless noted we use one memory server and manager for all the experimental evaluation. All the

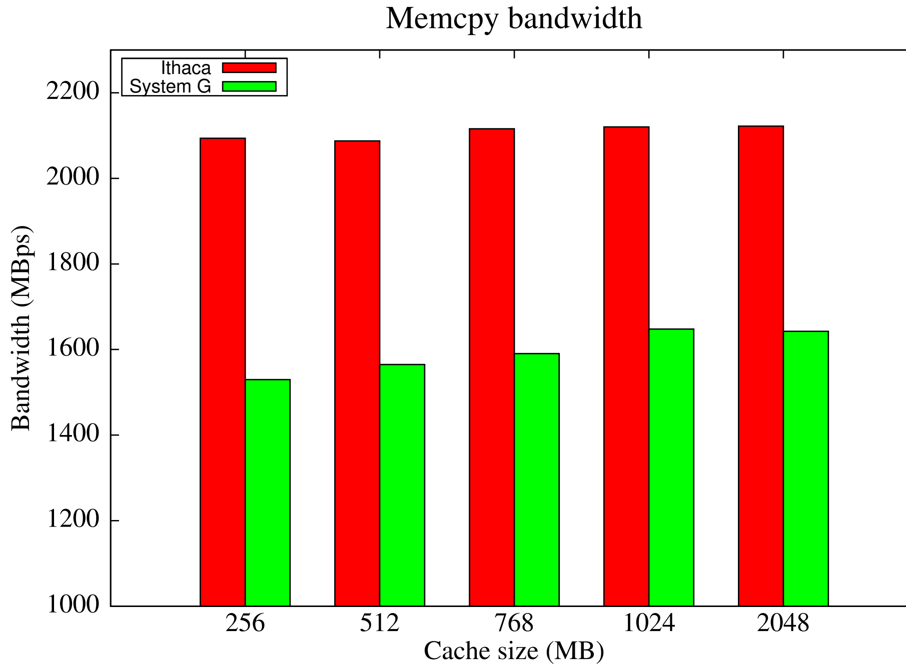


Figure 6.2: Memcpy bandwidth achieved for varying cache size for Ithaca and SystemG.

results presented are from runs on SystemG unless otherwise stated. The data accessed by each thread fits in its local cache unless otherwise noted.

6.2.1 Memory bandwidth micro-benchmarks

The performance results in Figure 6.2 show that Samhita achieves nearly 2.2 GB/s on Ithaca when reading from remote memory and writing to local memory. This compares well with the memcpy bandwidth within a single Nehalem node of ∼3.9 GB/sec. This performance is also close to the achievable maximum of 2.6 GB/sec on quad data rate InfiniBand on this platform. The performance of memcpy on SystemG is bound by the memory bandwidth of the shared bus design in the Penryn processor. Internal timing results show that pages are brought in from the remote memory server at nearly 2.6 GB/sec, but the lower memory bandwidth on the processor increases the time taken to copy data from the global address space to the locally allocated memory. Since both Intel and AMD have moved to cc-NUMA architectures, we expect Samhita's performance to track the results on Ithaca and be bound by the interconnect's bandwidth.

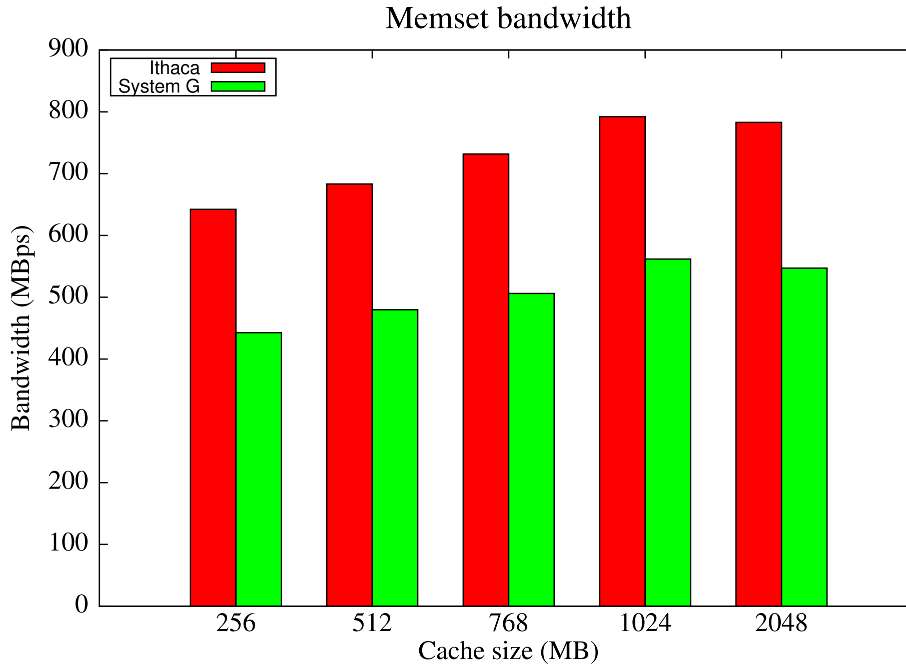


Figure 6.3: Memset bandwidth achieved for varying cache size for Ithaca and SystemG.

The performance results in Figure 6.3 show the overhead of transitioning a page from *read only* mode to *read-write* mode, creating the twin page and calculating and transmitting the XOR difference. Since the size of the memory in the global address space (1 GB) that is being memset is larger than the cache (except for the cache size of 2048MB), the cost of memset includes the cost of handling capacity misses and associated page evictions. Even in this scenario Samhita achieves a bandwidth of 790 MB/sec, which is nearly the same as the achievable bandwidth on 10 Gigabit Ethernet [36].

6.2.2 Micro-benchmark

In Figures 6.4, 6.5 and 6.6 we compare normalized compute time per thread between Pthreads (up to 8 threads) and Samhita (up to 32 threads). We normalize the runtime with the equivalent 1-thread compute time for the Pthreads implementation. This experiment compares the compute time between Pthreads and Samhita and how the compute time of Samhita varies with respect to Pthreads depending on the amount of computation performed in the inner loop, and how it is

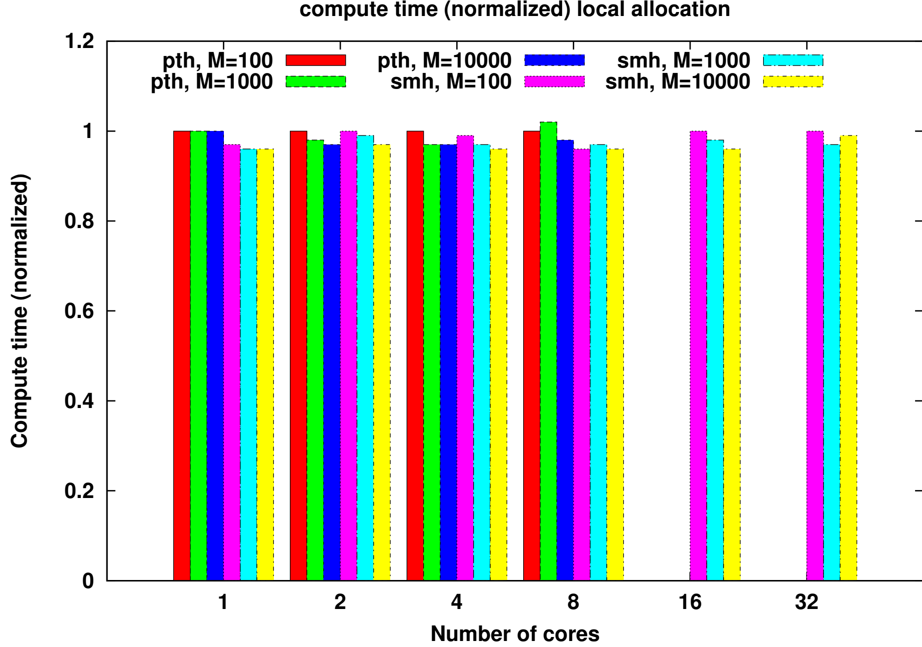


Figure 6.4: Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated locally.

affected by false sharing.

Figure 6.4 compares the normalized compute time for Pthreads and Samhita as we vary both the number of compute threads and the amount of computation performed in the inner loop, with the data allocated locally. The figure shows that the normalized compute time for Pthreads and Samhita are very similar. In the absence of false sharing the time spent in computation for Samhita is very similar to the equivalent Pthread implementation, even for a relatively small amount of computation (small M).

Figure 6.5 compares the normalized compute time when the data is allocated globally. The figure shows that when the amount of compute performed is low the added penalty incurred by Samhita due to false sharing and other overheads is noticeable. However, as we increase the amount of compute this cost is amortized and the amount of time spent in computation by Samhita is very comparable to Pthreads. This underlines the fact that even if there is some false sharing the penalty can be amortized by the amount of computation performed by the application.

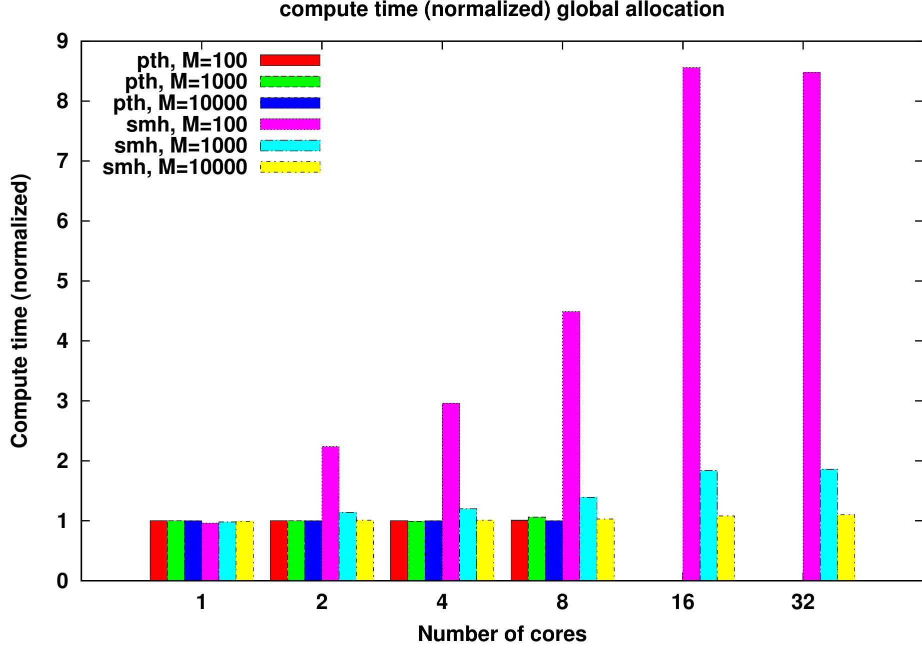


Figure 6.5: Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally.

Figure 6.6 compares the normalized compute time when the data is allocated globally and the access pattern is strided, which increases the amount of false sharing among threads. We see that when the amount of computation performed is relatively small there is a higher penalty compared to the global allocation case. However, once again this cost can be amortized by increasing the amount of compute performed over the data that is shared among the compute threads.

Figures 6.7, 6.8 and 6.9 compare the compute time per thread for the Samhita implementation as we vary the amount of data accessed in the ordinary region versus the number of computation threads. Figure 6.7 clearly shows how the computation time increases with the amount of work and amount of data accessed in the ordinary region, as expected. However, compute time per thread does not increase as the number of threads increases. This once again underlines the fact that when there is no false sharing the Samhita implementation does not incur any additional penalty.

Figure 6.8 shows the same trend as local allocation, with the amount of time spent in computation increasing as the amount of data accessed in the ordinary region increases. Due to modest false

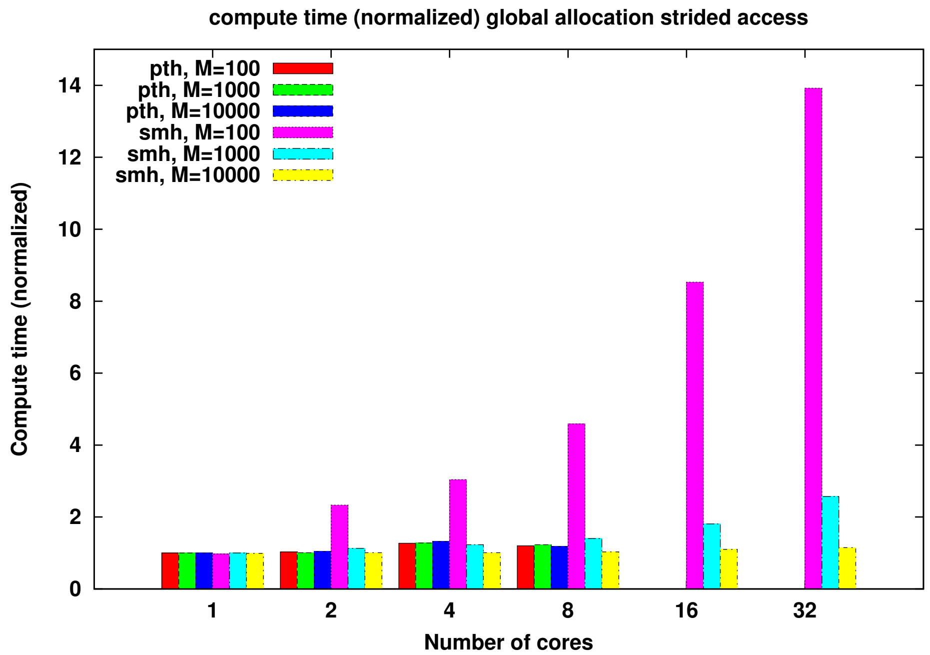


Figure 6.6: Normalized compute time vs. number of cores. The compute time is normalized with the equivalent 1-thread compute time for Pthreads. The number of inner iterations M is varied, $M = \{100, 1000, 10000\}$. The memory for each thread is allocated globally, but access using strides.

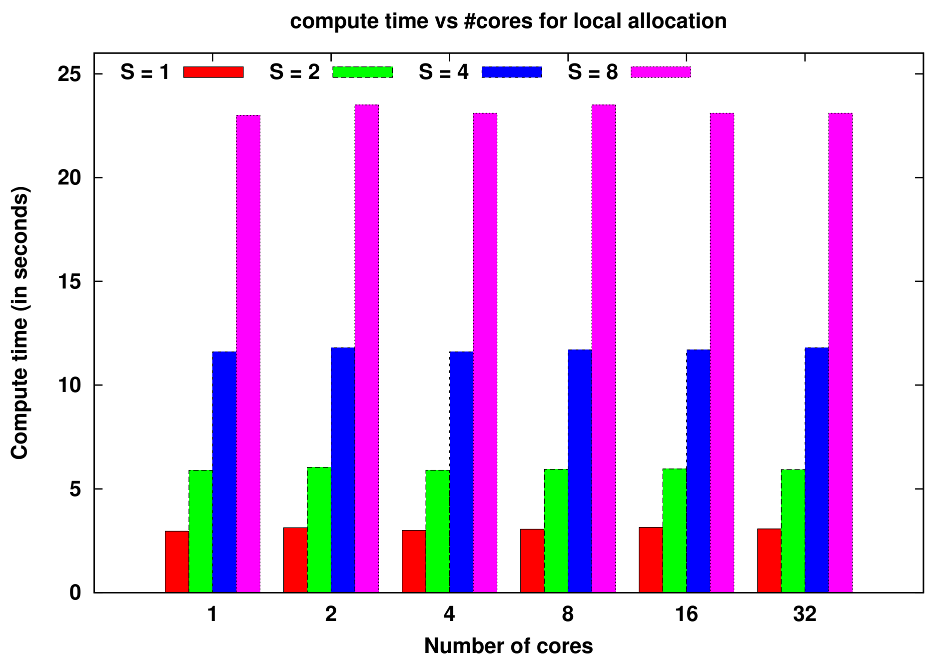


Figure 6.7: Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated locally.

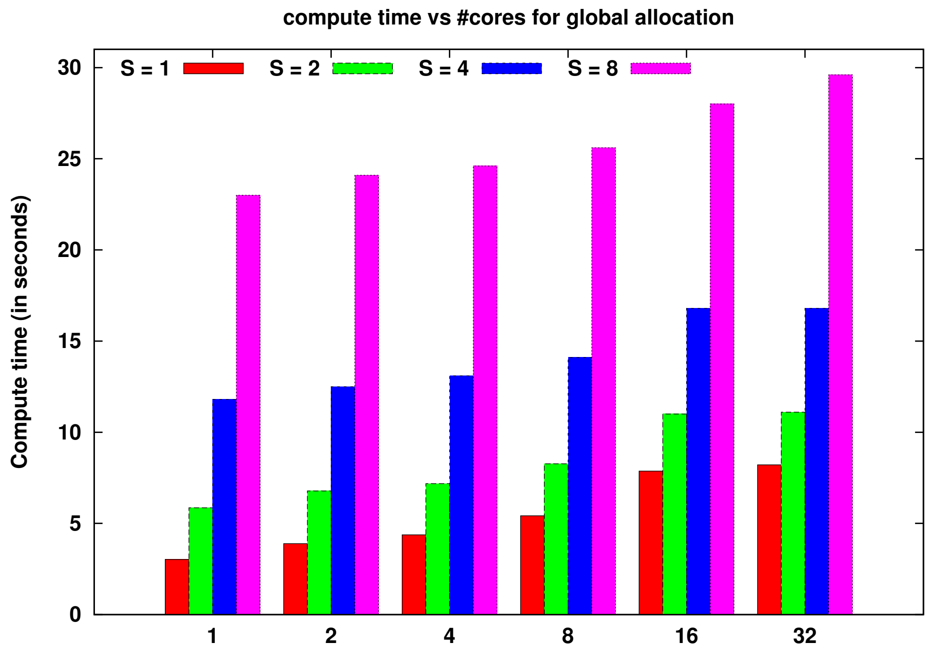


Figure 6.8: Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally.

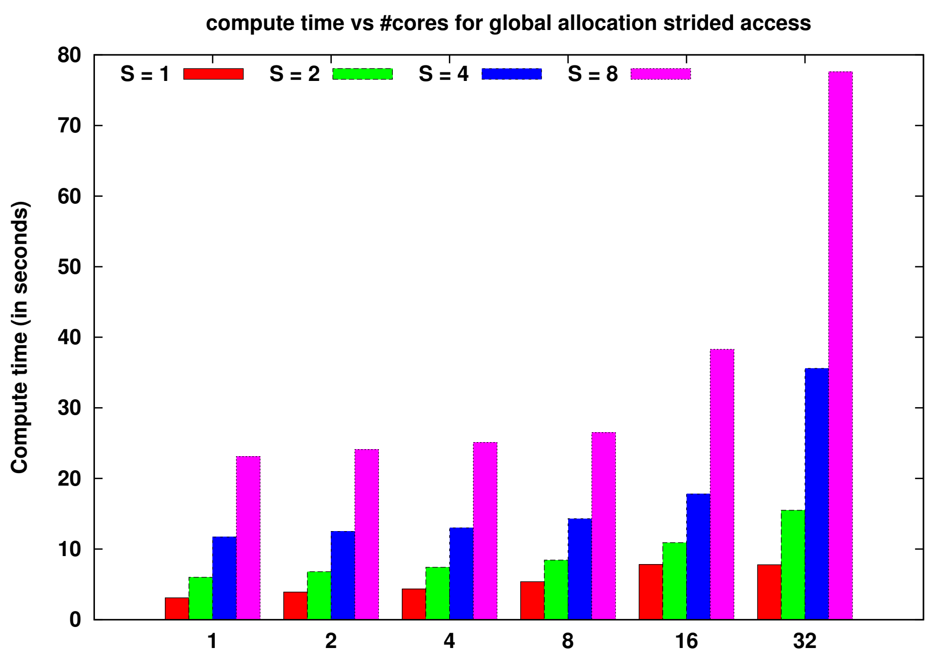


Figure 6.9: Compute time vs. number of cores. The number of rows of doubles allocated is varied, $S = \{1, 2, 4, 8\}$, for a fixed $M = 1000$. The memory for each thread is allocated globally, but accesses using strides.

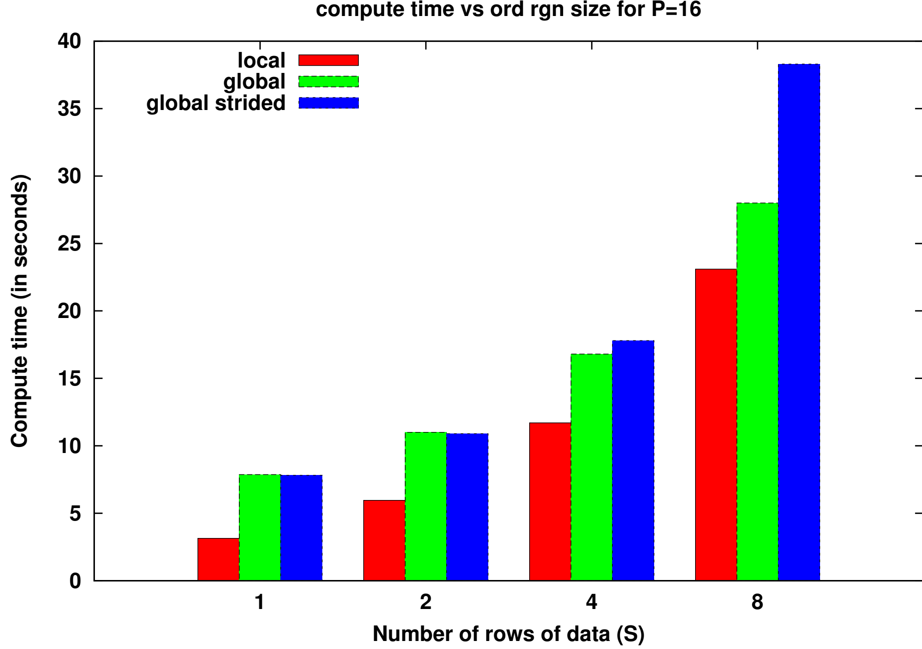


Figure 6.10: Compute time vs. number of rows of doubles allocated. Compute time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$.

sharing, the compute time per thread does grow slowly as the number of compute threads increases. However, comparing Figure 6.7 and 6.8, we see that the penalty is not significant.

Figure 6.9 shows a similar trend for global strided allocation. However, due to the access pattern which increases false sharing, we see that there is a higher penalty incurred in the compute time. This penalty increases as the amount of data increases, which results in higher data false sharing.

Figure 6.10 compares the compute time for the Samhita implementation for 16 compute threads, with respect to the number of blocks S assigned to each compute thread. When the number of blocks is one there is no difference in the access pattern between global and global strided allocations. We see that as the size of the ordinary region grows, the compute time increases as expected, and the penalty incurred in compute time increases based on the amount of false sharing.

Figure 6.11 compares the synchronization time for Samhita for the 16-thread run, again varying S, the number of blocks per thread, run on SystemG. The evaluation shows that when there is no false sharing (local allocation) the increase in synchronization cost is hardly noticeable. False

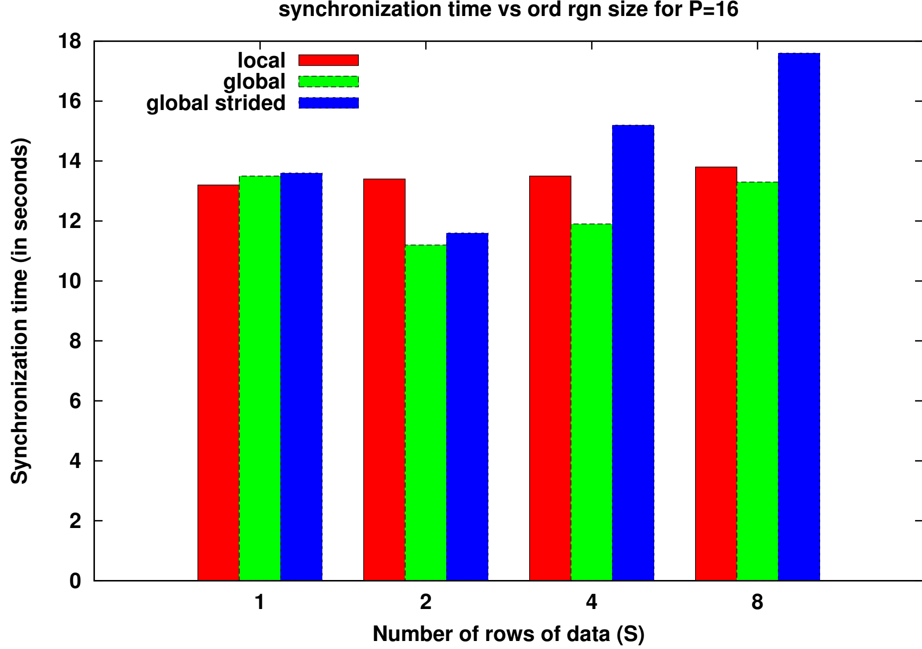


Figure 6.11: Synchronization time vs. number of rows of doubles allocated. Synchronization time for local, global allocation and global allocation with strided access are compared for $S = \{1, 2, 4, 8\}$ for $M = 1000$, $P = 16$ and $B = 256$.

sharing does have an impact on synchronization since during synchronization there is increased data movement. However, Samhita's synchronization operations move only the minimum amount of data required, so that even with increased false sharing the increase in synchronization cost is not dramatic.

Figure 6.12 compares the synchronization time between the Samhita and Pthread implementation, varying the number of threads for $M = 100$ iterations of the inner loop. The figure shows that Samhita does incur an increased cost for synchronization. However, this is expected as the synchronization operations in Samhita perform memory consistency operations, which are expensive, unlike Pthreads which performs only synchronization. The graph also shows that Samhita's synchronization overhead is not exceptionally high when compared to Pthreads, and the increase with the number of threads is not dramatic.

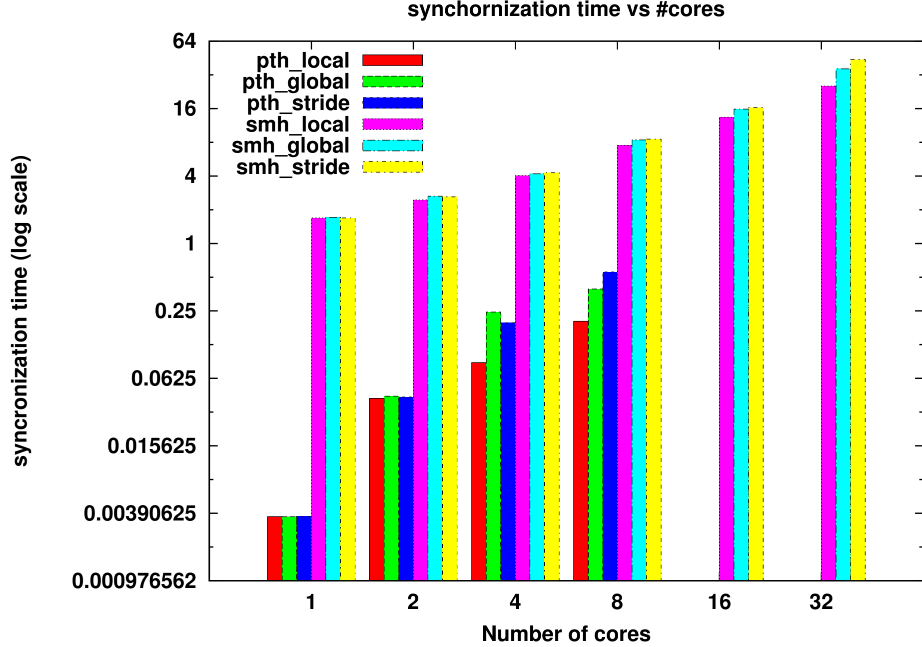


Figure 6.12: Synchronization time (log scale) vs. number of cores. Synchronization time for Pthreads and Samhita are compared for local, global allocation and global allocation with strided access. $M = 100$, $B = 256$ and $S = 2$ are kept fixed.

6.2.3 STREAM TRIAD

We present results from three different implementations for Samhita. *samhita_page* refers to the implementation of Samhita that uses page granularity for consistency region updates. *samhita_full* refers to the implementation that uses fine grain object granularity for consistency region updates. However, all the store operations are instrumented and the decision to call the instrumentation function for consistency region stores is based on a runtime check. *samhita* refers to the implementation that uses fine grain object granularity for consistency region updates. In this implementation we use static analysis to instrument only consistency region stores.

Figure 6.13 compares the strong scaling bandwidth achieved by the Pthreads and Samhita implementations. All three Samhita implementations achieve a reasonable sustained bandwidth, which scales as we increase the number of cores. The bandwidth achieved by the *samhita_page* implementation is close to 75% of that achieved by the Pthreads implementation for the 8 core run. The bandwidth achieved by the *samhita_full* implementation is close to 85% of the Pthreads im-

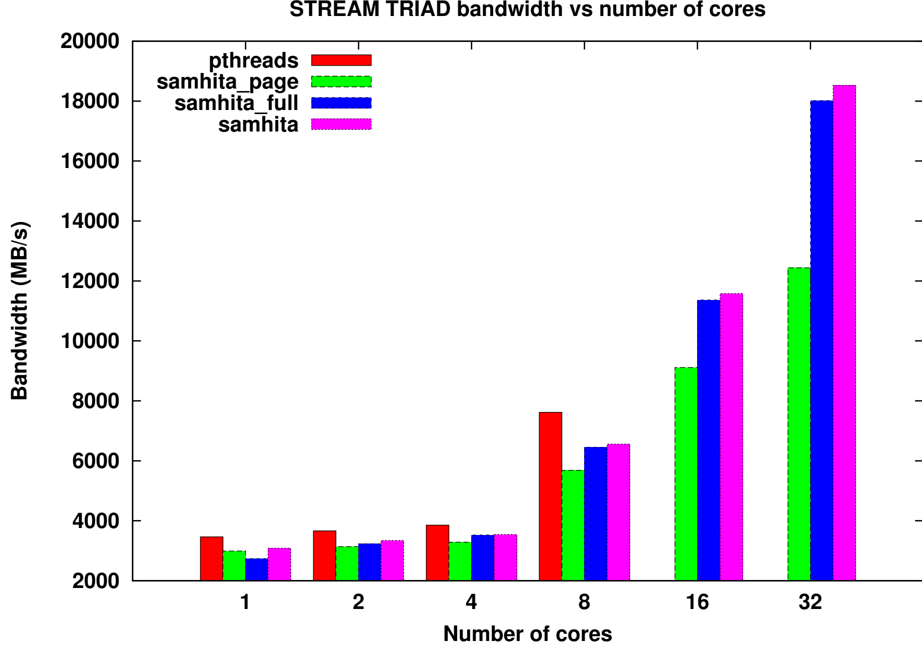


Figure 6.13: Sustained memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Vectors of dimension $n = 16M$.

plementation, while *samhita* implementation achieves close to 86% of the Pthread implementation. The improved performance of the implementation of Samhita that use fine grain object granularity updates for consistency region stores is due to the decreased complexity of the memory consistency operations during synchronization. This underlines the importance of using fine grain object level updates for consistency region stores. We also note that the bandwidth achieved for 1–4 cores is similar due to the fact that our physical nodes are dual socket and our placement policy fills the first socket before filling the second socket. We see that the *samhita* implementation performs consistently better than *samhita_full* implementation. This is because we have completely avoided instrumenting any store in the STREAM TRIAD application and do not pay the penalty of the runtime check paid by *samhita_full* implementation.

Figure 6.14 presents weak scaling results for up to 256 processors. We use 20 memory servers and one manager for the weak scaling evaluation. The performance of Samhita implementations tracks Pthreads up to 8 cores and continues to scale well up to 256 cores, before synchronization costs begin to constrain scalability. We once again notice that the *samhita* implementation performs

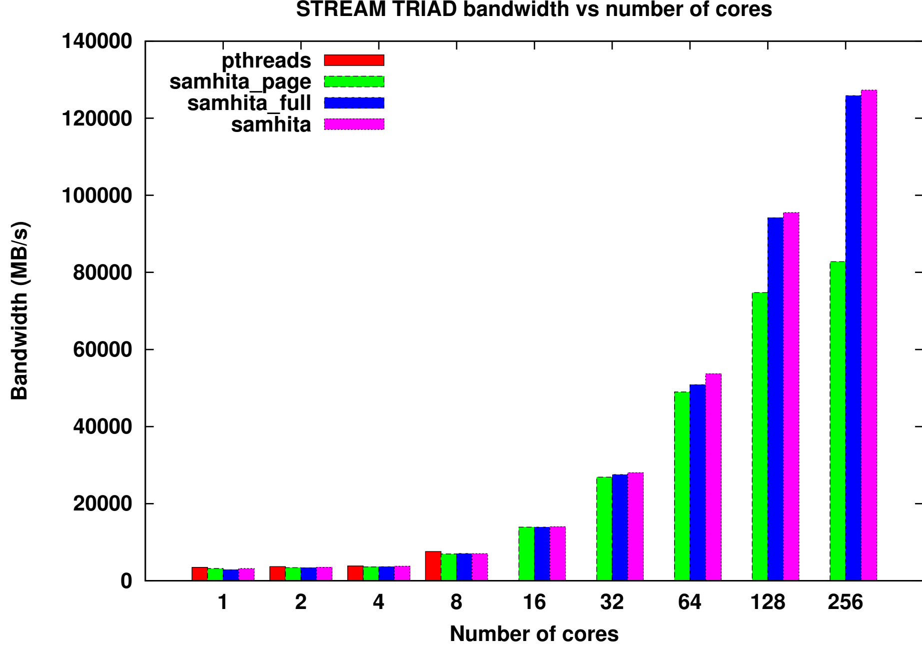


Figure 6.14: Sustained memory bandwidth vs. number of cores for Samhita and Pthreads based STREAM TRIAD benchmark. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for $p = 1$ is $n = 16M$; problem size for $p = 256$ is $n = 4G$.

consistently better than *samhita_full* because it does not have the overhead of the runtime check for every store performed.

In the weak scaling results shown in Figure 6.14 the data associated with each process fits entirely in the Samhita cache associated with that process. Figure 6.15 shows the same results for Samhita, along with results for a problem size twice as big. The larger problem no longer fits in the local Samhita cache, which results in capacity misses; the entire data must be streamed in for each iteration. We see that when the resulting data spills occur there is a clear impact on the achieved bandwidth. However, we also notice that the Samhita implementation still continues to scale well; we lose at most a factor of two despite having to refill the cache on each iteration with data served from remote memory servers. This illustrates the benefit of our optimization for fetching remote pages and the benefits of the simple prefetching strategy used in our current implementation.

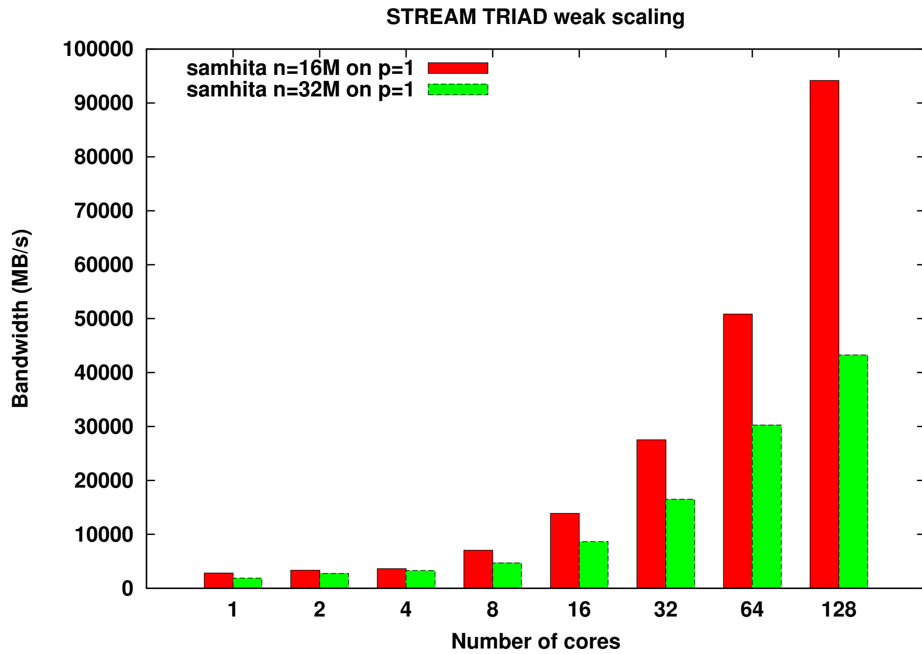


Figure 6.15: Sustained memory bandwidth vs. number of cores for Samhita for two different problem sizes. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for small problem on $p = 1$ node is $n = 16M$; problem size for large problem on $p = 1$ node is $n = 32M$.

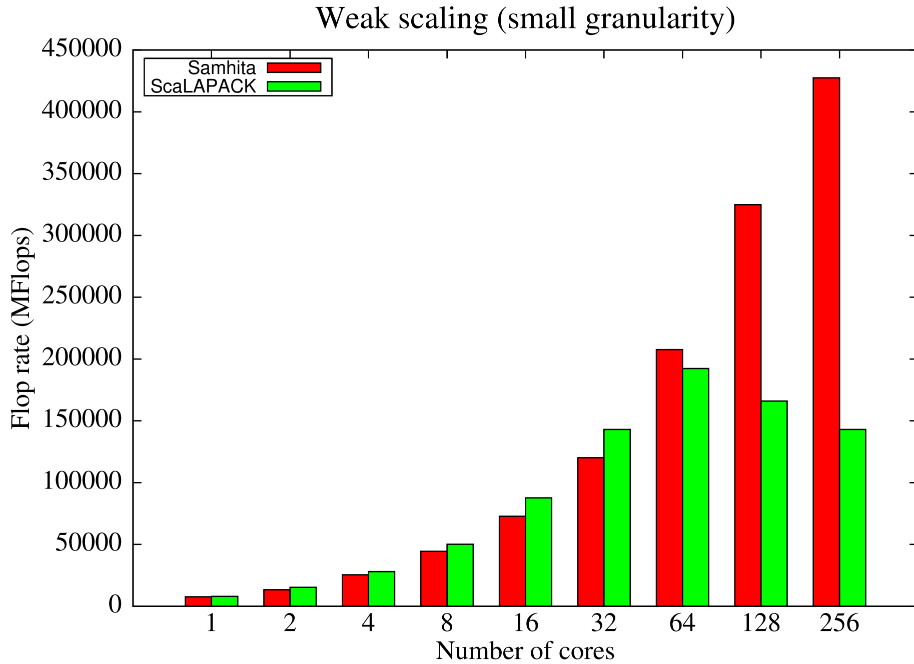


Figure 6.16: Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 4000$; problem size for $p = 256$ is $n = 64000$.

6.2.4 LU factorization

Figure 6.16, 6.17 and 6.18 show weak scaling results for small, medium and large problem sizes, run on Ithaca. Performance on SystemG is similar. By 'weak scaling' we mean that the data set size n^2 scales with the number of cores. For all experiments on Ithaca we assigned 8 processes per compute node, and used 16 memory servers, with each memory server serving up to 15GB of memory. We see that for modest core count, LU over Samhita trails ScaLAPACK by only a few percent. At 64 threads, Samhita is slightly faster than ScaLAPACK, and then the advantage grows as ScaLAPACK incurs greater and greater communication costs for large p. Note that this is not unexpected since this version of the parallel LU algorithm (parallelized over block columns) is known to eventually hit a scalability wall due to the fact that the number of block columns per core shrinks as p grows. What is encouraging however, is that the Samhita version is able to scale reasonably well up to 256 cores.

Strong scaling results from Ithaca are given in Figure 6.19, where the number of threads grows

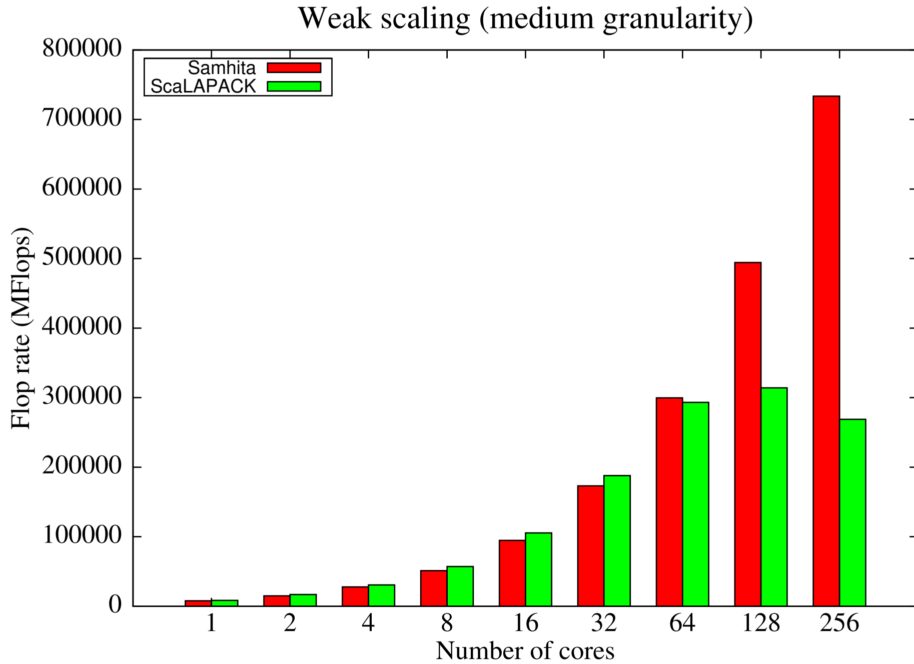


Figure 6.17: Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 8000$; problem size for $p = 256$ is $n = 128000$.

but problem size is fixed at $n = 30000$. Again we see that the Samhita implementation tracks very well the performance of the MPI based ScaLAPACK implementation, and surpasses it at $p = 64$.

Our final LU example illustrates the case of a single threaded application that requires a large address space. For this experiment we allocated 1GB for the local Samhita cache. The data for the smallest problem shown in Figure 6.20 ($n = 10000$) fits in the cache, but all other problems do not. There is a performance penalty of less than a factor of 2 when the cache size boundary is crossed, but beyond that the performance degrades very slowly as the problem size grows. Since the LU algorithm makes repeated passes through essentially the entire data set, the cost of capacity evictions in this case is high. In contrast, the only alternative today for single threaded applications needing a large address space is to swap to disk, which incurs a performance penalty of over five orders of magnitude.

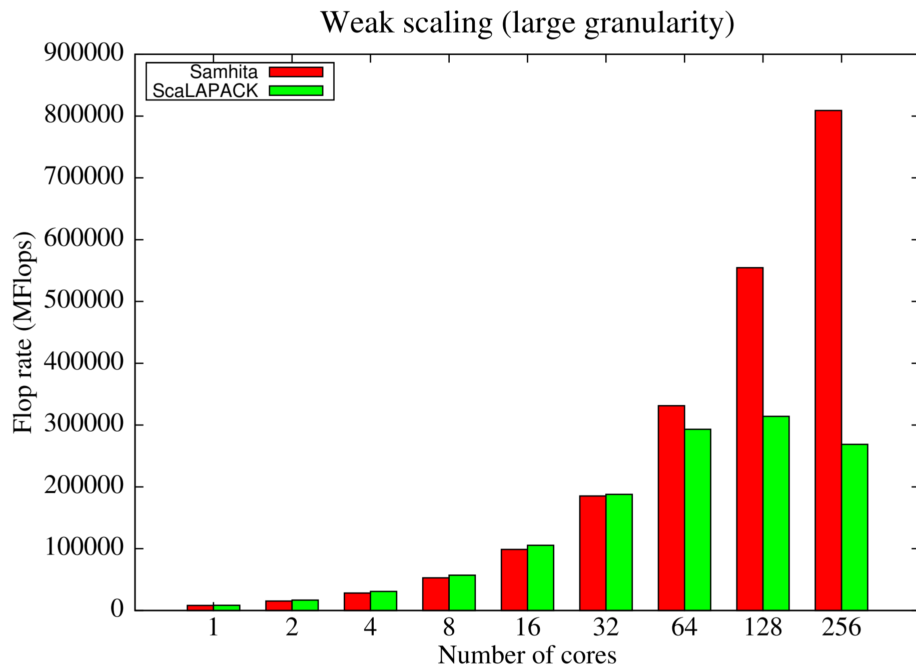


Figure 6.18: Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Data size n^2 and number of threads p are scaled proportionally, i.e., n^2/p is a constant. Problem size for $p = 1$ is $n = 10000$; problem size for $p = 256$ is $n = 160000$.

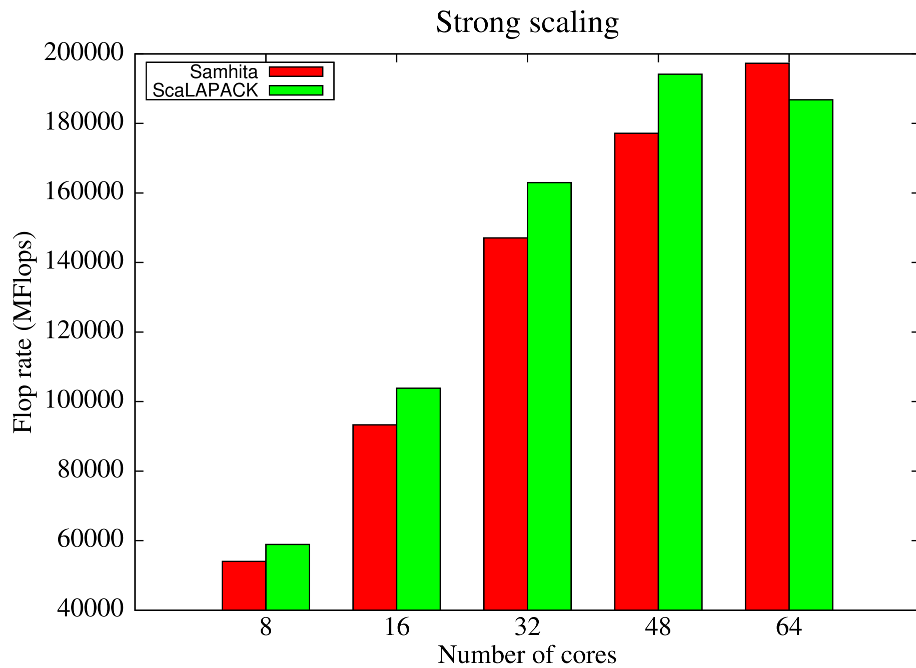


Figure 6.19: Computational rate vs. number of threads for Samhita based LU and ScaLAPACK. Fixed problem size $n = 30000$.

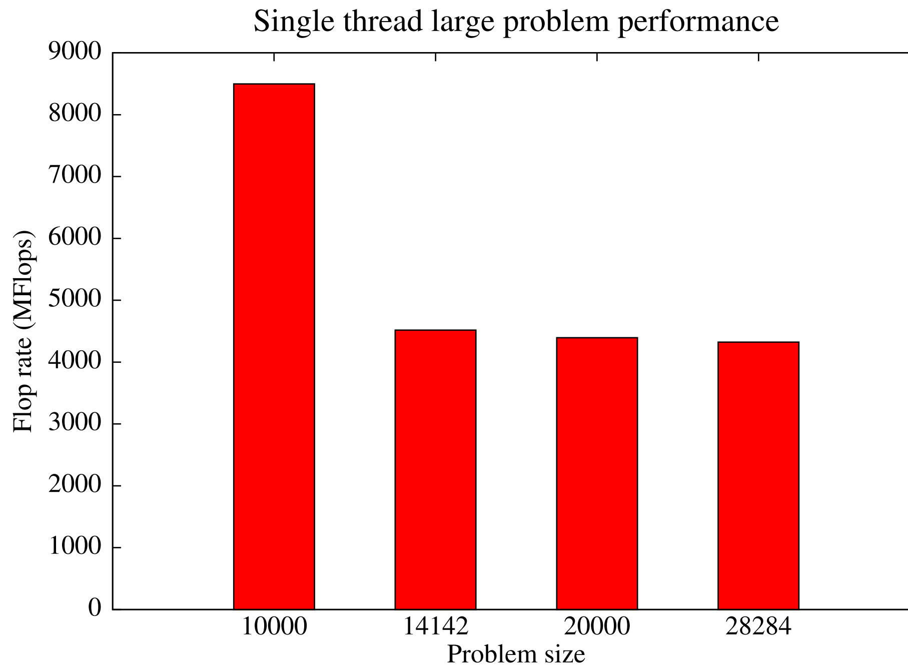


Figure 6.20: Computational rate vs. problem size for single threaded Samhita based LU with cache size of 1GB.

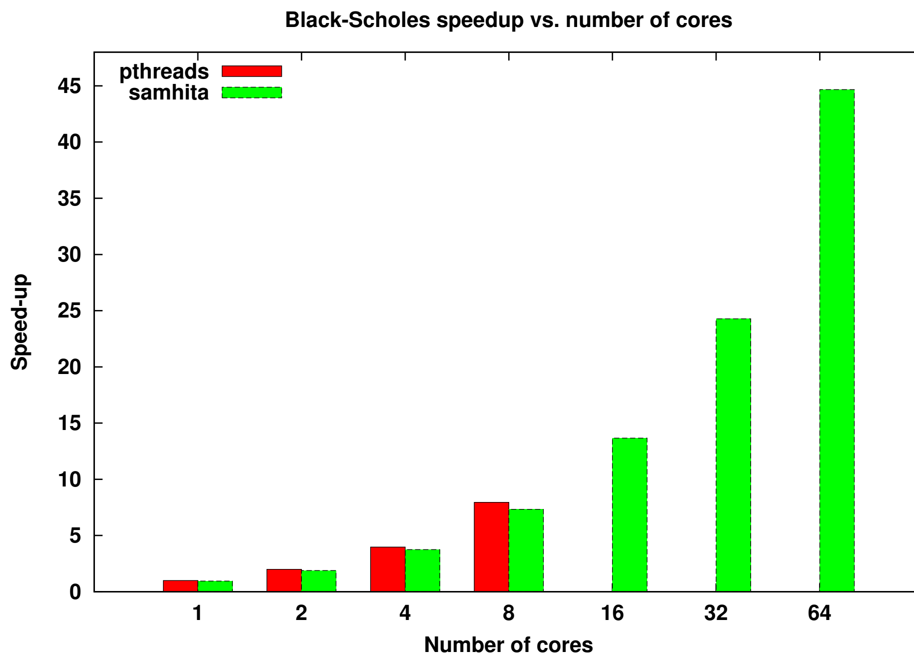


Figure 6.21: Parallel speedup vs. number of cores for PARSEC Black-Scholes application.

6.2.5 Black-Scholes

The benchmark is a scalable master-worker algorithm, and Figure 6.21 shows excellent strong scaling on up to 64 cores on SystemG. We notice that the Samhita implementation tracks the Pthreads implementation very closely within a node and continues to scale across multiple nodes. The total data allocated by this benchmark does not fit in local cache. However, the data accessed during the computational phase of the application fits in the local cache of each thread.

6.2.6 Jacobi sweep

We present results from three different implementations for Samhita. *samhita_page* refers to the implementation of Samhita that uses page granularity for consistency region updates. *samhita_full* refers to the implementation that uses fine grain object granularity for consistency region updates. However, all the store operations are instrumented and the decision to call the instrumentation function for consistency region stores is based on a runtime check. *samhita* refers to the implementation that uses fine grain object granularity for consistency region updates. In this implementation we use static analysis to instrument only consistency region stores.

Figure 6.22 compares the strong scaling speedup of Pthreads and six Samhita implementations of the Jacobi benchmark. The Pthreads implementation and three of the Samhita implementations use a mutex variable to protect the global variable that accumulates the residual error on each iteration. We notice that the lock based implementation for Samhita does not speedup as the number of processors are increased for *samhita_page* implementation. The reason for the performance degradation is due to the strong memory consistency provided by RegC. Performance profiling shows that the majority of time is spent in the barrier that follows the consistency region. Expensive memory consistency operations are required to reflect the memory updates made in the preceding ordinary region. However, the fine grained object level granularity based implementations, *samhita_full* and *samhita*, show strong scaling results up to 16 processors. This improvement in performance can be solely attributed to the fine grain updates used to propagate the changes made in the consistency region.

One approach to improve the performance would be to relax the consistency model. This would

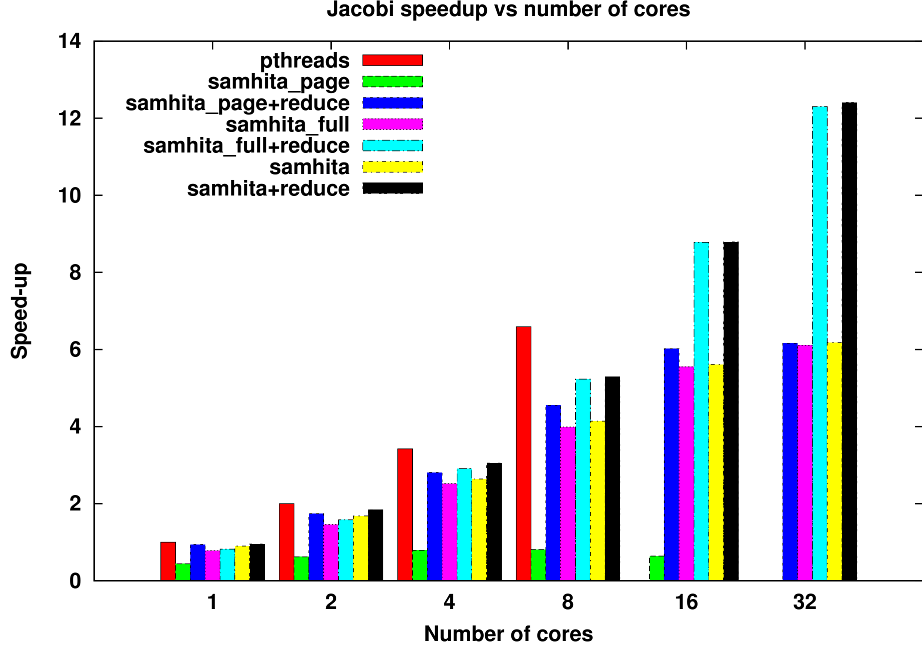


Figure 6.22: Parallel speedup vs. number of cores for Jacobi. Speedup is relative to 1-core Pthreads execution time.

be achieved at the cost of sacrificing programmability. Instead, we extend the programming model by providing a reduction operation (as in OpenMP or MPI) that replaces the operation performed in the consistency region but is implemented efficiently by the Samhita runtime system. The second set of results in Figure 6.22 show the improvement in performance when using the reduction operation extension. We notice that the reduction based Samhita implementation *samhita_page+reduce* achieves just over 69% of the speedup achieved by Pthreads. The reduction based implementation *samhita_full+reduce* achieves just over 79% of the speedup achieved by Pthreads, while *samhita+reduce* achieves just over 80% of the speedup achieved by Pthreads. The improvement we see between *samhita_page* and *samhita_page+reduce* is dramatic. Though there is an improvement between *samhita_full* and *samhita_full+reduce* and *samhita* and *samhita+reduce* it is not as dramatic as in the *samhita_page* case. This once again underlines the importance of having different update mechanism for ordinary and consistency regions.

Figure 6.23 presents weak scaling results for Jacobi on up to 256 processors. The weak scaling experiments use 20 memory servers and one manager. We see that all the Samhita implementa-

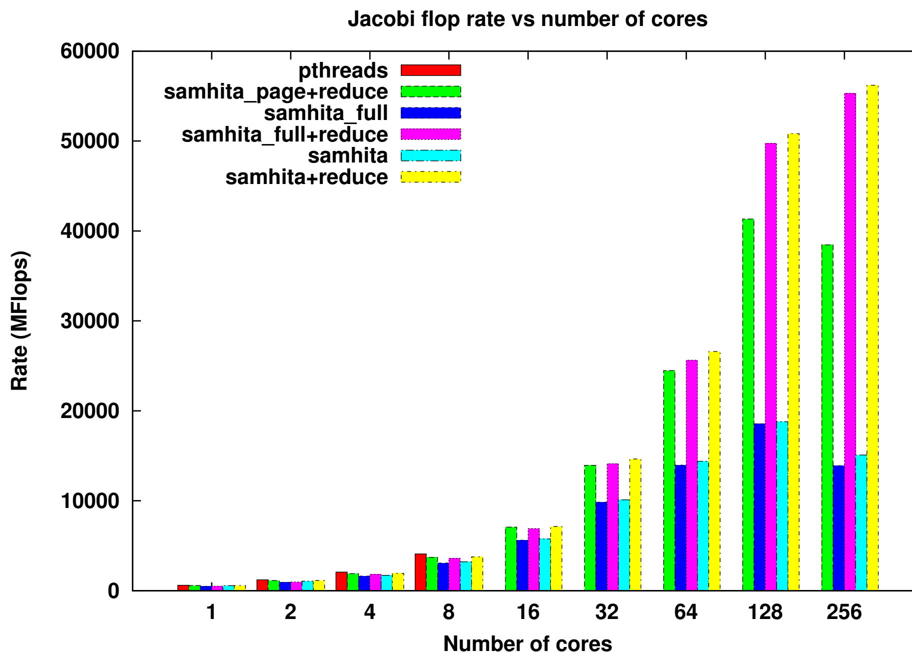


Figure 6.23: Computation rate vs. number of cores for Samhita and Pthreads based Jacobi. Data size $3n^2$ and number of cores p are scaled proportionally, i.e., $3n^2/p$ is a constant. Problem size for $p = 1$ is $n = 4096$; problem size for $p = 256$ is $n = 65536$.

tions of Jacobi track the Pthreads case very well up to 8 cores. The lock based implementations *samhita_full* and *samhita* show reasonable scalability up to 128 threads. The benefit of the reduction extension is again clear, with good scalability for the Samhita implementations up to 128 cores for *samhita_page+reduce* and up to 256 cores for *samhita_full+reduce* and *samhita+reduce*. Beyond 128 cores and 256 cores the scalability of this algorithm is limited, i.e., as problem size and core counts grow, the cost of synchronization eventually outweighs the computation. We notice that fine grain update based implementations of Samhita has reduced complexity in tracking the page state transitions, which yield in the improved performance of the reduction implementation as compared to the *samhita_page* with reduction.

6.2.7 Molecular dynamics

We present results from three different implementations for Samhita. *samhita_page* refers to the implementation of Samhita that uses page granularity for consistency region updates. *samhita_full*

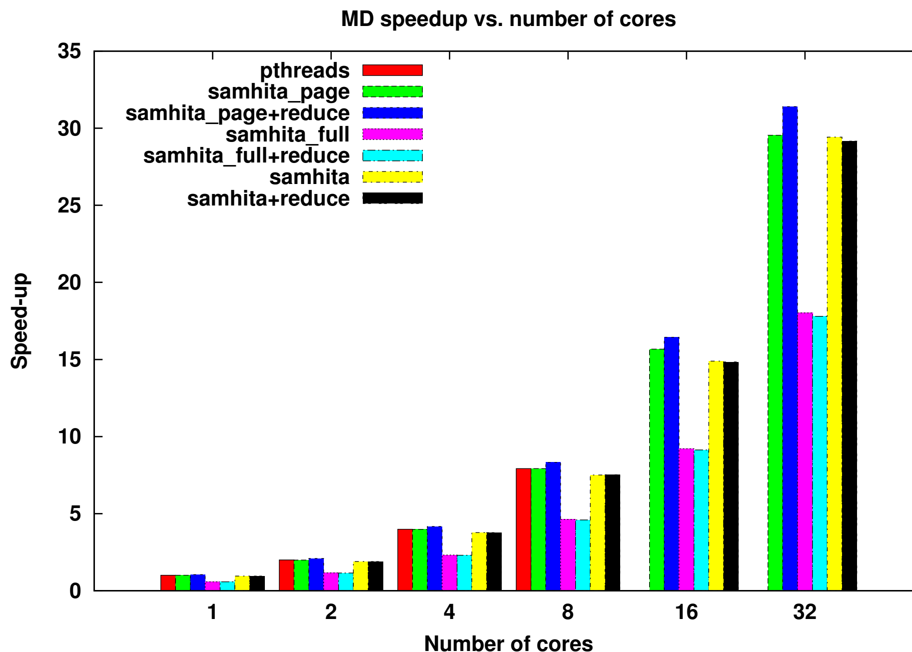


Figure 6.24: Parallel speedup vs. number of cores for molecular dynamics. Speedup is relative to 1-core Pthreads execution time.

refers to the implementation that uses fine grain object granularity for consistency region updates. However, all the store operations are instrumented and the decision to call the instrumentation function for consistency region stores is based on a runtime check. *samhita* refers to the implementation that uses fine grain object granularity for consistency region updates. In this implementation we use static analysis to instrument only consistency region stores.

Figure 6.24 compares the strong scaling speedup of Pthreads and the Samhita implementations. We see that the two different Samhita implementations, *samhita_page* using mutex variables and the other *samhita_page+reduce* using reduction variables, track the Pthreads implementation very closely. This benchmark result clearly indicates that applications that are computationally intensive (the computation per particle is $O(n)$) can easily mask the synchronization overhead of Samhita enabling the application to scale very well. Also, in this case we notice that both the fine grain updates implementation of Samhita, *samhita_full* using mutex variables and *samhita_full+reduce* using reduction operation, do not perform as well as the page based implementation. We perform a run-time check to determine if a store is a consistency region store or not. Since, the molecular

dynamics application performs a large number of ordinary region stores we see an increased overhead due to this check. Static analysis to avoid ordinary region stores helps gain back most of this performance loss. This is reflected in the performance of both Samhita implementations, *samhita* using mutex variables and *samhita+reduce* using reduction variables. We see that the performance is very close to that of the page based implementation.

To conclude, the fine grain object granularity updates for consistency region stores that uses static analysis to not instrument ordinary region stores performs well for all scenarios. This emphasizes that we can provide a sufficiently strong memory consistency model and performant implementation, which will enable programmers to execute most threaded codes to run on clusters with trivial code modification.

6.3 Heterogeneous system performance evaluation

In this section we present our evaluation using the heterogeneous system. For this evaluation we execute the memory server on `cpu0` and the manager on `cpu1` on the host. Our current implementation uses Intel's InfiniBand proxy as the InfiniBand transport for Samhita Communication Layer (SCL). We present results on up to 120 threads being executed on two Intel Xeon Phi coprocessors. We execute 60 threads on each coprocessor for the 120 thread run, and 32 threads on each coprocessor for the 64 thread run. For each of these evaluation the data fits in the local cache of each Samhita compute thread.

6.3.1 Memory bandwidth micro-benchmarks

Figure 6.25 shows that Samhita achieves $330 - 360$ megabytes per second of read bandwidth for `mic0` and nearly 320 megabytes per second of read bandwidth for `mic1`. The `memcpy` bandwidth achievable for copying between two locally allocated memory regions is nearly 850 megabytes per second. Internal profiling results show that Samhita communication layer maps in the page at the full bandwidth achievable by the OFED implementation using Intel's SCIF communication library. The achieved lower bandwidth is due to the time taken by the software layer to copy the data

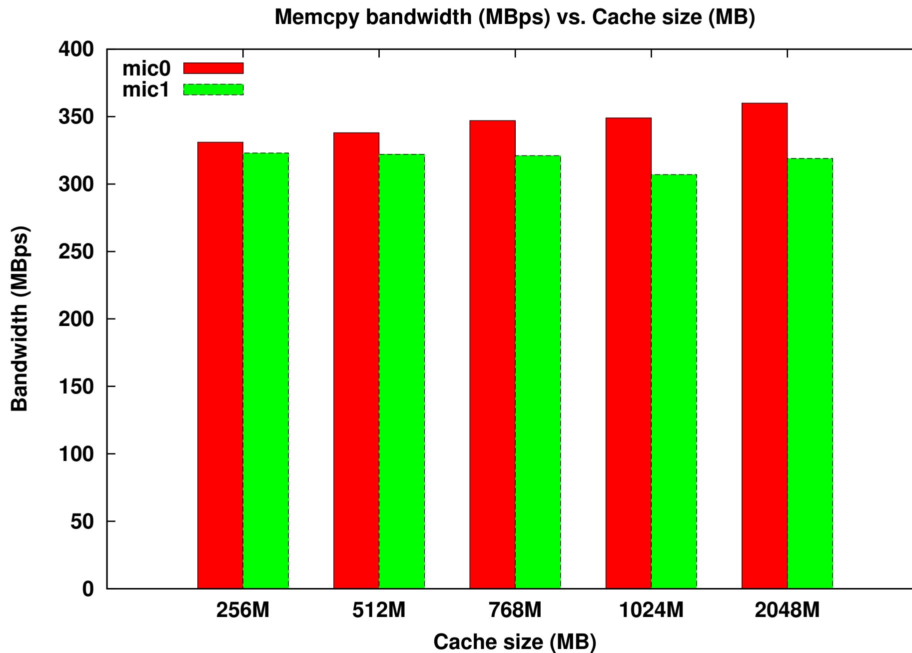


Figure 6.25: Memcpy bandwidth achieved with varying cache size.

from the host's main memory to coprocessor main memory and then copying the data to the actual memory region locally allocated on the co-processor.

The `memset` bandwidth results presented in Figure 6.26 reflect the overhead associated with Samhita's cache management technique. In this scenario the bandwidth achieved is $80 - 105$ megabytes per second. Note that the amount of data written (1GB) is more than the local cache size, except when the cache size is 1GB or 2GB. Nonetheless, Samhita achieves a write bandwidth that is one-third of its read bandwidth. These rates can be improved by supporting Intel's SCIF layer as one of the interconnection networks supported by Samhita's communication layer (SCL).

6.3.2 STREAM TRIAD

Figure 6.27 compares the strong scaling bandwidth achieved by the Pthreads and Samhita implementations. The Samhita implementation tracks the Pthread implementation reasonably well up to 8 threads before the synchronization costs start dominating the computational time. Figure 6.28 presents the weak scaling results for up to 120 threads. We see that the Samhita implementation

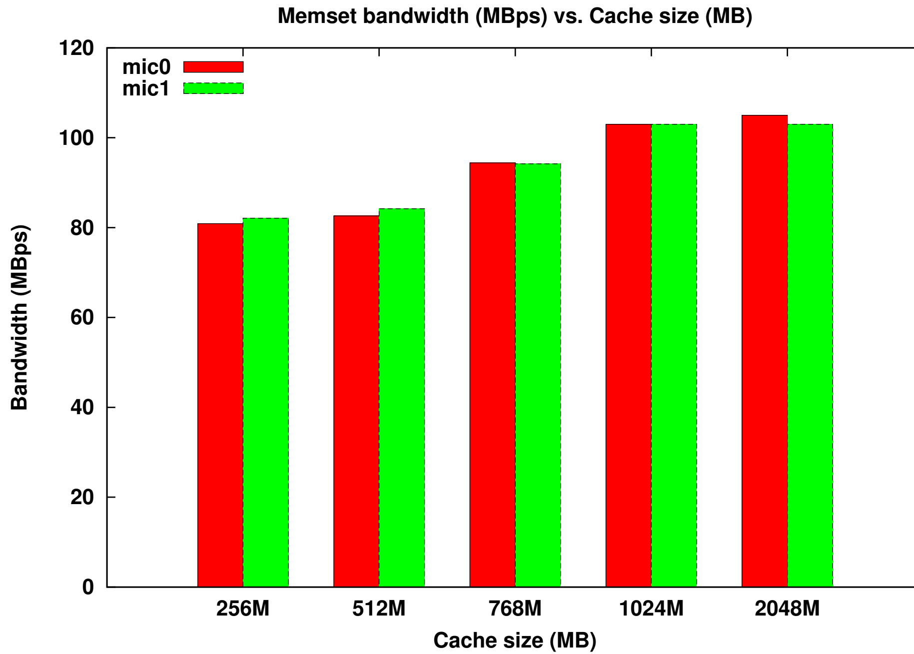


Figure 6.26: Memset bandwidth achieved with varying cache size.

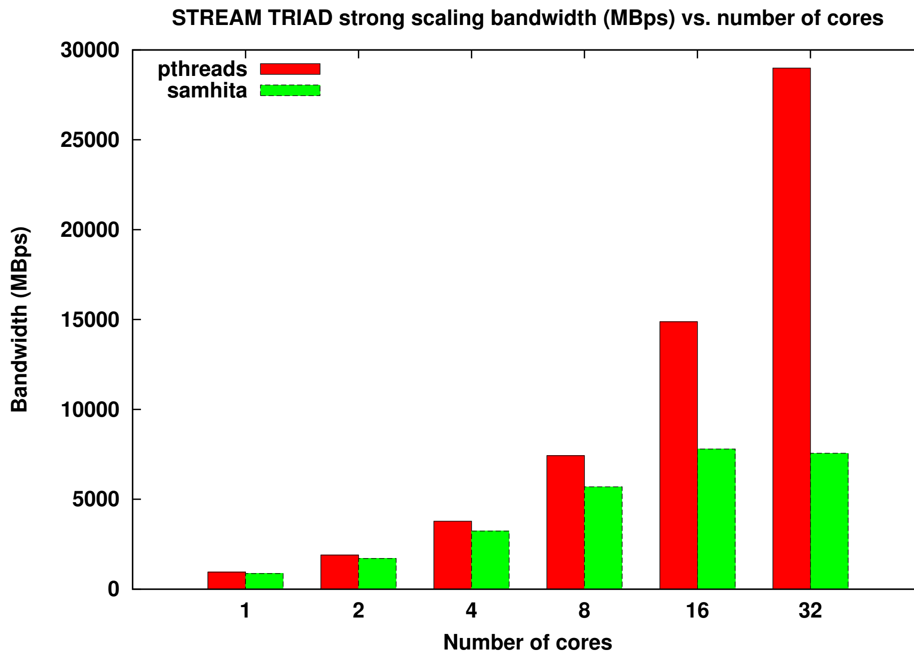


Figure 6.27: Achieved memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Vectors of dimension $n = 2M$.

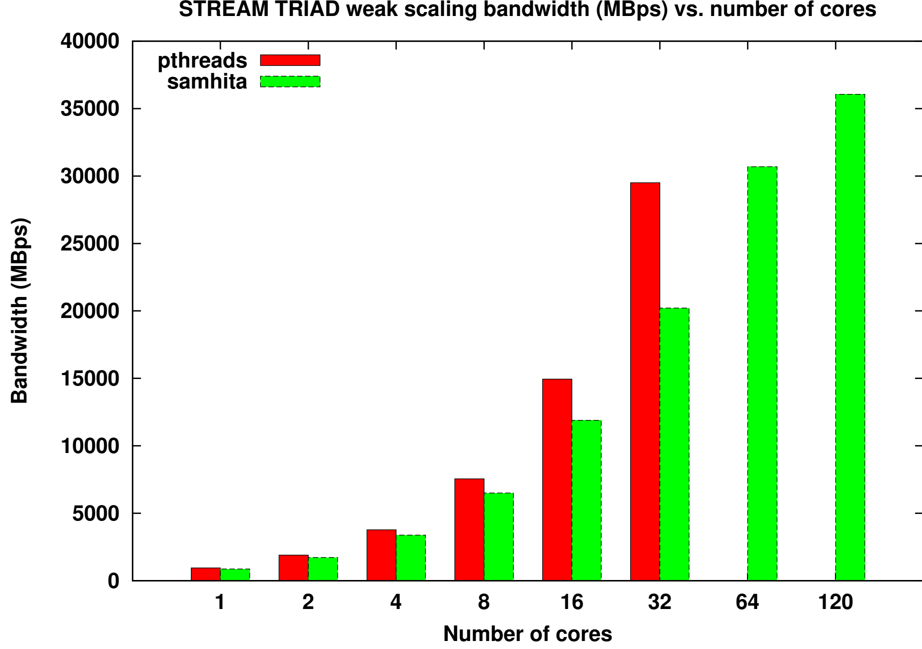


Figure 6.28: Achieved memory bandwidth vs. number of cores for STREAM TRIAD synthetic benchmark. Data size $3n$ and number of cores p are scaled proportionally, i.e., $3n/p$ is a constant. Problem size for $p = 1$ is $n = 2M$; problem size for $p = 120$ is $n = 240M$

tracks the Pthread implementation well up to 16 threads. It continues to scale till 64 threads before synchronization costs can not be amortized easily and the achieved bandwidth reduces.

6.3.3 Jacobi sweep

Figure 6.29 compares the strong scaling speed-up of Pthreads and Samhita implementation. The Samhita implementation tracks the Pthread implementation well up to 16 threads before synchronization costs start dominating the compute time. Figure 6.30 presents the weak scaling results for the Jacobi algorithm up to 120 threads. We see that the Samhita implementation tracks the Pthread implementation very closely up to 32 threads on a single coprocessor. It continues to scale reasonable well across both the coprocessors.

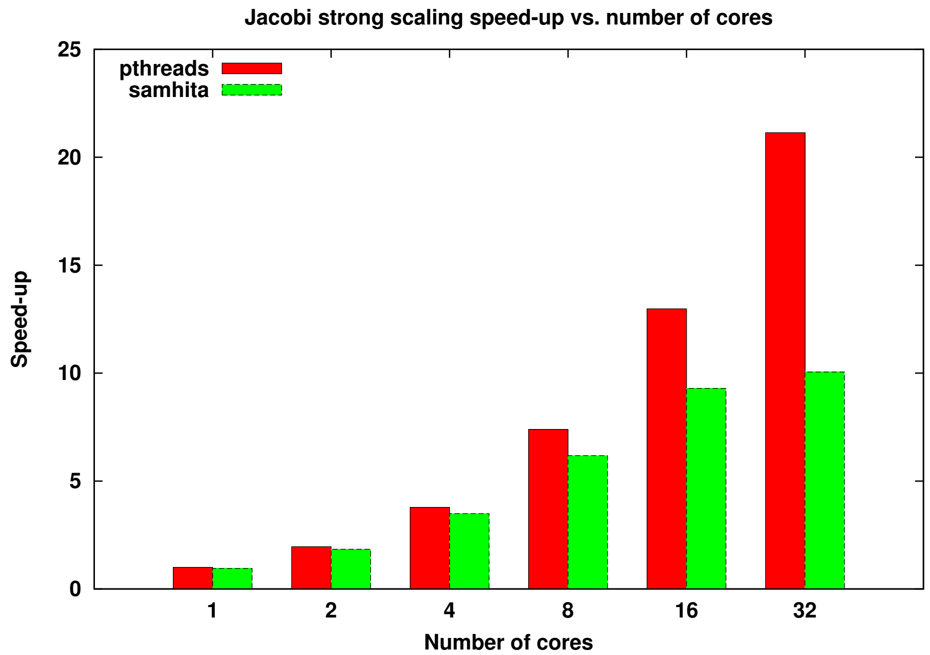


Figure 6.29: Parallel speedup vs. number of cores for Pthread and Samhita based Jacobi kernel. Speedup is relative to 1-core Pthread execution.

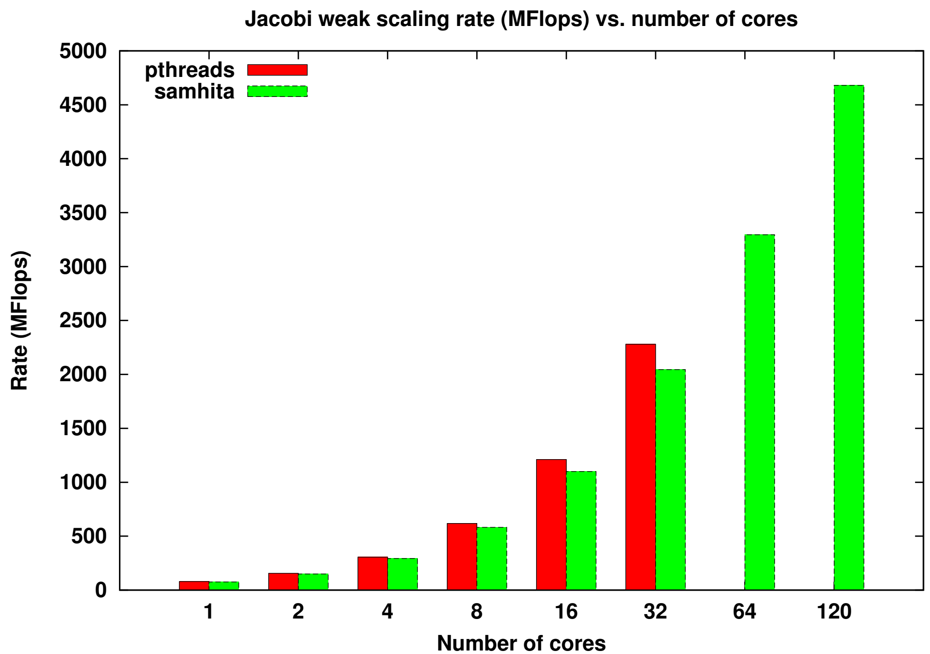


Figure 6.30: Computational rate vs. number of cores for Pthread and Samhita based Jacobi kernel. Data size $3n^2$ and number of cores p are scaled proportionally, i.e., $3n^2/p$ is a constant. Problem size for $p = 1$ is $n = 1344$; problem size for $p = 120$ is $n = 14722$.

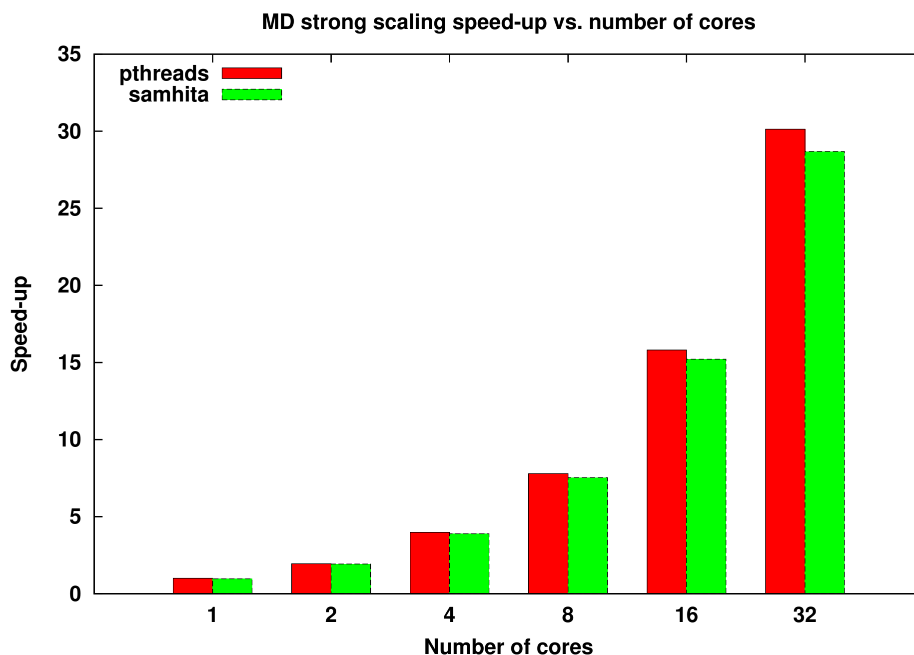


Figure 6.31: Parallel speedup vs. number of cores for Pthread and Samhita based MD application. Speedup is relative to 1-core Pthread execution.

6.3.4 Molecular dynamics application

Figure 6.31 compares the strong scaling speed-up of Pthreads and Samhita implementation. We notice that the Samhita implementation tracks the Pthread implementation very closely up to 32 threads. This indicates that computationally intensive application can easily mask the synchronization overhead in Samhita, enabling the applications to scale well.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this dissertation we introduced Samhita, a virtual shared memory system for non-cache-coherent systems. Samhita uses a novel cache based architecture that solves the problem of providing shared memory over independent memories. We have implemented the Samhita Communication Layer (SCL) to support quad data rate (QDR) InfiniBand switched fabric. We present results on up to 256 processing cores on two different clusters. To our knowledge, the results reported are the largest scale evaluation by an order of magnitude and achieve the highest performance of any distributed shared memory (DSM) system to date. Based on current and future interconnection trends we believe that Samhita is a viable alternative to message passing systems like MPI for developing application to run on cluster supercomputers.

We introduced regional consistency (RegC), a new memory consistency model that allows programs written using familiar threading models such as Pthreads to be easily ported to a non-cache-coherent system. The RegC model allows Samhita to provide an API that is very similar to the common Pthreads API. This makes porting most codes nothing but a "find and replace" operation. Performance results show that our Samhita implementations achieve computational speedup comparable to the original Pthreads implementations on a single node with trivial code modifications, and illustrate the performance improvements achieved by a simple programming model extension

and by distinguishing ordinary and consistency region stores. Weak scaling results on up to 256 processor cores demonstrate that scalable problems and algorithms scale well over Samhita.

We have ported Samhita for heterogeneous system that use Intel's Many Integrated Core (MIC) coprocessors. We currently use Intel's InfiniBand proxy to communicate between the host CPUs and the MIC coprocessors over the PCI express bus. We presented results on up to 120 processing cores running across two MIC coprocessors. To our knowledge, we are the first virtual shared memory system that supports the familiar thread based shared memory model between host and coprocessor processing cores. The initial performance results are encouraging and show the promise of providing virtual shared memory that spans the host and coprocessors, enabling existing shared memory codes to execute on these heterogeneous systems.

7.2 Future work

The future work can be divided into two categories: engineering improvements and further performance improvement optimizations. We first list the engineering improvements below:

1. We currently do not support Intel's K1OM ABI [59] in our store instrumentation tool. We will add support for the newer ABI, which will allow us to implement the more performant implementation of the RegC model.

2. Samhita communication layer (SCL) currently uses the Intel's InfiniBand proxy for communicating over the PCI express bus. We will add support for Intel's Symmetric Communication Interface (SCIF) [12]. This will reduce the overhead of communication over the PCI express bus.

3. Currently, in the heterogeneous implementation we store the *page twins* on the coprocessor's main memory. We can store these *page twins* on the host's main memory. This would not only increase the local cache size associated with each thread, but also allow us to offload the expensive *page diff* calculation to the host CPUs.

4. We can improve the alias analysis performed by the store instrumentation tool, which can im-

prove performance by reducing the number of redundant stores instrumented in a consistency region.

We now list the future performance optimization improvements that can be explored below:

1. Currently, the Samhita communication layer (SCL) is not aware of the topology on which the Samhita threads are running. We can explore the option of adding topology awareness to the communication layer. This will allow us to have efficient implementations of barrier synchronization operation and the reduction operation.

2. Currently, each thread on a compute node has its own local cache. We can explore the option of providing a shared global cache for all the threads of an application that are running on a single compute node. By providing a shared cache one can intelligently place threads that share data on the same compute thread, improving performance by exploiting statistical multiplexing provided by a shared cache.

3. Prefetching is a common technique used to improve cache based systems. We plan to explore pre-eviction, which complements prefetching, and which can be run in the background. This will remove the expensive *page diff* from the critical path during a capacity miss. In the presence of a shared cache we can have a single dedicated thread that can be used to perform the task of pre-eviction for all the threads that are executing on a single compute node.

4. The current implementation of Samhita does not provide an application developer any mechanism to support new consistency models. For one to implement a new consistency protocol one needs a way to annotate when memory consistency operations are needed to be performed. Samhita can provide APIs which will help the programmer with this annotation. The application programmer can then use this framework to implement the consistency protocol that works best for the application's memory access pattern.

5. Since Samhita provides virtual shared memory across non-cache-coherent systems we can explore the opportunities for load balancing and also opportunities of thread migration to data instead of moving data to compute.

Bibliography

[1] TOP500 Supercomputing Sites. http://top500.org, accessed June 1, 2013.

[2] CUDA parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html, accessed June 10, 2013.

[3] OpenCL - the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/, accessed June 10, 2013.

[4] The chapel parallel programming language. http://chapel.cray.com/, accessed June 11, 2013.

[5] Partitioned global adress space. http://www.pgas.org/, accessed June 11, 2013.

[6] X10: Performance and productivity at scale. http://x10-lang.org/, accessed June 11, 2013.

[7] elfutils. https://fedorahosted.org/elfutils, accessed June 14, 2010.

[8] GNU C library. http://www.gnu.org/software/libc, accessed June 14, 2010.

[9] Infiniband trade association. http://www.infinibandta.org, accessed June 14, 2010.

[10] The openfabrics allicance. http://www.openfabrics.org, accessed June 14, 2010.

[11] GNU M4. http://www.gnu.org/software/m4/, accessed June 15, 2013.

[12] Intel xeon phi coprocessor system software development guide. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-software-developers-guide.pdf, accessed June 15, 2013.

[13] memcpy. http://pubs.opengroup.org/onlinepubs/009695399/functions/memcpy.html, accessed June 15, 2013.

[14] memset. http://pubs.opengroup.org/onlinepubs/009695399/functions/memset.html, accessed June 15, 2013.

[15] mremap. http://man7.org/linux/man-pages/man2/mremap.2.html, accessed June 15, 2013.

[16] munmap. http://pubs.opengroup.org/onlinepubs/7908799/xsh/munmap.html, accessed June 15, 2013.

[17] Message passing interface forum. http://www.mpi-forum.org/, accessed June 9, 2013.

[18] OpenACC, directives for accelerators. `http://www.openacc.org/`, accessed June 9, 2013.

[19] OpenMP. `http://openmp.org/wp/`, accessed June 9, 2013.

[20] POSIX threads. `http://www.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html`, accessed May 27, 2010.

[21] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. Society of Industrial and Applied Mathematics (SIAM), Philadelphia, PA, third edition, 1999.

[22] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *Proceedings of the IEEE Compcon Spring '93*, pages 528–537, 1993.

[23] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[24] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society of Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[25] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, and Edward W. Felten. Virtual memory mapped network interface for the SHRIMP multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 142 – 153, 1994.

[26] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[27] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: Performance and cost. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 314 – 324, 1994.

[28] Benny Wang-Leung Cheung, Cho-Li Wang, and Kai Hwang. JUMP-DP: A software DSM system with low-latency communication support. In *Proceedings of the 4th International Workshop on Cluster Computing – Technologies, Environments, and Application (CC-TEA)*, 2000.

[29] UPC Consortium. UPC language specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[30] Gary S. Delp, David J. Farber, Ronald G. Minnich, Jonathan M. Smith, and Ming-Chit Tam. Memory as a network abstraction. *IEEE Network*, 5(4):34 –41, July 1991.

[31] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, March 1990.

[32] Antonio J. Dorta, Casiano Rodríguez, Francisco de Sande, and Arturo González-Escribano. The openmp source code repository. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 244–250, 2005.

[33] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. `http://people.redhat.com/drepper/nptl-design.pdf`, accessed June 14, 2010.

[34] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 434 – 443, 1986.

[35] Sandhya Dwarkadas, Nikolaos Hardavellas, Leonidas Konotothanassis, Rishiyur Nikhil, and Robert Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proc. of the 13th Intl Parallel Processing Symp and 10th Symp on Parallel and Distributed Processing*, 1999.

[36] Wu-chun Feng, Pavan Balaji, C. Baron, Laxmi N. Bhuyan, and Dhabaleswar K. Panda. Performance characterization of a 10-gigabit ethernet toe. In *Proceedings of the 13th Symposium on High Performance Interconnects*, pages 58–63, 2005.

[37] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. *ACM SIGOPS Operating System Review*, 23(5):211 – 223, December 1989.

[38] Guang R. Gao and Vivek Sarkar. Location consistency—a new memory model and cache consistency protocol. *IEEE Transaction on Computers*, 49(8):798–813, August 2000.

[39] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen mei W. Hwu. An asymmetric distributed shared memory model for heterogenous parallel systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, 2010.

[40] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80 – 112, January 1985.

[41] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–26, 1990.

[42] James R. Goodman. Cache consistency and sequential consistency. Technical Report Technical Report Number 61, IEEE Scalable Coherence Interface (SCI) Working Group, March 1989.

[43] William Gropp and Ewing Lusk. MPICH working note: The second-generation ADI for the MPICH implementation of MPI, 1996.

[44] Paul N. Hilfinger, Dan Bonachea, Kaushik Datta, David Gay, Susan Graham, Amir Kamil, Ben Liblit, Geoff Pike, Jimmy Su, and Katherine Yelick. Titanium language reference manual, v2.20. Technical Report UCB/EECS-2005-15.1, University of California, Berkeley, 2005.

[45] Lorin Hochstein and Victor R. Basili. A preliminary empirical study to compare mpi and openmp. Technical Report ISI-TR-676, Information Sciences Institute, 2011.

[46] Lorin Hochstein, Victor R. Basili, Uzi Vishkin, and John Gilbert. A pilot study to compare programming effort for two parallel programming models. *Journal of System Software*, 81(11):1920–1930, November 2008.

[47] Weiwu Hu, Weisong Shi, and Zhimin Tang. JIAJIA: A software DSM system based on a new cache coherence protocol. In *Proceedings of the 7th International Conference on High Performance Computing and Networking (HPCN) Europe*, 1999.

[48] Liviu Iftode, Jaswinder Pal, and Kai Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 277–287, 1996.

[49] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213 – 226, 1995.

[50] Wolfgang Karl and Martin Schulz. Hybrid-DSM: An efficient alternative to pure software DSM systems on NUMA architectures. In *Proceedings of the 2nd International Workshop on Software DSM*, 2000.

[51] Peter Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 13 – 21, 1992.

[52] Pete Kelher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenpoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, 1994.

[53] Jeffery Kuskin, David Oleft, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA)*, pages 302 – 313, 1994.

[54] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690 – 691, September 1979.

[55] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–, 2004.

[56] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, September 1979.

[57] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The stanford Dash multiprocessor. *IEEE Computer*, 25(3):63 – 79, March 1992.

[58] Kai Li. IVY: A shared virtual memory systems for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing (ICPP)*, pages 94 – 101, 1988.

[59] H. J. Lu, Milind Girkar, Michael Matz, Jan Hubička, Andreas Jaeger, and Mark Mitchell. System v application binary interface k1om architecture processor supplement. `http://software.intel.com/sites/default/files/m/0/c/e/8/5/44184-k1om-psabi-1.0.pdf`, accessed June 15, 2013.

[60] Creve Maples and Larry Wittie. Merlin: A superglue for multicomputer systems. In *Proceedings of the IEEE Compcon Spring '90*, pages 73 – 81, 1990.

[61] John D. McCaplin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[62] Frank Mueller. On the design and implementation of DSM-Threads. In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315 – 324, 1997.

[63] Ming Chung Ng and Weng Fai Wong. Orion: An adaptive home-based software distributed shared memory system. In *Proceedings of the 7th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 187 – 194, 2000.

[64] Ranjit Noronha and Dhabaleswar K. Panda. Designing high performance DSM systems using InfiniBand features. In *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 467 – 474, 2004.

[65] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1 – 31, August 1998.

[66] Chistian Osendorfer, Jie Tao, Carsten Trinitis, and Martin Mairandres. ViSMI: Software distributed shared memory for InfiniBand clusters. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA)*, pages 185 – 191, 2004.

[67] Markus Pizka and Christian Rehn. Murks - a POSIX thread based DSM system. In *Proceedings of the International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 642 – 648, 2001.

[68] Jelica Protić, Milo Tomašević, and Veljko Milutnović. A survey of distributed shared memory systems. In *Proceedings of the 28th Annual Hawaii International Conference on System Sciences (HICSSS)*, volume 1, pages 74 – 84, 1995.

[69] Muralidharan Rangarajan and Liviu Iftode. Software distributed shared memory over virtual interface architecture: Implementation and performance. In *Proceedings of the 4th Annual Linux Showcase & Conference, Extreme Linux Workshop*, 2000.

[70] Bratin Saha, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Mohan Rajagopalan, Jesse Fang, Peinan Zhang, Ronny Ronen, and Avi Mendelson. Programming model for a heterogeneous x86 platform. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, pages 431–440, 2009.

[71] Daniel J. Scales, Kourosh Gharachorloo, and Chadramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, 1996.

[72] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture (ISCA)*, pages 234–243, 1987.

[73] Thobias S. Trevisan, Vítor Santos Costa, Lauro Whately, and Claudio L. Amorim. Distributed shared memory in kernel mode. In *Proceedings of the 14th Symposium on Computer Architecture and High Performance Computing (SCAB-PAD)*, 2002.

[74] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM (CACM)*, 33(8):103–111, August 1990.

[75] Hu Weiwu, Shi Weisong, Tang Zhimin, and Li Ming. A lock-based cache coherence protocol for scope consistency. *Journal of Computer Science and Technology*, 13(2):97 – 109, March 1998.

[76] Songnian Zhou, Michael Stumm, and Tim McInerney. Extending distributed shared memory to heterogeneous environments. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS)*, pages 30 – 37, 1990.