博士論文

Doctoral Dissertation

A Decentralized Implementation of Software Distributed Shared Memory
（ソフトウェア分散共有メモリの非集中型実装）

遠藤 亘

Wataru Endo

学籍番号: 48-177403

指導教員: 田浦 健次朗 教授

東京大学大学院 情報理工学系研究科 電子情報学専攻

Department of Information and Communication Engineering
Graduate School of Information Science and Technology
The University of Tokyo

東京大学
THE UNIVERSITY OF TOKYO

Abstract

The low application productivity of supercomputers has been a remaining problem in the history of parallel computing. Most of today's supercomputers are distributed-memory machines, which force application programmers to manually issue communication requests. Shared memory, the other memory model, is considered more productive than distributed memory, but it is widely believed that shared-memory programming is not scalable due to the model itself. In the 1990s, many researchers had developed software systems called Distributed Shared Memory (DSM) systems that realize the shared-memory model on top of distributed-memory machines. Nowadays, however, only very few researchers are investigating this idea because of the observed results of its poor performance.

The hardware environment for supercomputing has been largely improved for two or three decades. The interconnection network between nodes has also been significantly improved. The technology trends of both hardware and software have enlightened the possibility of better DSM systems than traditional DSM systems. For example, there were very few DSM systems that utilized the interconnection feature called Remote Direct Memory Access (RDMA), which appeared in the 2000s. Most of RDMA implementations can only support point-to-point communications and it is necessary to implement a decentralized coherence protocol for DSMs to utilize RDMA. Multi-core architectures have become dominant in today's processors and utilizing multi-threading in a node possibly improves the performance of DSM systems, but traditional DSM systems are not optimized for modern hardware.

In this dissertation, we propose a decentralized approach to implement distributed shared memory based on RDMA and multi-core architecture. This approach is demonstrated as a software DSM system Menps. To decentralize the coherence protocol of Menps, we have introduced three novel ideas: the floating home-based method, the hybrid invalidation of logical timestamps and write notices, and the fast release. The evaluation results of Menps show that the proposed coherence protocol can improve the performance of software distributed shared memory compared with the existing methods.

To exploit the performance of RDMA on multi-core architectures, we have developed a new communication library MECOM for Menps. MECOM is a communication library based on a new technique to implement software communication offloading using user-level threading. The proposed software offloading technique using atomic operations improves the message rates of RDMA communications in multi-threading environments compared with the conventional locking schemes.

To accelerate the thread scheduling in Menps, we also have implemented a user-level threading library ComposableThreads. ComposableThreads provides compile-time parametricity for different purposes in system programming. This library can be not only employed as the tasking layer for Menps, but also customized as an efficient general-purpose threading library.

This dissertation describes the design and implementation of the DSM library Menps. As an integrated research prototype for transparently executing shared-memory multi-threading applications, Menps provides many interesting insights about how we can design parallel computer architecture.

Acknowledgment

First, I would like to especially thank my supervisor Professor Kenjiro Taura for his continuous support both on my research and PhD life. This work would never be accomplished without his patience for my research activities. We had so many discussions to improve my work and I have borrowed many ideas that originate from his insights. I have been always encouraged by his strong enthusiasm for the improvement of parallel computing systems.

I would like to express my deep gratitude to Assistant Professor Shigeyuki Sato. Without his support, I would neither enter the doctoral course nor continue conducting this research. He was also the first person who really understood what my whole work is all about. His advice was always quite detailed and have significantly strengthened the presentation of my study.

All of the committee members gave me many important comments for defense preparation. I had several great discussions about the early work of this dissertation with two of the advisors Associate Professor Hidetsugu Irie and Professor Masashi Toyoda. I appreciate that both of them kindly spent their time sharing and improving my ideas. Professor Masaru Kitsuregawa, Associate Professor Ryota Shioya, and Dr. Hiroya Matsuba gave me insightful advice to clarify the strengths of my study at the pre-defense.

I am very grateful to Assistant Professor Hiroko Midorikawa for her comments about my DSM study at the workshops, pre-defense, and so on. We had several nice discussions in order to "revive" this research field. With her solid comments, I could enhance both the implementation and presentation qualities for preparing the defense.

I would like to show my gratitude to Dr. Sriram Krishnamoorthy, who kindly gave me technical advice in several offline meetings. I was really surprised by his diverse expertise in this research field. It was a very encouraging opportunity for me to discuss my ongoing work with him.

I would also like to thank Ilya Zhukov, who was our mentor during the International HPC Summer School 2018. We had great discussions on improving my work. His valuable opinions based on the performance analysis have been incorporated into my research.

I would like to thank all of our laboratory members who I have enjoyed the research with. I had a great collaboration with Takuya Fukuoka in the last two years of my doctoral course. It was an exceptional experience in my life to conduct the research together with him. I have enjoyed creating in a new wonderful logging tool with Shunpei Shiina and Tomokazu Higuchi. I would also like to appreciate Shintaro Iwasaki for discussing my rough ideas both online and offline. I appreciate all of the members who participated in the TP-PARSEC project: Dr. An Huynh, Christian Helm, and Byambajav Namsraijav. In addition, most of my work for this dissertation is based on the research conducted by Shigeki Akiyama and Jun Nakashima, who gave me much important advice until my master's course.

I had been helped many times by the former secretary of our laboratory Hiromi Takano. I remember how she was anxious about our laboratory life. I am also thankful to the current secretary Sachie Ikeya for handling lots of paperwork.

Finally, I would like to appreciate my family who has brought me up and let me take this long journey.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Background

Over the past decades, supercomputers have been created by many institutions to conduct a large amount of computation. The typical purpose of supercomputing is to accelerate large-scale scientific computations. The continuous improvement of the computation power of supercomputers is key to proceed with many research fields in science and technology. The research area related to supercomputing is also called high-performance computing (HPC).

A modern supercomputer is composed of multiple compute nodes, which are connected by an interconnection network. The users select a set of the compute nodes to run their application programs and those nodes cooperatively execute the same program in parallel. Because each node owns a different memory resource, the memory is distributed from a viewpoint of the whole program. This hardware form is abstracted as a memory model called distributed memory (Section 2.1), where each parallel execution unit (= a process) has its distinct address space. The processes can communicate with each other to share the intermediate results of the execution. The communication is explicitly expressed by the programmers in the form of message passing or other communication models. The de-facto standard interface of distributed-memory programming is Message Passing Interface (MPI) [132], which is used by most of today's distributed-memory applications.

In contrast to the performance benefit of supercomputers, their application productivity has been often criticized over decades. To execute application programs efficiently on supercomputers, programmers usually need to port their applications for MPI. Porting to MPI requires a lot of effort because the programming model of MPI is significantly different from those of the sequential codes. This low application productivity prohibits rapid prototyping of applications for supercomputers and makes them expert-friendly.

The opposite of distributed memory is called shared memory (Section 2.2). In a shared-memory architecture, all of the parallel computation units (= threads) share the same address space and the same content of data. Most of today's processors are multi-core processors, which provide shared-memory interfaces for programmers at the hardware level. Shared memory is considered productive for application programmers because the shared-memory programming models provide a global view of memory similar to that of sequential codes. The programmers are familiar with shared memory through the threading interfaces specified by each programming language and each operating system.

Distributed shared memory (DSM) [56] (Section 2.2.2) is the form of shared-memory implementations that provides shared-memory programming interfaces for the applications but is internally implemented on distributed-memory machines. To make memory coherent on distributed memory, the processes in DSM systems synchronize their caches using the interconnect network. The communication protocols for synchronizing data between nodes are called coherent protocols. The programming model of DSM exactly matches with other shared-memory architectures because it differs in its implementation method. In the 1990s, there were a number of researchers who were interested in developing the DSM systems, and they have proposed many valuable ideas that appear in slightly different contexts in modern parallel computing. However, due to the scalability problems of the research prototypes at that time, software DSM has lost attention from the HPC community since the 2000s, and few researchers are still trying to develop software DSM systems today.

In the 2000s, Partitioned Global Address Space (PGAS) systems (Section 2.2.3) appeared as a replacement for DSM systems. PGAS provides a non-coherent global-view address space separated from local address spaces. Although PGAS provides the global address space for application developers, its programming models are still far from shared-memory programming models. In shared-memory systems, memory reads or writes are observed synchronously, but real processors are handling them properly to accelerate memory accesses. The application programmers are implicitly assuming that the latency of memory accesses is hidden by the memory system. However, because the PGAS interfaces provide the synchronous memory access but prohibit coherent caching in the implementation level, the underlying system has no chance to accomplish latency hiding. Due to this inherent problem, in PGAS systems, programmers need to consider how to aggregate the memory accesses by themselves, which often leads to poor application productivity (e.g. [201]).

In this dissertation, we have focused on preserving the infrastructure for parallel programming which is commonly accepted by the community. The existing infrastructure is based on many computing components such as the applications, hardware, compilers, and operating systems. Introducing a new programming model (e.g. PGAS) may provide better performance than preserving the existing infrastructure (e.g. running existing shared-memory programs on DSM systems), but it does not mean that such a new programming model can immediately replace all of the existing frameworks and tools. It is apparent that there are a huge number of the existing applications or tools which cannot follow the drastic changes of the underlying layers. In that sense, shared memory is one of the solid abstractions for the modern hardware and accelerating the shared-memory infrastructure has a significantly important role to gradually improve our computing environments for end-users. Therefore, we have tried to revisit the idea of distributed shared memory rather than focusing on modifying the programming models.

Ideally, we think that the following properties should be achieved for productive parallel computing environments:

Application productivity

DSM systems provide productive programming environments because application programmers can write programs with global pointer-based accesses. It is believed that PGAS systems are more productive than message passing, but compared with DSM, they still have strong restrictions on the interface.

Application portability

It should be easy to run the applications on different platforms. DSM systems are also good at this point because the application programs can be migrated from ordinary multi-threaded programs for today's multi-core architecture. On the other hand, the application portability of PGAS systems is often not good because each of them provides a different programming model incompatible with shared-memory programming. For message passing, MPI provides a portable interface that enables the applications to run on different supercomputing environments.

System portability

It enables the systems to run on different platforms. Ideally, they should be purely library-based implementations in order to avoid relying on special compiler techniques or kernels, which cannot be easily employed in current supercomputers. Because page-based DSM systems (Section 3.4.1) can be run without any special compilers or kernels, they are good candidates for testing on current supercomputers.

Efficiency

It is important to check both the sequential performance and the parallel speedups because the primary reason to use distributed-memory systems is their performance. It is widely believed that efficiency is the main problem of software DSM systems. Both PGAS and message-passing can be efficiently implemented more easily than DSM because of their primitive interface which can simply be mapped to the hardware features.

System maintainability

It is also important to develop the systems which can be maintained easily even in research prototypes.

When productivity and portability are seriously concerned, page-based DSM is the best candidate for preserving the existing infrastructure compared with other methods for easing distributed-memory programming. It does neither require any compiler modification, kernel modification, nor special hardware. The remaining problem is its efficiency, but due to the lack of recent research prototypes, it has not been easy to understand the fundamental performance problems of software DSM.

1.2. Motivation

The main problem establishment of this dissertation is *"how to accelerate page-based software DSM systems transparently on current computing infrastructure."* In this statement, there is strong pressure on the software DSM systems from both sides: the underlying system side and the application side. To maximize the benefit of page-based DSM, the applications running on those systems should not be changed from the single-node shared-memory programs. Without any explicit information from the applications, the underlying DSM systems can only work as instructed by the application and prepare for future events in the background by guessing the future behavior. There should also not be any modification on the compilers, the kernel, and the hardware. Every component of them has pressure to limit the design space of user-level page-based DSM systems. Thus, what the system programmers for DSM systems can do in this problem establishment is to develop the methods for an efficient runtime system that can exploit the performance of the existing parallel computing systems while preserving the semantics of the running application.

In the past two decades, the underlying environments of DSM systems have been largely improved. The first is the improvement of network performance. As pointed out by Ramesh et al. [158], the difference between inter-node communications vs. intra-node communications has been shrinking over the decades. Recent hardware technologies such as silicon photonics may further shrink the bandwidth gap of those.

In addition to the improved performance properties, recent interconnection networks support a feature called Remote Direct Memory Access (RDMA) (Section 2.3.1). RDMA provides high bandwidth and low latency communications by its zero-copying approach. It also bypasses the kernel system calls and enables the user-level programs to access the hardware with minimal overhead. However, for system programmers, it is important to be aware that the interface of RDMA is significantly different from other common communication interfaces including MPI and TCP/IP. RDMA has several severe restrictions about memory transfers, which complicate the system development. As parallel programming improves the performance of sequential programs with an additional development cost, RDMA provides efficient data transfers with its restrictive interfaces.

Argo [100] is a recent example of software distributed shared memory systems designed to exploit the performance of modern interconnects for efficient coherence actions. All of the coherence actions of Argo are purely implemented in RDMA. However, because Argo is based on conventional techniques such as directory-based coherence and fine-grained diff merging, it is still far from the communication patterns which can be efficiently implemented by RDMA.

We have noticed that the hardware assumptions for the traditional DSM implementations are not matching the modern hardware characteristics (see also Section 3.2). Although the existing work including Argo [84, 100, 143, 92, 147, 159] (described in Section 6.1.2) has tried to implement an efficient RDMA-based protocol, their protocol still suffers from large software overhead to process fine-grained communications.

The second improvement is the wide adoption of multi-core processors. In the processors' trend, the number of cores in recent processors has been continuously increasing to achieve better performance. The extreme case of this trend is the emergence of many-core processors, which have hundreds of cores per processor. Traditional DSM systems assumed single-core or small multi-core environments, but this assumption has already changed. We think that the coherence protocols and the implementation strategy of DSM systems should also follow the trend of increasing core counts. User-level threading (Section 2.6.2) is beneficial for scheduling both application tasks and system tasks with inter-node communications in a unified manner.

We have been thinking that many problems are left in the existing work of DSM systems. The concept of shared memory is similar to what RDMA provides in their one-sided behaviors, but conventional DSM systems perform differently from the RDMA semantics. Why is there a huge gap between traditional coherence protocols and the underlying hardware? Since RDMA is conceptually similar to shared-memory interfaces, there should not be such a huge difference between the application interface and the real hardware feature. Also, how can we efficiently schedule the CPU workload coming from both the system and application sides? If most of the runtime system is written in software as in software DSM, they should be efficiently scheduled in an integrated manner, but the scheduling problem of communications is still an active research topic apart from DSM. Therefore, this dissertation tries to seek answers to these problems.

1.3. Contributions of this dissertation

The main motivation of this dissertation is to adapt DSM systems into modern computing infrastructure with RDMA and multi-core architecture. To demonstrate our ideas about DSM, we have implemented a software page-based DSM library named Menps (Chapter 3) from scratch. The coherence protocol of Menps is focused on minimizing software overhead of coherence by approaching the semantics of RDMA.

To adapt DSM to RDMA, it is necessary to make the coherence protocol decentralized because RDMA communications are normally point-to-point and cannot support broadcasting or multicasting. To overcome the challenges of decentralizing the coherence protocol of DSM, we present three novel techniques in this dissertation. First, we developed a method called *floating home-based method* which automatically "floats" the shared data. Second, we propose a new invalidation scheme that can eliminate the use of centralized cache directories, which is a typical scalability bottleneck in DSM systems. Our approach combines logical timestamp-based coherence and write notices to accomplish "directory-less" cache invalidations. Third, based on both the floating home-based and the new invalidation scheme, a new feature called *fast release* is proposed to shrink the latency of barriers. With these three methods, Menps is implemented as a DSM library that can transparently execute OpenMP programs on distributed-memory machines. The evaluation shows that Menps can scale some of the existing shared-memory programs with minimal modifications.

To implement the efficient DSM system based on the proposed protocol, it is also necessary to provide inter-node efficient communications. Although recent HPC interconnects are assumed to achieve low latency and high bandwidth communication, in practical terms, their performance is often bounded by the network software stacks rather than the underlying hardware because message processing requires a certain amount of computation in CPUs. To exploit the hardware capacity, some existing communication libraries provide an interface for parallelizing accesses to network endpoints with manual hints. However, with growing core counts per node in modern clusters, it is increasingly difficult for users to efficiently handle communication resources in multi-threading environments.

We have implemented a low-level communication library MECOM (Chapter 4) that can automatically schedule communication requests by offloading them to multiple dedicated threads via lockless circular buffers. To enhance the efficiency of offloading, we developed a novel technique to dynamically change the number of offloading threads using a user-level thread library. We demonstrate that our offloading architecture exhibits better performance characteristics in microbenchmark results than the existing approaches.

Additionally, to exploit the parallelism of today's multi-core architectures in Menps, we have also developed a user-level threading library ComposableThreads (Chapter 5). ComposableThreads not only provides useful tools to develop Menps and MECOM, but also it works as a standalone user-level threading library with compile-time parametricity.

The main research questions in this dissertation are described with short answers below:

1. Is it possible to build a decentralized DSM system efficiently using the latest hardware?
   - Short answer: Yes, because Menps is based on a decentralized approach that exploits the performance of RDMA. The coherence protocol of Menps is decentralized by nature because it does not have any data structures which are fixed to a certain process.
2. How should we construct a multi-threaded communication library which can accelerate a software

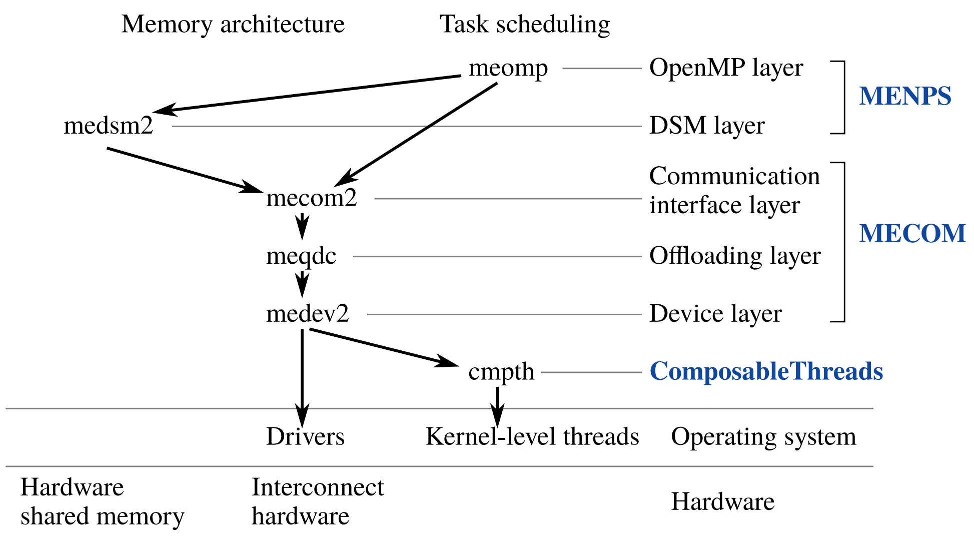


Figure 1.1.: Software stack of Menps.

DSM?

- Short answer: Software communication offloading based on user-level threading is a reliable solution for this problem. It is not easy to build a multi-threaded communication library without the scheduling knowledge of the other parts of the program. If the entire program is assumed to running on user-level threads, both the communication and the other tasks can be effectively scheduled. This dissertation deals with the performance of the communication library using Menps, but this offloading scheme is generally applicable to other applications.

3. It is possible to decompose the design of a user-level threading library which is suited for low-level system development?

   - Short answer: Yes. ComposableThreads provides the low-level functionalities for user-level threading while keeps the comparable performance with MassiveThreads. Menps can be considered as "one of the applications" of ComposableThreads because it can perform as the underlying threading system of Menps. ComposableThreads also generalizes the idea of software communication offloading as a lock delegation method (Section 2.5).

1.4. Structure of this dissertation

The software stack of Menps is shown in Figure 1.1. This dissertation explains the high-level DSM component first and going down to the lower-level parts in this software stack. The dissertation is composed of the following chapters:

- Chapter 2 is the background part to understand the techniques of Menps. This chapter briefly explains the basic technical topics including distributed-memory vs. shared-memory architectures, memory consistency models, user-level threading and mutual exclusion.
- Chapter 3 proposes a decentralized distributed shared memory library Menps, which combines three novel techniques (logical timestamp-based coherence, floating home migration, and fast release) for decentralized coherence.
- Chapter 4 introduces a low-level communication library MECOM to implement Menps. MECOM is based on communication software offloading, which improves the message rates with multi-threading

environments.

- Chapter 5 explains a user-level threading library ComposableThreads, which accomplishes a modular and customizable approach to implement user-level threading.
- Chapter 6 briefly lists the existing studies about DSM systems, communication libraries, and task-parallel systems. It also mentions the other possible research directions which were not chosen for MENPS including compiler-based techniques, PGAS systems, and so on.
- Finally, Chapter 7 concludes the dissertation summarizing the contributions and future work of three new systems.

2. Background

2.1. Distributed-memory programming interfaces

It is easy to start thinking distributed-memory interfaces first rather than other memory interfaces since they are close to how real modern computer architectures are constructed. *Distributed memory* is a memory interface of parallel computers that does *not* share the memory among the execution units (i.e. *processes*). Each process has its distinct address space and executes a sequential program with a distinction that it can communicate with other processes if necessary. Programmers need to explicitly specify in their programs how each process transfers messages with each other. Distributed memory is a simple abstraction of the hardware architectures of parallel computers compared to shared memory. Distributed memory has several different communication models such as message passing, active messages, and remote memory access. All of these models require explicit communication calls in the programs.

2.1.1. Message passing

Message passing is a communication model such that a message is sent when both a sender process and a receiver process agree with their message transfer. Message Passing Interface (MPI) [132] is, as its name indicates, the current de-facto standard interface for message passing. MPI's specification calls its message-passing model point-to-point communication. There are many MPI implementations from different vendors, but currently, most of them are derived from two famous implementations: MPICH [3] and Open MPI [187].

The execution model employed in MPI is called SPMD (Single Program Multiple Data), which means that the same program is executed on multiple execution units operating on different parts of data. MPI programs simultaneously execute multiple processes called *ranks* and normally those processes do neither increase nor decrease during the execution.

The basic interface of message passing is a pair of send/receive functions, e.g. the pair of `MPI_Send` and `MPI_Recv` in MPI:

```
int MPI_Send(const void* buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Recv(void* buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm, MPI_Status* status);
```

Both the sender and receiver processes specify their buffers (= `buf` and `count`), another rank (= a process ID), and the tag value for distinguishing communications. When the tag matching succeeds, the communication is done by the MPI runtime system, and those function invocations return to the caller functions. Other features in these function signatures including datatypes or communicators are not used in the explanation later.

Although MPI is the de-facto standard API for the current supercomputers, the idea of message passing can be found in other places of the parallel computing community. Historically, there was another programming interface for message passing called Parallel Virtual Machine (PVM) [7] competing against MPI in the 1990s. Moreover, the usage of message passing is not limited to HPC; for example, Elixir [155] and Go [184] are the recent parallel programming languages based on message passing.

2.1.2. Active Messages (AM)

Active Messages (AM) [191] is a communication model that sends a message with a function (= *handler*) and invokes the function on the receiver side. AM is not included in MPI but is implemented in major libraries including GASNet [36]. AM is almost the same concept of remote procedure call (RPC), but it was introduced to be more lightweight than RPC so that it can eliminate buffering.

For example, in GASNet, this function sends an active message to another node with a message handler ID and `M` integer arguments:

```
int gasnet_AMRequestShortM(
    gasnet_node_t dest, gasnet_handler_t handler,
    gasnet_handlerarg_t arg0, ..., gasnet_handlerarg_t argM-1);
```

The implementation of GASNet may or may not call the message handler function during computations (via interruption or other mechanisms). In today's computing systems, it is costly to use interruptions for communication completions and polling is instead used. For polling-based implementations in GASNet, it is necessary to explicitly call the following polling function to process active messages from remote nodes:

```
int gasnet_AMPoll();
```

2.1.3. Remote Memory Access (RMA)

Remote memory access (RMA) is a communication model in which an initiator process solely issues communication between processes. As an example, MPI is not restricted to the message-passing model (or the point-to-point model) but also includes RMA. The importance of RMA is characterized by the fact that it can be mostly mapped to RDMA operations, which are described in Section 2.3.1.

The examples of RMA operations are get/put functions, which read and write the data on the remote side. Although RMA is one of the interfaces for distributed-memory programming, the concept of RMA is close to shared memory.

MPI includes the RMA interface since MPI-2 and added functionalities in MPI-3. Although the semantics of MPI RMA is similar to RDMA, they are not exactly matching the underlying hardware interfaces [174].

2.2. Shared-memory programming interfaces

2.2.1. Shared memory

Shared memory is a memory model in parallel computers which shares a single shared address space in all of the processors. The important properties of shared memory can be summarized as these two points:

Global addressing

Every object (or every word) on memory can be accessed by all of the processors (or *threads*). There is no restriction on which thread can access the data, but it does not mean that those objects need to be always synchronized in physical time.

Implicit communications

Programmers do not explicitly issue communications in their shared-memory programs. Since it is impossible to make memory coherent without communications between processors, communications are implicitly happening within the underlying memory architecture. Shared-memory systems can reorder memory accesses to hide communication latency.

The structure of shared-memory programs is usually similar to that of sequential programs because a single global address space is provided as in the sequential case. With the global-view memory abstraction, shared memory gives an intuitive programming style to the programmers.

Most of the modern commodity processors are multi-core processors (also known as chip multiprocessors (CMP)) which provide a shared-memory view to programmers. Most of the Non-Uniform Memory Access (NUMA) systems also implement a shared-memory model, which are called cache-coherent Non-Uniform Memory Access (ccNUMA). When the processors are distributed, they are called distributed shared memory (DSM) systems (Section 2.2.2).

It is known that shared-memory programming styles are error-prone because it gives an illusion as if shared-memory programs behave "almost" the same as the sequential ones. The detailed semantics are defined by the memory consistency models (Section 2.4) and are sometimes counter-intuitive for most of the programmers. To solve this problem, shared-memory systems with the guarantee of *determinism* have also been explored [34] (see also Section 6.3.1).

Typical shared-memory systems have a caching mechanism managed separately in each process, which is called private caches. In principle, it is possible to consider an architecture in which there is no private cache in every core and the entire memory is physically shared by all of the cores, but such an architecture is far from the real hardware. There is an interesting observation here: most of the current hardware shared-memory systems are *indeed* distributed. The private caches are synchronized via internal implicit communications. Communication protocols for shared-memory systems are called *coherence protocols*.

Although consistency and coherence are often confused, they have different meanings [177]. Consistency (or memory consistency) definitions provide rules about correct memory accesses. Programmers need to understand consistency to make correct shared-memory programs. On the other hand, coherence (or cache coherence) is a functionality of shared-memory systems to make the caches invisible to software. The programmers cannot determine whether memory accesses are cached only by observing the results of loads and stores. In other words, cache coherence only affects the performance of memory accesses, but memory consistency changes the correct semantics of application programs.

The memory consistency models will be described in Section 2.4. The issues around memory consistency models are sometimes tricky. The implementation of cache coherence especially by the software is described in Section 3.4.

2.2.2. Distributed Shared Memory (DSM)

Distributed Shared Memory (DSM) [56] is the form of a memory system that provides shared-memory interfaces on distributed-memory architectures. DSM is *not* a programming interface because its interface exactly corresponds to what shared memory provides. DSM is defined as a possible implementation strategy of shared-memory systems.

As mentioned in Section 1.1, software DSM systems had been extensively studied in the 1990s, which will be listed in Section 6.1. One of the famous and widely used examples of software DSM systems is TreadMarks [102, 103]. Menps (Chapter 3) is also categorized as a software DSM system and followed the traditional methodology of the DSM research while making improvements for modern hardware.

At the hardware level, the difference between distributed-memory and shared-memory machines is ambiguous in general. For example, most of the current multi-core processors have a private L1 cache for every core, which is not physically shared among the cores in fact. The content of private caches is synchronized via coherence actions, which are also implemented by the DSM systems.

Moreover, it is not always possible to distinguish DSM from other shared-memory architectures such as ccNUMA because each node of ccNUMA owns its memory and synchronizes it via coherence protocols. In the hardware level, the distinction between ordinary shared memory systems and distributed shared memory systems may not be visible indeed.

The difference between PGAS and DSM is often confused in the literature. In this dissertation, DSM is defined as transparent shared-memory systems with coherent caches that can be globally accessed via ordinary load and store operations. There are still several systems that cannot be simply mapped to either described in Section 6.2.

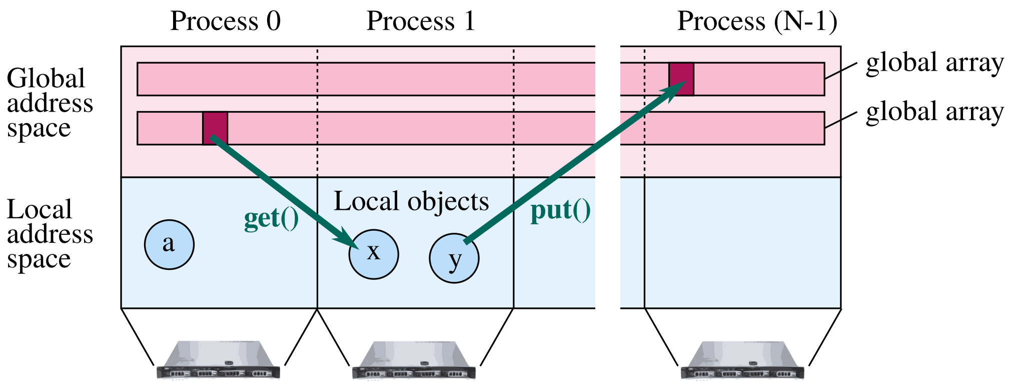


Figure 2.1.: Partitioned Global Address Space (PGAS) model.

2.2.3. Partitioned Global Address Space (PGAS)

Partitioned Global Address Space (PGAS) [144, 66, 137, 46, 43, 44, 133, 135, 19] is a memory interface that provides both a global address space and local distinct address spaces. Figure 2.1 illustrates the PGAS model. When accessing the global memory in PGAS, programmers need to invoke *get/put* functions[1] with the pair of a local address and a global address. The original concept of PGAS has derived from DSM, but because modern PGAS systems are not cache-coherent, their programming models are different from those of traditional DSM systems.

As a memory model, the most important difference of PGAS from shared memory is that the address space is divided into the global space and the local spaces. The global space can be accessed by all of the processes, but the local spaces provided individually to the processes are hidden from other processes. From this distinction over the memory space, the application programmers of PGAS can explicitly control the memory transfers between the global memory and the local memory and tune their programs to reduce communications.

It is relatively easy for PGAS systems to exploit the performance of RDMA (e.g. [68]). Compared with DSM, because PGAS systems do not have caches of global memory, reading or writing remote memory always incur communications. If the remote address is given in a certain procedure, get/put functions can be implemented via RDMA READ/WRITE operations (Section 2.3.1).

It is not easy for the programmers to handle get/put operations of PGAS because they always need to manage the mapping and timing of synchronizing the global memory and the local memory. This problem is similar to register allocation in compilers, for example. Because modern compilers are capable of automatically mapping the available registers (= local memory) into the variables associated with some memory addresses (= global memory), today's programmers do not care about which register is mapped to a specific address. Although there may be room to improve the performance by manually mapping those registers, few people are interested in such an "unproductive" method. However, because PGAS systems cannot automatically manage the relationship and consistency between local memory and global memory, programmers of PGAS applications need to explicitly control them as they manually allocate registers in assembly languages.

PGAS applications can also exploit the knowledge of "partitioning" of the global space into multiple processes because the local part of the global data can be efficiently accessed by the local process like its local address space. Accesses to the remote data always resort to remote accesses, which dreadfully degrade the performance. This is not a restriction of the PGAS model itself but an implementation property of common PGAS systems.

[1] Some language-based systems (e.g. UPC [66], Chapel [43]) do not require get/put function calls and allow ordinary syntaxes for accessing arrays, but this is simply because global objects (or arrays) are explicitly labeled and determined by compilers as global accesses.

PGAS systems can be further classified into global-view and local-view PGAS systems. In global-view PGAS systems, a global array is indexed by a global index irrelevant to the owner process ID. On the other hand, in local-view PGAS systems, the index for a global array is represented by the pair of an owner process ID and a local index. Local-view PGAS systems provide a similar interface to RMA and they are usually implemented as a simple wrapper to RMA. Because of the simplicity, it is easy to implement a local-view PGAS system efficiently. However, from the perspective of productivity, global-view PGAS is better because local-view PGAS provides a too *hardware-centric* view.

Since the research about PGAS systems is still active in the HPC community compared to DSM, there are a number of examples of PGAS systems and applications, which will be discussed in Section 6.2.

2.3. Interconnection networks

Currently, InfiniBand [89] is the most famous interconnection network for supercomputers. The rest of this dissertation employs InfiniBand both for the description and evaluation, but most of the contents can be applied to other interconnection architectures. Existing low-level communication systems for HPC interconnects are briefly described in Section 6.4.1.

2.3.1. Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a hardware feature of interconnection networks used in supercomputers. Communications via RDMA bypass the kernel and achieve both low latency and high bandwidth. RDMA is supported by most of the modern interconnection network architectures such as InfiniBand, Omni-Path [32], or Tofu [11].

InfiniBand Verbs [127] is the low-level interface for handling InfiniBand devices. InfiniBand Verbs offers these communication functions:

SEND operations
: When the sender process invokes `ibv_post_send()` and the receiver invokes `ibv_post_recv()`, the message between those processes is transferred. SEND operation is a variant of message passing (Section 2.1.1) and similar to `MPI_Send()` and `MPI_Recv()`.

RDMA WRITE operations
: The initiator process solely writes its data on the memory of another process.

RDMA WRITE with Immediate operations
: Not only the initiator process solely writes on the remote memory, but also it inserts the completion on the destination process.

RDMA READ operations
: The initiator process solely reads the data of another process.

RDMA ATOMIC operations
: The initiator process solely does a *read-modify-write* on the remote memory. There are two types of atomics supported: compare-and-swap and fetch-and-add.

All of the four RDMA operations can be issued by the initiator process with `ibv_post_send()` and do not use `ibv_post_recv()`.

RDMA operations have some restrictions compared with local memory operations:

1. *The CPU cores on the remote process cannot get the completion notifications* of the RDMA operations[2]. For RDMA WRITE or ATOMIC, it is still possible for the remote process to poll the memory writes, but it is available only if the remote process knows the destination address and consumes its CPU resources.

[2]Except for RDMA WRITE with Immediate, which is exactly introduced to notify the remote process.

2. *Contiguous memory transfers* are almost mandatory. In InfiniBand, some RDMA operations provide the ability to scatter/gather the accesses for the local buffers, but for the remote buffers, each discontiguous transfer requires a work request, which usually comes with software overhead.
3. The memory buffers *must be registered to the network interface cards (NICs)* beforehand to transfer the data using RDMA. Registered buffers are internally pinned to physical memory so that the address translation of page tables can be fixed and the interconnect hardware can access the memory independently of the operating system's behavior. In InfiniBand Verbs, memory registration is done by the function `ibv_reg_mr()`. It is known that the memory registration is a heavy operation especially for small buffers [131, 71].
4. The basic RDMA operations (WRITE/READ) are cache-coherent according to the CPU's consistency model, but the RDMA ATOMIC operations do not guarantee the atomicity when they are combined with the CPU's atomic instructions [50]. In detail, InfiniBand Verbs defines three levels of atomic support: `IBV_ATOMIC_NONE` (no support), `IBV_ATOMIC_HCA` and `IBV_ATOMIC_GLOB`. The current generation of hardware only supports `IBV_ATOMIC_HCA`, which only guarantees the atomicity of atomic operations inside an HCA. This problem can be solved if `IBV_ATOMIC_GLOB` is supported by NICs.
5. The other restrictions of RDMA ATOMIC are the alignment and the data size of atomic operations. InfiniBand Verbs support only 8-byte ATOMIC operations and the destination address must be 8-byte aligned.

Section 3.4 will describe that the coherence actions of Menps were designed to match these restrictions. The detailed interface of InfiniBand Verbs will be explained in Section 4.2.1. MECOM (Chapter 4) is a communication library designed for exploiting the performance of RDMA in multi-core architectures.

2.3.2. Performance modeling of communication systems

Parallel Random Access Machine (PRAM) [70] is a basic abstraction model of shared-memory computing systems. PRAM is useful to examine the complexity of parallel algorithms, but in practice, PRAM is far from real hardware because it assumes that each processor issues memory access in constant time.

LogP [55] is a parallel machine model to reflect the performance characteristics of real hardware. LogGP [13] extends LogP to capture the performance of large message transfers. LogGP defines the following parameters to represent the communication systems:

- L: Latency (or delay), which is an upper bound of communicating a message from its source to its destination processor.
- o: the overhead, which is defined as the length of time that the source and destination processors need to engage in the transmission or reception of each message.
- g: gap between messages, defined as the minimum time interval between consecutive message transmissions or receptions. The reciprocal of g is the available per-processor bandwidth for short messages, which is also known as a *message rate*.
- G: Gap per byte, defined as the time per byte for long messages. The reciprocal of G is the available per-processor *bandwidth* for long messages.
- P: the number of Processes, which is the number of ranks in MPI.

These parameters are used in Chapter 4 to conduct the microbenchmarks on InfiniBand. MECOM is focused on improving message rates of inter-node communications in multi-threaded systems.

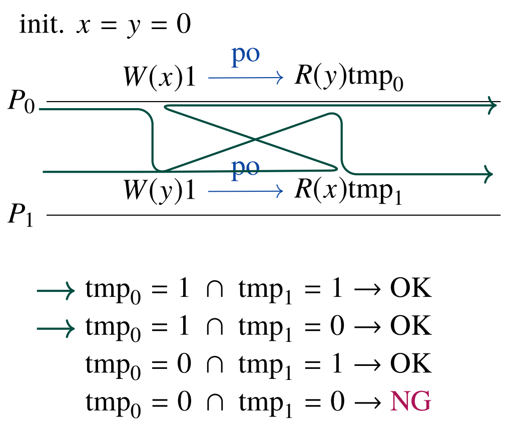

init. $x = y = 0$
P_0
$W(x)1 \xrightarrow{\text{po}} R(y)\text{tmp}_0$
P_1
$W(y)1 \xrightarrow{\text{po}} R(x)\text{tmp}_1$
$\text{tmp}_0 = 1 \cap \text{tmp}_1 = 1 \rightarrow \text{OK}$
$\text{tmp}_0 = 1 \cap \text{tmp}_1 = 0 \rightarrow \text{OK}$
$\text{tmp}_0 = 0 \cap \text{tmp}_1 = 1 \rightarrow \text{OK}$
$\text{tmp}_0 = 0 \cap \text{tmp}_1 = 0 \rightarrow \text{NG}$

Figure 2.2.: Example of global ordering in sequential consistency.

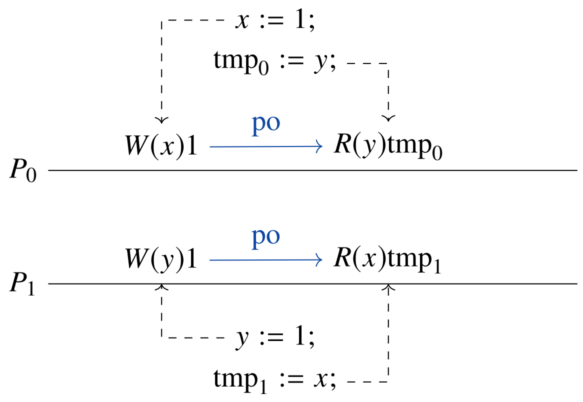

$x := 1;$
$\text{tmp}_0 := y;$
P_0
$W(x)1 \xrightarrow{\text{po}} R(y)\text{tmp}_0$
P_1
$W(y)1 \xrightarrow{\text{po}} R(x)\text{tmp}_1$
$y := 1;$
$\text{tmp}_1 := x;$

Figure 2.3.: Example of a program order in sequential consistency.

2.4. Memory consistency models

2.4.1. Sequential Consistency (SC)

Memory consistency models [164] are rules for shared-memory programs that define which value of a write is visible to another core's read. There is a common consensus that *Sequential Consistency (SC)* [116] is the most strict memory consistency model for parallel computers. The original definition of SC is described as the following two constraints:

1. The result of any execution is the same as if the operations of all the processors were executed in some sequential order.
2. The operations of each individual processor appear in this sequence in the order specified by its program.

It is known that the constraints of SC can be translated into the following two terms [8]:

1. All of the memory accesses follow one *global total order* which may vary on each execution (Figure 2.2).
2. All of the memory accesses also follow the *program order* which is specified by the programmer beforehand (Figure 2.3).

SC is an intuitive model for programmers because they can reason as if every memory access is sequentially executed on a single memory system. However, the optimization opportunity for the underlying memory

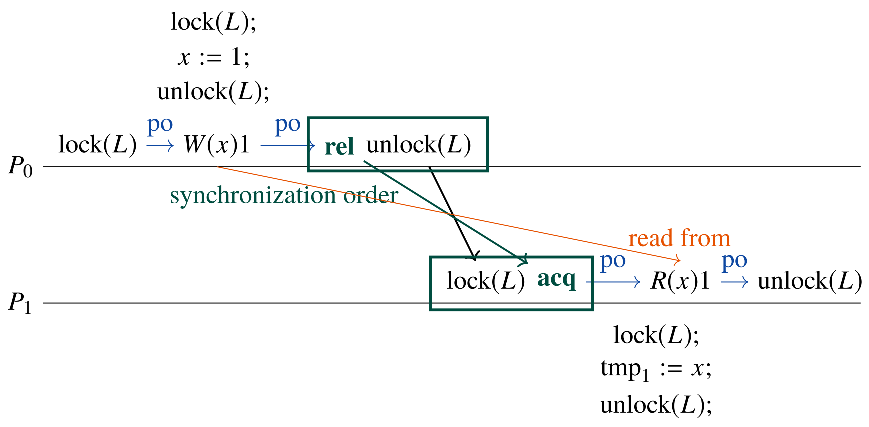

lock(L);
$x := 1$;
unlock(L);
P_0
lock(L) $\xrightarrow{\text{po}}$ $W(x)1$ $\xrightarrow{\text{po}}$ **rel** unlock(L)
synchronization order
read from
P_1
lock(L) **acq** $\xrightarrow{\text{po}}$ $R(x)1$ $\xrightarrow{\text{po}}$ unlock(L)
lock(L);
$\text{tmp}_1 := x$;
unlock(L);

Figure 2.4.: Example of a release-acquire synchronization pair based on the mutex synchronization.

system is restricted because it cannot reorder memory accesses. Therefore, DSM designers have extensively studied the relaxation of memory models to avoid high communication costs in distributed memory.

2.4.2. Data-race-free consistency models

Because DSM systems are executed on fully distributed systems where the higher communication costs are observed than multi-core systems, memory reordering to hide the network latency is important. There are many examples of relaxed memory consistency models to achieve better performance in DSM systems: Release Consistency [74, 102], Entry Consistency [28], and Scope Consistency [87].

Adve et al. formalized a uniform memory model called *data-race-free-1* [8], which unifies the important models such as Release Consistency. The essence of this model is that *the behavior of application programs without data races is equal to the behavior as if it is executed on sequential-consistent memory.* Most of the recent parallel programming languages have converged to this SC-for-DRF model because it can provide both the SC-centric intuitive model and the optimization opportunity (e.g. [175]). Menps also assumes the data-race-freedom of the target applications (Section 3.3).

In DRF-based models, all of the memory accesses are classified into two types:

Synchronized accesses

Memory accesses with synchronization between threads. The examples are mutex lock/unlock and atomic operations.

Unsynchronized accesses

Memory accesses without synchronization. These correspond to ordinary load/store instructions.

Synchronized accesses can be labeled with one of the multiple memory orders such as *release* (e.g. unlocking a mutex) and *acquire* (e.g. locking a mutex). Based on these classifications of memory accesses, happens-before partial order $\xrightarrow{hb}$ is defined as the following two conditions:

1. $\xrightarrow{hb}$ follows the program order.
2. If an acquire access a_2 returns the value written by a release access a_1, then $a_1 \xrightarrow{hb} a_2$ (e.g. Figure 2.4).

In DRF models, all of the results of memory accesses are defined by $\xrightarrow{hb}$ instead of a global total order. In general, there may not be a global total order for memory accesses in relaxed consistency models including RC.

In relaxed consistency models, a *data race* is defined for a pair of two unsynchronized memory accesses that satisfy all of these three conditions:

1. All of the accesses are performed on the same memory location.
2. At least one of them is a write.
3. They are not ordered by $\xrightarrow{hb}$. In other words, they are *concurrent*.

The programs which incur data races are racy programs, which are considered illegal in most of the relaxed consistency models. For example, in the C++11 memory model [27], if a program contains a data race, the execution result becomes an *undefined behavior*, making the output results totally undefined as the language. The general practice to avoid data races is to adopt synchronization primitives such as mutexes or atomic operations.

Although the relaxation of memory consistency models is not in the main scope of this dissertation, there are many DSM studies that focus on how relaxed consistency models perform better than more strict ones. The examples of other relaxed consistency models will be partly described in Section 6.1.3.

2.5. Mutex implementations

A mutex (mutual exclusion) [82] is a key component of multi-processor programming. In particular, for shared-memory systems, the performance of mutexes significantly affects the overall performance because the shared data is normally synchronized via mutexes.

A test-and-set lock (TAS) lock is the most simple implementation of spinlocks. A TAS lock is implemented using atomic operations to set and reset a bit. The storage cost is one bit or usually a byte for convenience. A ticket lock [128] is used to improve the fairness of spinlocks.

It is well known that simple spinlock implementations suffer from poor scalability when highly contended [39]. To improve the performance of spinlocks, queue-based locks [82], which chain the waiting cores in a list, have been proposed. MCS locks [129] is a famous example of queue-based locking. Another famous example of queue-based locking is CLH locks [54]. It is also important that queue-based locks require a temporary space that is guaranteed to exist from the lock acquisition to the release.

It is commonly known that there are implementations of mutexes without atomic operations such as Lamport's bakery algorithm [114, 82]. However, such algorithms require memory barriers for synchronization, which degrades the performance as in atomic operations.

Some advanced implementations of mutual exclusion will be described in Chapter 6.

2.5.1. Characteristics of mutexes

There are multiple aspects which affect the performance of mutexes (see also [105, 77]):

Uncontended performance
The latency to lock/unlock a mutex when there are no other competing threads.

Contended performance ($\approx$ scalability)
The latency to lock/unlock a mutex when there are multiple threads competing to lock the same mutex.

Fairness
Locks with fairness guarantee that there is an upper limit to obtain a lock for every lock acquirer.

NUMA-awareness
The strong FIFO ordering becomes the bottleneck in NUMA systems. NUMA-aware locking schemes employ hierarchical methods (hierarchical mutexes) for the scalability on NUMA systems.

Lock delegation
Some sorts of critical sections can be executed in a different thread. Lock delegation is also known as software offloading, detached execution [148], and software combining. There are mainly two benefits of detached execution of critical sections:

1. Cache locality is improved because the same thread can execute the critical sections accessing the same data.
2. A thread that requires critical sections can return to its execution immediately after the queue operation.

Storage

For example, in standard queue locks (e.g. MCS, CLH), the storage cost is $O(1)$ per lock ignoring the temporary space for queue entries. Some methods including Array-Based Queuing Lock (ABQL) [82] require $O(n_{\text{threads}})$ per lock where n_{threads} is the number of threads that may access the mutex.

2.6. Thread scheduling

2.6.1. Kernel-level threading

On shared-memory systems, programs can create threads to express the parallelism in their programs. A scheduler is a component that controls the execution of threads, which affects the performance of multi-threaded programs.

In modern operating systems, threading is normally implemented at the kernel level (kernel-level threading). For example, Pthreads (or POSIX Threads) [185] is a standard interface supported by the Linux kernel or other UNIX-like systems. Strictly speaking, Pthreads itself is a threading API which does not correspond to any specific implementation. However, because the current Linux-based systems implement Pthreads at the kernel level, we usually assume that Pthreads is an interface for kernel-level threading.

The most basic feature of threading libraries is the form of fork-join parallelism: the pair of *fork* and *join*. Fork creates a new thread and join waits for the completion of another thread which is created via fork. The full set of threading features includes other features such as *exit*, *yield*, *detach*, and thread-local storage (TLS). Threading libraries also include synchronization mechanisms such as mutexes and condition variables.

2.6.2. User-level threading and task parallelism

User-level threading (ULT) is a technique to manage a thread of execution in user space rather than in kernel space. User-level threading is based on lightweight context switching in user space and thus provides fast thread creation and destruction. There are a number of examples of user-level thread libraries including MassiveThreads [134]. Other user-level threading systems are discussed in Section 6.5.

Most of the user-level threading systems support a similar set of functions to that of Pthreads. The primitive functions such as *fork*, *join*, *exit*, and *yield* can be implemented in user space. Those systems usually implement cooperative scheduling (also known as non-preemptive scheduling), that does not force the threads to suspend during execution but encourage them to yield the processors.

User-level threading libraries can schedule (user-level) threads efficiently because they typically use the standard scheduling strategy *work stealing* [72] for load balancing. Work-stealing schedulers are also known as *task-parallel systems* because an application program is divided into small chunks of codes (= tasks).

In distributed-memory programming, there are two different aspects in which task-parallel libraries are beneficial:

1. To hide the latency of communications between nodes in distributed-memory environments, user-level threads (or coroutines) are useful because they can be switched to another with a minimal cost when the communication is ongoing. The basic concept of MECOM (Chapter 4) is to employ user-level threading to hide the communication latency on distributed-memory systems.
2. Application computations can be shared or migrated using user-level threading because it enables to control call stacks on the memory in a flexible manner. Menps includes a simple OpenMP runtime system that transparently shares the call stacks of OpenMP programs on top of software DSM

(Section 3.6.3). User-level threading can be utilized to implement tasking on distributed-memory systems, which is also known as distributed work stealing (see also Section 6.5.3).

2.6.3. OpenMP

OpenMP [5, 45] is a standard interface for writing shared-memory programs. OpenMP is designed as a language extension and requires compiler support, which is provided in most of today's famous compilers such as GCC and Intel Compiler. Programmers insert the OpenMP directives as C pragmas (or Fortran comments) into their programs to parallelize them.

In general, it is easier and simpler to write OpenMP programs than to write multi-threaded programs on Pthreads because OpenMP programs look similar to sequential programs. When OpenMP pragmas are ignored, they can be executed as sequential programs. OpenMP provides an intuitive parallel programming style for programmers, who are used to write sequential programs.

The following OpenMP example borrowed from [45] calculates the matrix-vector product in parallel:

```
void mxv(int m, int n, double* a, double* b, double* c) {
    int i, j;
    #pragma omp parallel for default(none) \
            shared(m,n,a,b,c) private(i,j)
    for (i = 0; i < m; i++) {
        a[i] = 0.0;
        for (j = 0; j < n; j++)
            a[i] += b[i*n+j]*c[j];
    }
}
```

When the program reaches a code fragment annotated with the parallel construct (`omp parallel`), the fragment is executed on multiple threads. In the example, the code block of the outermost loop is a parallel region. Other parts outside parallel regions are executed sequentially as in a sequential program. The loop construct (`omp for`) means that the threads divide the work of the specified for-loop. The example specifies both of the constructs in a single pragma.

The basic execution model of OpenMP is SPMD-style, but the specification of OpenMP has been extended to other programming models such as task-parallel programming.

The DSM system Menps proposed in this dissertation is evaluated through a simple OpenMP layer `meomp` as described in Section 3.6.

3. MENPS: A decentralized distributed shared memory library designed for modern interconnects

3.1. Introduction

In the history of parallel computing, there were a number of efforts to simplify programming on distributed-memory systems. An example of such an effort is the development of Partitioned Global Address Space (PGAS) systems, where global address space is separate from the local address spaces. While PGAS systems provide global-view memory models, their programming models are still different from shared-memory systems and the application developers need to modify a large portion of sequential or shared-memory application codes. In this paper, we revisit the method called distributed shared memory (DSM) [56], where transparent coherent address space is provided on distributed-memory architectures.

Although the DSM model provides better application productivity than message passing or PGAS models, DSM is difficult to scale due to coherent caches. There is a fundamental research question whether it is possible to build a scalable coherent cache mechanism in distributed systems. As a counterargument, for example, Martin et al. [124] pointed out the possibility of scalable hardware coherent caches. We also believe that the scalability problems of shared-memory systems are coming from their implementation issues rather than the model itself. There are several inherent problems in coherent caches including false sharing or cache invalidation traffic, but we developed techniques to mitigate these problems.

From the perspective of hardware technology trends, in the 1990s, the developers of DSM systems assumed the computing environments connected by interconnection networks with long latencies and limited bandwidth. The network speed has largely improved in the latest interconnects, however, the latency between computing nodes is still an order of magnitude longer than that of the memory latency and degrades the performance of DSM systems. As argued by Kaxiras et al. [100], we think it important to utilize RDMA to solve the latency issue of DSM. Although the one-sided communication model seems to be close to the behavior of DSM systems, exploiting RDMA for DSM necessitates different methods from earlier systems, which heavily depended on remote invocations of message handlers. The communication in PGAS can be naturally mapped to RDMA, but it is not in DSM systems due to coherence protocols.

We have tackled several fundamental problems to scale DSM systems. The first problem is false sharing. It is well-known that the negative effects of false sharing can be mitigated by adopting multiple-writer protocols, in which a single cache block can be simultaneously written by multiple cores. To implement a multiple-writer protocol, it is common to aggregate diffs to a home process (home-based protocols). Because the home assignment is an important issue to shrink the write latency, several systems implemented dynamic home migration schemes. We observe that existing methods for home management require message handlers for handling coherence protocols and are therefore not suitable for RDMA. Based on this observation, we devised a method that migrates home solely with RDMA.

The second issue is the use of directories to maintain coherence. Directory-based coherence is the standard method for cache invalidation in large-scale systems, but cannot easily scale the traffic for sending invalidation messages. There are a number of variations in improving directory-based coherence, but few of them are decentralized. We focused on recent multi-core studies including logical timestamp-based coherence [198, 199] as an alternative to directory-based coherence. Timestamp-based coherence protocols do not need to track sharers with directories but may lead to unnecessary invalidations. We propose an idea to combine both timestamp-based coherence and write notices to solve those problems.

Based on these observations and ideas, we present the design and implementation of our DSM library Menps. Menps can execute a subset of shared-memory OpenMP [5] programs transparently with little or no modifications. To evaluate our library, we picked up five applications from the NAS Parallel Benchmarks [20] which could be executed. Menps can accelerate some OpenMP programs using multiple nodes compared to a normal OpenMP runtime running on a single node. We also show better application performance of Menps than that of a DSM library Argo [100], which is tuned for modern hardware.

Here are the contributions of this chapter:

- We have implemented a decentralized DSM library Menps which can transparently execute OpenMP programs on distributed-memory machines with minimal code modifications of the applications. Menps does not require any special compiler or code transformation. We have developed a scheme to share the call stacks of an application program running on the DSM so that automatic variables of the application code can be globally shared in addition to global variables and heap memory.
- A migration scheme called *floating home-based protocol* in DSM systems suitable for RDMA is introduced (Section 3.4.2). To eliminate the use of message handlers, the proposed migration scheme greedily moves the data so that the diff merge is locally done by the writer process. For better scalability, the idea of queue locks is incorporated into the migration scheme.
- A new invalidation method without cache directories is proposed. Our method combines two existing invalidation techniques: logical leases [198, 199] and write notices [102] (Section 3.4.5). It keeps the benefits of logical leases while it mitigates the latency derived from the over-invalidation of logical leases. Logical timestamps enable us to manage both cache block leases and write notices uniformly. Write notice invalidations can effectively transfer the memory using a single RDMA READ operation in the best case, and timestamp-based invalidations can eliminate the use of broadcasting or centralized data structures. To our knowledge, this is the first attempt to implement an invalidation scheme on software DSM in a decentralized fashion.
- Based on both the floating home-based method and the hybrid cache invalidation, the method called *fast release* is proposed (Section 3.4.6). The fast release reduces the overhead of fences when the home nodes of memory blocks are not migrating to other nodes.
- The experimental performance of Menps is demonstrated (Section 3.7). The experimental results have shown the advantages of our proposed protocols. Menps has generally outperformed Argo for NAS EP and CG [20].

3.2. Importance of RDMA-based design for DSM systems

Figure 3.1 shows the comparison of three methods for remote write. We have made a microbenchmark to measure the execution time of diff merging to show the importance of designing the coherence protocols specifically for RDMA. In this benchmark, the following three methods are compared:

ContiguousWrite
: Write the whole buffer via RMA. The transfer size of RMA does not depend on the written size. This method cannot be directly applied to process multiple-writer cases of DSM systems.

DiscontigousWrite
: Write the modified data per byte *separately* via RMA (to deal with multiple-writer cases). DiscontigousWrite simulates the implementation method of existing RDMA-based DSM systems including Argo DSM [100] and NEWGENDSM [141, 143].

PackDiff
: Pack the modified bytes into a message and unpacks it on the receiver side. PackDiff simulates the method of traditional DSM systems (e.g. TreadMarks [102, 103]).

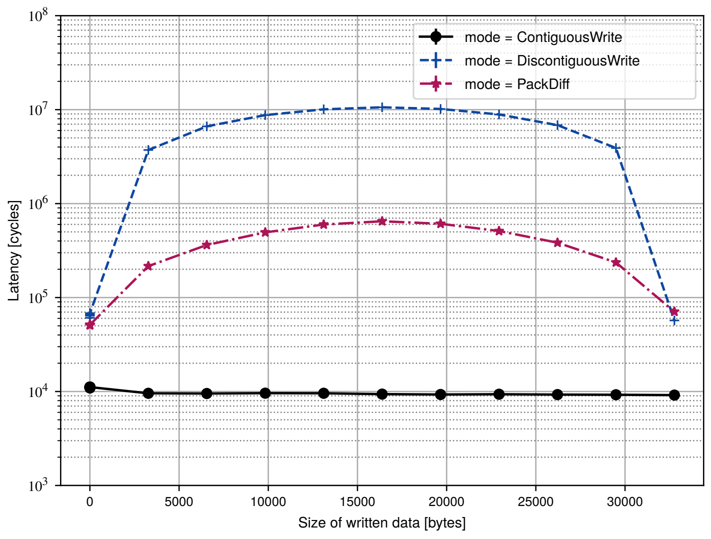


Figure 3.1.: Performance comparison of three methods for remote write. The cache block size is fixed to 32 KiB. The written bytes are randomly selected. This benchmark program used MECOM (Chapter 4) and MPI RMA. Y-axis (the latency) is logarithmic.

It is observed both DiscontigousWrite and PackDiff has much more overhead than ContiguousWrite. This is because they always scan the whole memory block to detect the writes. When a memory block is not written ($x = 0\,\mathrm{B}$) or completely changed ($x = 32\,\mathrm{KiB}$), DiscontiguousWrite and PackDiff have similar performance characteristics because both the number of messages and their sizes are almost the same. However, when a block is moderately written (e.g. $x = 16\,\mathrm{KiB}$), the performance of DiscontiguousWrite is highly degraded because of the overhead of discontiguous one-sided transfers. The performance of PackDiff is also degraded because it has overhead in encoding and decoding the diffs in a discontiguous way.

This result gives several interesting observations. Translating the conventional communications into RDMA (DiscontigousWrite) does not provide better performance for diff processing. Message passing (PackDiff) may perform faster than such a design that simply uses RDMA (DiscontigousWrite). More importantly, it is necessary to design the protocols like ContiguousWrite to exploit the advantage of RDMA. *RDMA-based DSM systems should neither transfer the memory in a fine-grained manner nor scan the memory pages for write detection using CPUs.* Thus, in reality, the overall design of RDMA-based coherence becomes very different from conventional methods.

Figure 3.2 shows the comparison of RDMA WRITE latency between the old interconnect VIA and the current interconnect InfiniBand FDR. We used Figure 3 and Figure 4 of [92] to plot the latency of VIA. The latency of InfiniBand is measured using MECOM 2 with UCT. From this figure, it is first observed that the minimum latency of InfiniBand is about 1/18 of that of VIA. The more important observation is that the buffer size to reach the bandwidth limit is increased from about 1 – 10 KB (in VIA) to about 10 – 100 KB (in InfiniBand). This result indicates that it is becoming better to transfer a buffer (or a cache block) in a coarse-grained manner rather than reducing the bandwidth using complex software techniques.

3.3. Overview of Menps

As shown in Figure 1.1, Menps consists of three main components: the core `medsm2`, the communication layer `mecom2`, and the OpenMP runtime `meomp`. The DSM core component `medsm2` implements the cache coherence with the protocols proposed in Section 3.4. The communication layer `mecom2` is later described in Chapter 4. The communication layer is based on user-level threading (Section 2.6.2) and provides a unified interface to both MPI and UCX [172]. The DSM core component `medsm2` can be executed either on MPI-3

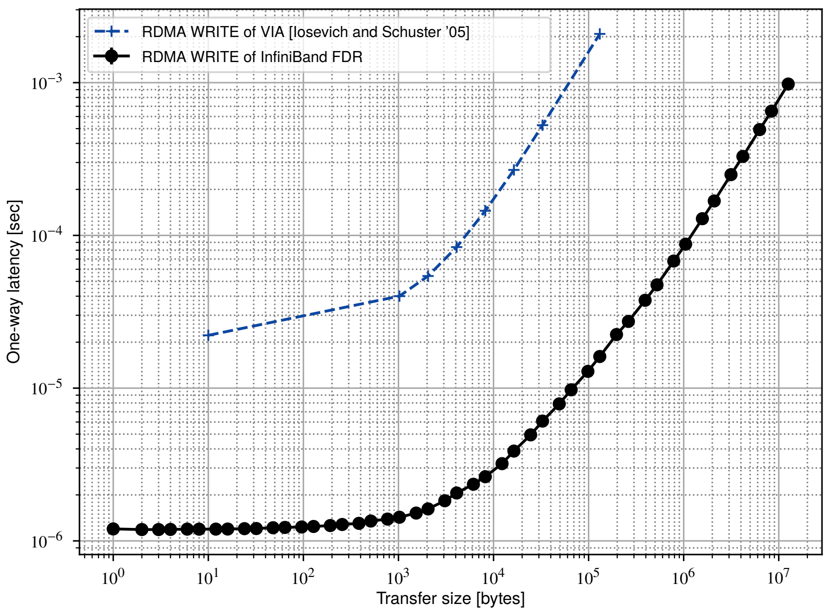


Figure 3.2.: Comparison of latency of RDMA WRITE between VIA [92] and InfiniBand FDR.

RMA or UCT (a low-level component of UCX). MPI collective and point-to-point communications are used only for minor purposes such as all-to-all communications in an OpenMP barrier and lock notification. The OpenMP runtime component `meomp` traps the internal application binary interface (ABI) calls of the existing OpenMP runtime systems and transforms them into the DSM functionalities.

Menps is based on the data-race-free (DRF) [9] assumption (described in Section 2.4.2) on the input OpenMP applications and provides the SC-for-DRF consistency, which is adopted in modern languages such as C++11 and Java.

The launcher of Menps starts as a regular MPI program, sets up the environment, and executes a single shared-memory application using multiple processes. Menps supposes that each computing node may run multiple processes, each of which may run multiple threads in general.

Menps divides the global space into separated memory regions (called *segments*) and each segment consists of multiple contiguous cache blocks. Menps allows users to specify the block size of each segment individually to adapt their different purposes. Menps does not share global variables by default because the underlying system hides many global variables that cannot be shared between processes. Menps manages only global variables with which a special attribute is specified. Menps also provides `malloc` for shared space. Only the call stacks of applications are shared by default.

3.4. Coherence actions of Menps

Although the basic interface of shared-memory systems is as simple as multiple threads can read and write the same address space, there are many design options in DSM systems because they need to handle multiple issues in memory and network systems. In this section, we describe the fundamental problems in implementing shared-memory systems and try to illustrate how they can be avoided in real implementations.

3.4.1. Existing methods for writer management

Shared-memory systems require a method to control memory access privileges of the application program in order to differentiate cache hits and misses. To do so, we have implemented Menps as a page-based software

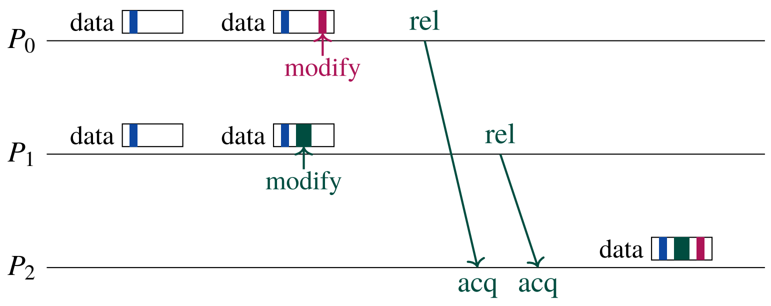


Figure 3.3.: A false sharing situation in release consistency. Because both the preceding processes P_0 and P_1 wrote different words of the same cache block, P_2 needs to observe the writes of both.

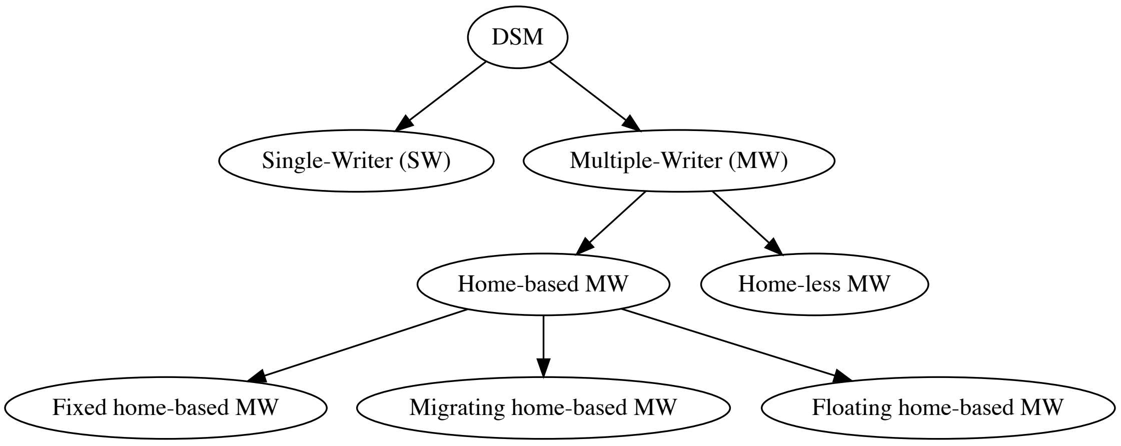


Figure 3.4.: Classification of single-writer and multiple-writer protocols.

DSM system[1] [119]. That is, the validity of a cache is checked by the memory protection mechanism provided by both the operating system and the underlying hardware. Invalid caches are protected (via `mprotect()`) and accessing those caches invokes a replaced segmentation fault handler to resolve the cache miss. Section 6.1.4 further discusses this topic while describing the existing work.

In real shared-memory systems, multiple words may reside on the same cache block. False sharing is a phenomenon in which multiple cores are simultaneously writing on different words of the same cache block. Figure 3.3 shows an example of false sharing in release-consistent memory. In general, false sharing is not a good practice in shared-memory programming, but it is necessary for the systems to support such situations in order to execute shared-memory programs transparently. To deal with false sharing in Menps, the floating home-based method is introduced for writer management.

Figure 3.4 shows the classification of the methods for writer management in DSM systems. Based on this diagram, which also classifies the proposed floating home-based method, the following section explains why we reached the floating home-based method to design Menps.

The simplest way to manage multiple writes on the same block is to prohibit the writable state except for a single core at a certain time. This method is called a *Single-Writer (SW)* protocol, which is employed by most of today's multi-core processors. Figure 3.5 shows an example of false sharing in single-writer protocols.

[1]Page-based DSM is also known as shared virtual memory (SVM) because it can be seen as an extension of virtual memory mechanisms from swap devices into remote memory.

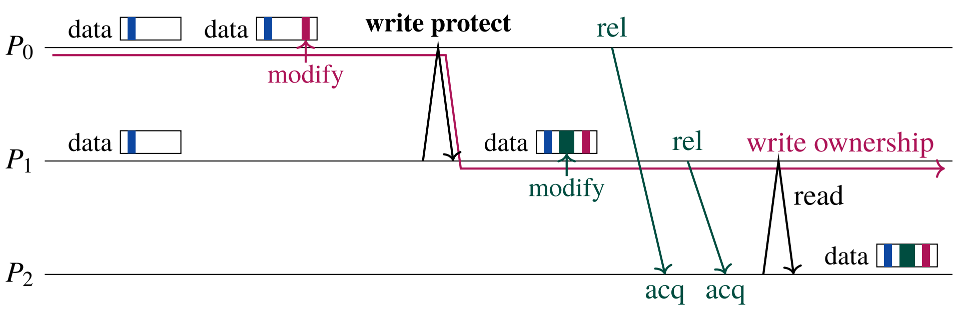


Figure 3.5.: A false sharing situation in single-writer protocols.

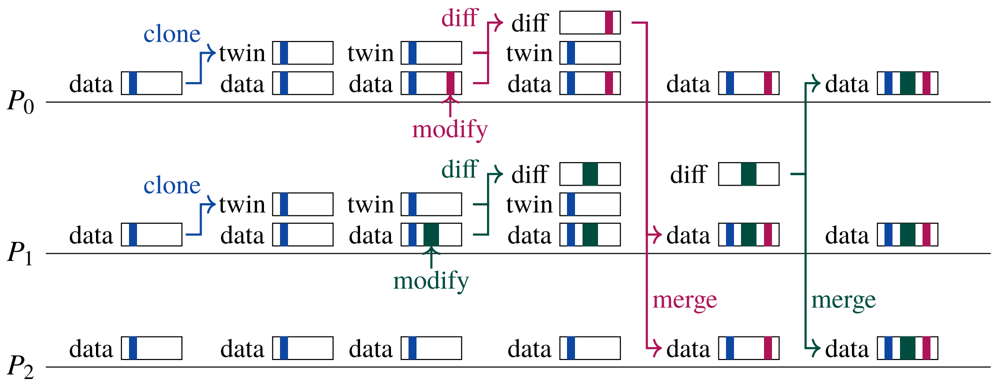


Figure 3.6.: A false sharing situation in multiple-writer protocols. Both P_0 and P_1 can concurrently writing on the same cache block and they distribute their diffs to others.

They allow the write ownership of each cache block only for one process at the same time. If another process without the ownership needs to write the same cache block, it is necessary to prohibit the writes from the previous writer so that the ownership can be transferred.

The main benefit of SW protocols is their simplicity, but their performance is highly degraded in false sharing. Due to the cache granularity restriction of page-based DSM, the frequency of false sharing becomes higher than hardware shared memory or other software techniques (see Section 6.1.4). Furthermore, it is almost impossible to implement practical SW protocols purely on RDMA because restricting the writes on the remote processes requires `mprotect()` system calls which are normally accomplished by message handlers.

To mitigate these drawbacks of SW protocols, the DSM researchers have developed Multiple-Writer (MW) protocols which can control writes concurrently as shown in Figure 3.6. When causing false sharing, the writes from different processes must be merged as a single cache block. A standard method to implement merging is called twinning [41], which preserves a copy (= a *twin*) of the cache data before modifications and generates a *diff* by comparing the copy and the modified data. The main advantage of multiple-writer protocols is the reduction of the latency to start writing because a writer process can immediately proceed after the store. The disadvantages are the costs to manage twins and diffs.

Multiple-writer protocols are further classified into two types: homeless [102, 103] and home-based protocols [204]. In homeless protocols, readers of a cache block collect those diffs for the block and apply to their caches by themselves. On the other hand, in home-based protocols, writers accumulate diffs to a "home" process, which is specified for each cache block. Figure 3.7 shows how home-based DSM systems work. Although the variables x and y are sharing the same cache block, they can be written in parallel and merged later in a home process P_3.

Homeless protocols have these well-known problems [204]:

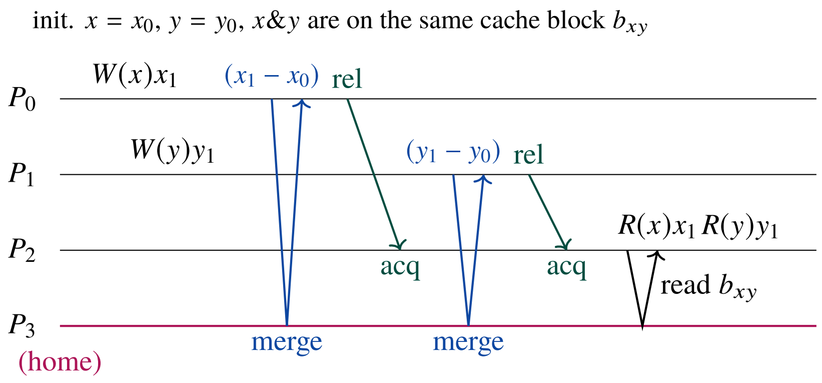


Figure 3.7.: A false sharing situation in home-based multiple-writer protocols. W$(x)x_1$ means writing x_1 to a variable x and R$(x)x_1$ is a read from x resulting x_1. A green edge represents a happens-before relation between a release (rel) and an acquire (acq). Both the processes P_0 and P_1 are writing on two different variables (x, y) on the same cache block.

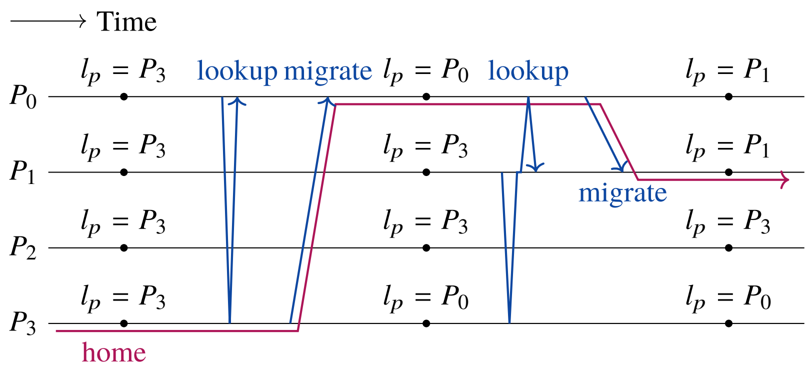


Figure 3.8.: Home migration using probable owners. l_P is the value of the probable owner link to another process. When the home process of a cache block migrates, the links of the source and destination processes are updated.

1. Because diffs on a writer must be maintained until all of the other readers apply them, the storage cost for diffs is expensive. To avoid increasing diffs infinitely, TreadMarks used a global garbage collection mechanism but this mechanism complicated the design and incurs poor scalability.
2. The same diff is applied individually in multiple reader processes and causes the duplicated calculations. In home-based approaches, the application of the same diff only happens once in the whole system.

One of the problems of home-based protocols is how to assign the home process for each cache block. A naive method is to statically assign the home using the block ID (Fixed home-based MW in Figure 3.4), but if the home process is different from actual writers, excessive communications are required to transfer the diffs to the home process in every write.

To solve the problems of fixing the home, there are some studies that argued the migration techniques [52] in home-based DSMs. If the home process can be dynamically changed during the execution, there is a coherence problem of the home process information. If the data is migrated from its old home to another, all of the other processes also need to be informed of its migration. Both broadcasting and directory-based coherence can be applied to refresh the old cached information of home processes. There is another method called *probable owner* [120], in which each process maintains a link to a possible home process at a certain time. Figure 3.8 shows how the probable owner method works. Each process has a link for each block

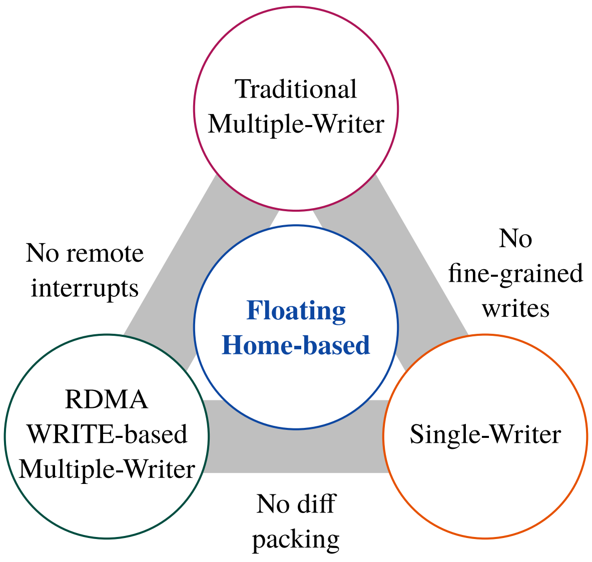


Figure 3.9.: Trilemma of RDMA-based DSM coherence protocols.

pointing to another possible owner and updates it only when the process noticed. If the link is stale and not pointing to the latest home, it is still possible to find the latest home by traversing the distributed links.

Because the probable owner does not require any data structures fixed to a certain process, it can accomplish distributed home migration. However, in the worst case, the length of the linked list becomes the number of processes and it incurs an additional latency to find the home process. Furthermore, if multiple processes are continuously migrating the same block with each other, it may cause a livelock and other processes may not reach the latest home.

Figure 3.9 shows the trilemma of RDMA-based DSM coherence protocols. Traditional Multiple-Writer protocols (e.g. the protocol of TreadMarks [102, 103]) suffer from the overhead of diff packing (PackDiff in Figure 3.1), which does not match the zero-copy semantics of RDMA. RDMA WRITE-based Multiple-Writer protocols (e.g. the protocol of Argo [100]) can be purely implemented via RDMA without diff packing, but they tend to issue fine-grained write messages which cannot be processed as RDMA WRITEs efficiently (DiscontiguousWrite in Figure 3.1). Finally, Single-Writer protocols (e.g. the protocol of MAGI [84]) are free from both diff packing and fine-grained writes and may work efficiently for cache blocks without false sharing, but they need to handle interrupts to stop writing on the previous writer, which cannot be implemented via RDMA. Other existing RDMA-based DSM systems are described in Section 6.1.2, but we think that none of those systems solved this trilemma to exploit the performance of RDMA for accelerating coherence protocols.

3.4.2. Floating home-based protocol for writer management

Our floating home-based approach is different from ordinary home-based protocols because ours “always migrates” the home process of the cache block on each release using probable owners. Typical home-based protocols send diffs to the home process for merging, but instead, our protocol migrates the data immediately and merges the diff on the last writer process. Figure 3.10 shows how our method floats the home. Our floating approach has several advantages compared to existing home-based methods:

1. The home process is automatically assigned to the last releaser. Assuming temporal locality of the writes of the application, the same process is expected to write again on the same cache block and thus it can complete the next write with smaller latency. This property is useful to get the performance benefit of the fast release described in Section 3.4.6.

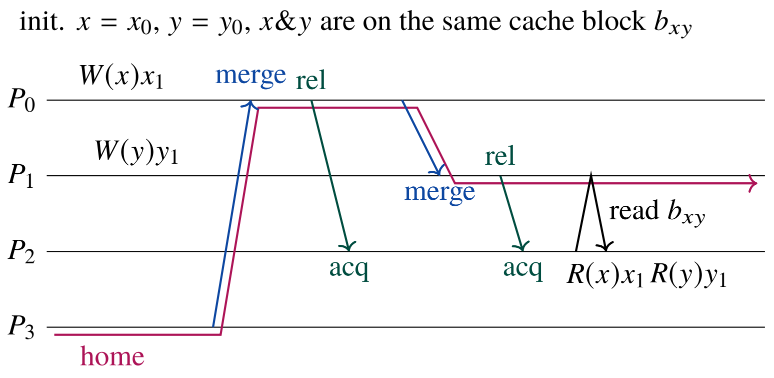


Figure 3.10.: An example of floating home-based DSM. When a process does a release synchronization, all of the memory blocks written by the releaser are merged. For the remotely owned blocks, they are migrated from the previous releasers first.

2. If the home process is remote, in ordinary home-based approaches, merging writes on the same block requires multiple write operations to the remote memory and tends to incur many small (RDMA WRITE) messages as shown in Section 3.2. It is inefficient to send such fine-grained messages in RDMA. Traditional DSM systems used message handlers to expand compressed diffs on a remote process, but such a method cannot be applied to RDMA-based implementations.

The disadvantage of our method is the necessity of write serialization because it is impossible to implement data migration that can be executed in parallel. However, because the hardware can also execute RDMA atomic operations without CPU involvement, the serialization process itself can be accelerated by RDMA. We note here that this approach is different from single-writer protocols that restrict the writable state only for a single writer. Our method only serializes the merge operations of those writes, and the latency of merging is possibly hidden by the system.

Merging a memory block is implemented using a global critical section regarding the block. Although this prohibits a concurrent execution of release operations in multiple processes, it simplifies the migration mechanism and enables the coarse-grained data synchronization similar to ContiguousWrite in Section 3.2. In summary, we found that introducing a lock for each block simplifies the implementation of these features:

1. Home migration. Because home-based protocols need to assign a home process for each cache block, it is necessary to avoid races of the information of home processes.
2. Diff processing (Section 3.5.4). If diff merging may happen in parallel, it is difficult to parallelize merging using SIMD, for example.
3. Timestamp-based invalidation (Section 3.4.5). The mutual exclusion of timestamp states is necessary to force the sharer processes to precisely invalidate old caches.
4. Atomic operations. It becomes easy to implement read-modify-write operations on the DSM space because the modifications on the same cache block from multiple processes can be serialized.

3.4.3. Directory-based cache invalidation methods

Once writes are properly handled in a shared-memory system, another problem is how to manage reads according to the consistency model. There are two sorts of protocols to propagate the written values to the following readers: invalidation-based and update-based protocols. When a cache block is newly written, update-based protocols synchronize the actual data with the readers, but invalidation-based protocols only invalidate those data and the actual data is transferred when the invalidated blocks are accessed later again. In this paper, we only focus on invalidation-based protocols because they can reduce unnecessary data updates in the readers, but the following discussion can also be applied to update-based protocols.

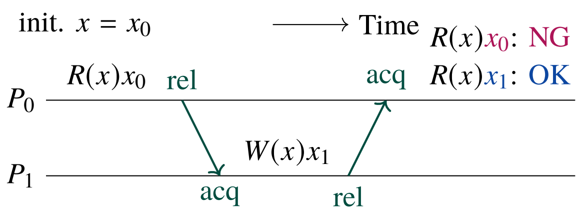


Figure 3.11.: Cache invalidation problem.

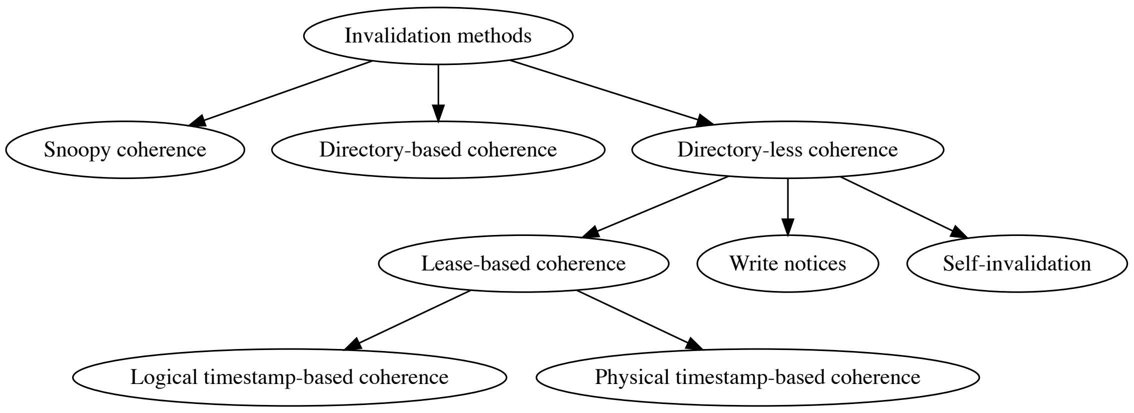


Figure 3.12.: Classification of cache invalidation protocols.

In invalidation-based protocols, it is necessary to calculate which cache blocks must be invalidated according to the consistency model. Figure 3.11 shows the cache invalidation problem. Shared-memory systems need to guarantee that a newly written value which happens-before the read of the old value is always visible to the readers.

There are a number of variants of invalidation-based cache coherence protocols. Figure 3.12 shows the comparison of cache invalidation methods. MENPS employs a hybrid approach of logical lease-based coherence and write notices (Section 3.4.5). This is not a simple combination of two existing methods because the hybrid approach has the synergy for exploiting RDMA and decentralizing invalidation. Before going into the proposed method, this section follows the arrows in Figure 3.12 from the root.

The standard method to invalidate old caches is *directory-based coherence* [118], in which the sharers are tracked in a data structure called a directory. Figure 3.13 shows how directory-based coherence invalidate an

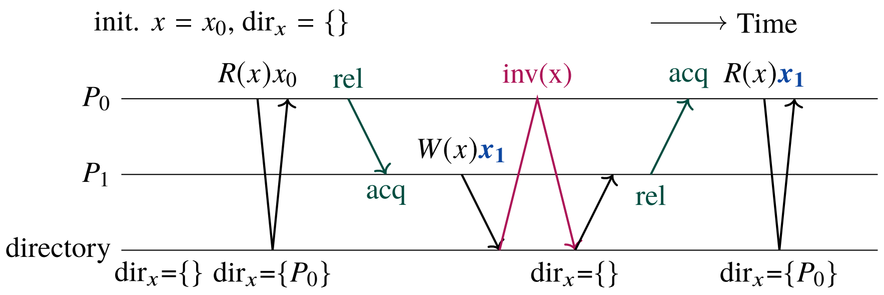


Figure 3.13.: Directory-based cache invalidations. On writing the cache block x at P_1 (= $W(x)x_1$), an invalidation message is sent to P_0 and invalidates the cache copy of x (= x_0).

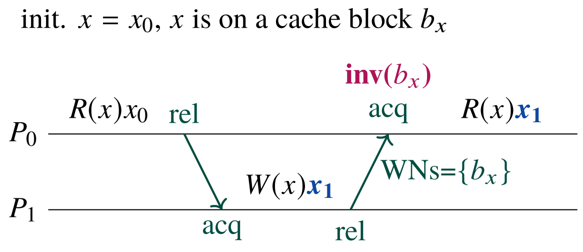


Figure 3.14.: An example of write-notice invalidations. The cache block holding x is written by P_1, and then P_1 sends its invalidation to other processes as a set of cache blocks (in this case, $\{b_x\}$) during the synchronization.

old cache block. In directory-based coherence, when a process fetches a memory block due to the read miss, the ID of the process is recorded to the directory, which holds a set of current sharers for the cache block. Each memory block has its own directory. When the memory block is written after reads, directory-based protocols send invalidation messages (= writer-initiated invalidations) to all of the sharer processes and wait for the acknowledgments from them. Basically, the writer process cannot return to its computation until the arrival of all of the acknowledgments because old caches must not be visible in other processes[2].

Directory-based coherence is more scalable than snoopy methods, which simply broadcast the invalidation messages to all of the cores and suffer from a large amount of invalidation message traffic. Many studies have been made on directory-based coherence both in hardware and software. Recent hardware studies [200, 69, 163] are mainly focused on reducing the storage and energy cost of directories using complex data structures, but because the data structures are more flexible in software DSMs than in hardware, the main problem of directories is rather the increase of invalidation message traffic proportional to the number of reader processes. The more studies about directory-based coherence and other cache invalidation techniques are listed in Section 6.3.

3.4.4. Directory-less cache invalidation methods

Write-notice invalidation, which is one of the *directory-less* invalidation methods, was proposed in a DSM system TreadMarks [102, 103]. Figure 3.14 shows an example of write-notice invalidation. As mentioned, SC-for-DRF requires the write propagation only when a release-acquire synchronization appears. A write notice represents a write (on the cache block b_x in Figure 3.14) of one of preceding releaser processes and can be carried on the synchronization. This method effectively reduces the number of invalidation messages because write notices can be aggregated as a list. However, because the number of write notices continuously increases for possible acquirers, it is necessary to discard those write notices without breaking the coherence. TreadMarks used the global garbage collection not only for diffs but also for write notices, which complicated the overall design.

Timestamp-based coherence (or lease-based coherence) [173, 176, 194] is recently proposed as an alternative to directory-based coherence. The basic idea of this method is that readers register a timestamp representing when all of the cache copies expire for each block. If a writer needs to invalidate the block, it waits for the timestamp expiration instead of direct message transfers toward the readers. The main difference from directory-based protocols is the elimination of sending invalidation messages from writers to readers. The data structure to track sharers is replaced with integers to hold the timestamp values.

A noteworthy example of timestamp-based coherence is Tardis [198, 199, 197], which utilizes *logical* timestamps for ordering. Figure 3.15 shows an example of logical lease-based invalidation. In Tardis, every cache replica has a read timestamp (`rts`) and a write timestamp (`wts`). A reader adds the “lease” value to the read timestamp on the shared cache to get a lease of the cache block. When a new write happens, a

[2]This may be delayed until synchronization points in relaxed-consistent memory.

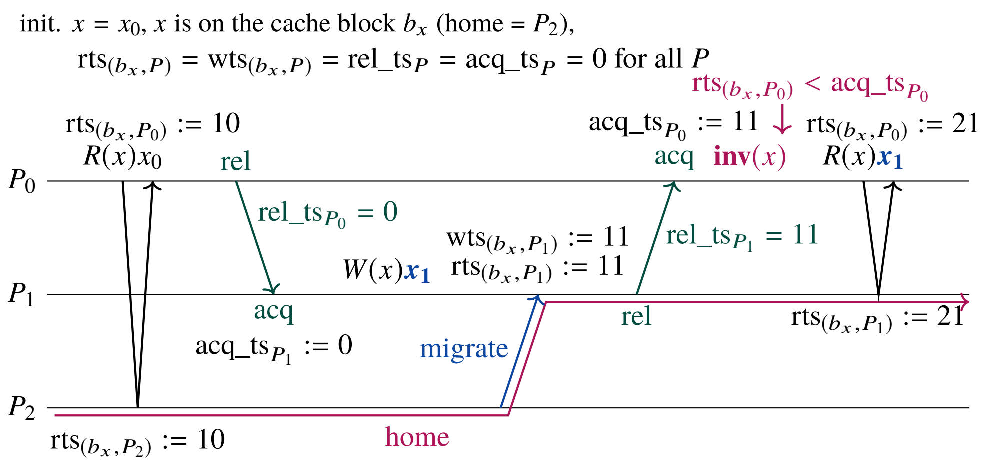

init. $x = x_0$, x is on the cache block b_x (home = P_2),
$\mathrm{rts}_{(b_x,P)} = \mathrm{wts}_{(b_x,P)} = \mathrm{rel\_ts}_P = \mathrm{acq\_ts}_P = 0$ for all P
$\mathrm{rts}_{(b_x,P_0)} < \mathrm{acq\_ts}_{P_0}$
$\mathrm{rts}_{(b_x,P_0)} := 10$
$R(x)x_0$
rel
$\mathrm{acq\_ts}_{P_0} := 11$
$\mathrm{rts}_{(b_x,P_0)} := 21$
acq
$\mathbf{inv}(x)$
$R(x)\boldsymbol{x_1}$
P_0
$\mathrm{rel\_ts}_{P_0} = 0$
$\mathrm{wts}_{(b_x,P_1)} := 11$
$W(x)\boldsymbol{x_1}$
$\mathrm{rts}_{(b_x,P_1)} := 11$
$\mathrm{rel\_ts}_{P_1} = 11$
P_1
acq
$\mathrm{acq\_ts}_{P_1} := 0$
migrate
rel
$\mathrm{rts}_{(b_x,P_1)} := 21$
P_2
$\mathrm{rts}_{(b_x,P_2)} := 10$
home

Figure 3.15.: Example of logical lease-based invalidation. The lease value is set to 10. P_1 only sends the logical timestamp value (rel_ts = 11) in the synchronization with P_0, and then P_0 invalidates x because the read timestamp (= 10) is smaller than acq_ts.

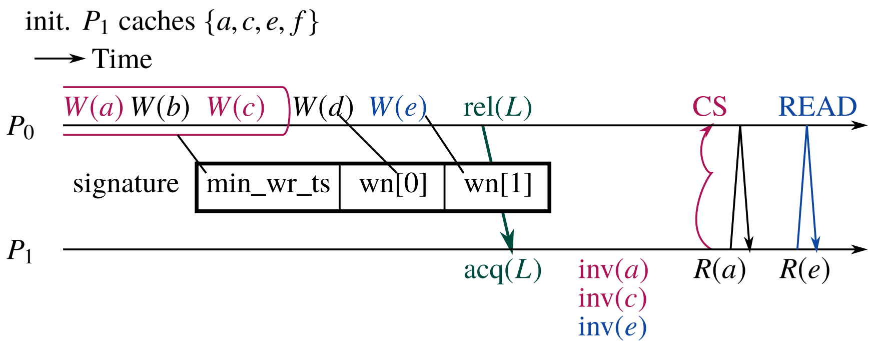


Figure 3.16.: Cache invalidation method of MENPS, a hybrid method of logical lease-based coherence and write notice invalidation.

writer increases the write timestamp to a greater value than the read timestamp. In Tardis, the writer can immediately proceed after the write because the writer can independently increment its logical timestamp. Logical timestamps do not require broadcasting because they are not related to physical time. The main drawback of this method is the increase of unnecessary cache misses because logical timestamps cannot exactly express the happens-before partial order by nature. To mitigate this problem, the updated version of Tardis implemented a method to dynamically select the lease value using prediction.

3.4.5. Cache invalidation method of MENPS

MENPS' approach combines logical lease-based coherence with write notices. Figure 3.16 shows the hybrid invalidation method of MENPS. We have noticed that if a cache block is invalidated with a write notice on a certain process and is read in the same process again, the updated data can be transferred from the releaser which produced the write notice rather than the latest releaser in physical time. This is because SC-for-DRF only guarantees that the preceding writes in a happens-before partial order are visible to the reads. Reading the data from the write notice sender can be efficiently implemented with a single RDMA READ operation because it does not require a lookup of the latest owner. We call this acceleration method "fast read" in this dissertation.

As mentioned, the drawback of write notices is the storage cost which increases as the program continues. The protocol of MENPS solves this problem using logical lease-based coherence. In the proposed protocol,

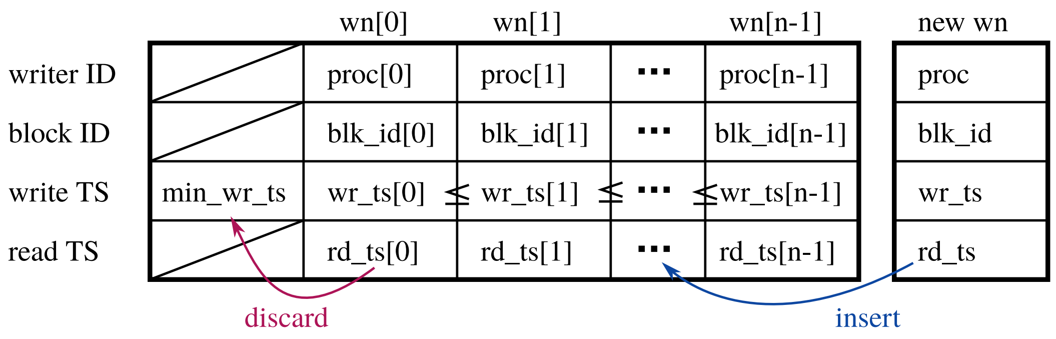


Figure 3.17.: Signature structure of Menps. When a new write notice (new_wn) is inserted, the oldest wn (wn[0]) is removed while taking the maximum of the timestamps.

when synchronization occurs between two processes, the releaser process sends metadata called a *signature* [194] to the acquirer process. The data structure for a signature is shown in Figure 3.17. A signature is constructed of both a (minimum) write timestamp and a list consisting of a limited number of write notices. Each write notice is a tuple of four integers: a writer process ID, a cache block ID, a write timestamp, and a read timestamp.

When a process merges a cache block for synchronization, a new write notice is inserted into the signature. Because the size of the write notice list is limited in our method, the write notices with old write timestamps are discarded if necessary. To maintain the coherence, the largest write timestamp of all of the discarded write notices is extracted and recorded as a minimum write timestamp. When an acquirer process receives a signature from the releaser, it invalidates cache blocks based on both the minimum write timestamp and the write notices. Because the minimum write timestamp represents all of the discarded write notices, our method can remove the write notices without breaking the consistency. It is reasonable to discard the oldest write notices first because of temporal locality.

If the timestamp-based invalidation causes a read miss later, a reader process has no information about which process wrote the block at last in the partial order. In such a case, it enters a critical section as in the diff merging of a release fence in order to update the read timestamp value of the last home process.

3.4.6. Fast release

When the application program enters a DSM barrier operation, all of the processes need to synchronize their data. Because Menps employs an eager protocol (described later in Section 6.1.1), it first completes the writes (= release) of dirty cache blocks so that succeeding processes can read those writes. Then, all of the processes exchange the signatures for cache invalidation and every process invalidates the readable caches based on the signatures.

The performance problem often arises in the former part of a barrier, writing back the dirty caches. *Even when any other processes are not accessing a cache block, a writer process needs to complete releasing its dirty data to pass through the barrier.* Releasing a cache block requires a combination of global lock, merge, and global unlock, which consumes at least thousands of cycles for each cache block.

The two proposed methods for cache coherence, the floating home-based method, and the hybrid cache invalidation method, enabled to invent a new technique named *fast release* to accelerate the release fence. The fast release method can skip a large part of the coherence action of releasing a cache block if the data is not migrating (= the most common case). In detail, the fast release is composed of the two following ideas:

1. Checking whether the owner process is not migrating to another to avoid global locking via RDMA. If the link value is not updated by any other processes, the process can guarantee that other processes did not migrate the data.

```
// The function which "releases" the corresponding block
void release(blk_id_type blk_id) {
    // Update the timestamp values here...

    // Complete the writes.
    std::atomic_thread_fence(std::memory_order_seq_cst);
    // complied as "mfence" on x86-64

    // Load the probable owner.
    if (prob.load(std::memory_order_relaxed) != this_proc) {
        // ... switch to "slow release" (lock/merge/unlock) ...
        return;
    }
    // It is confirmed that the data is not migrated.
    // This process assumes that the release is completed.
}
```

Listing 3.1: Pseudocode of checking the ownership in a release operation.

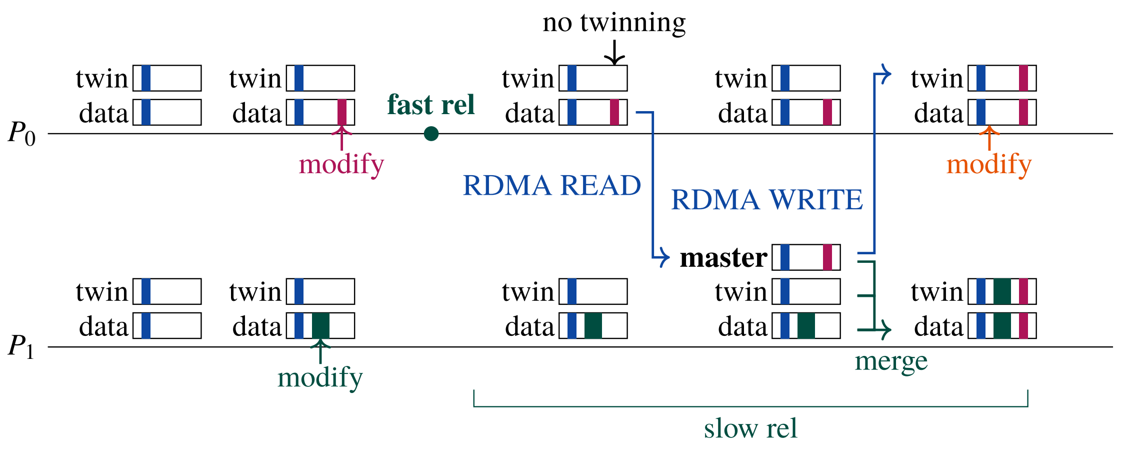


Figure 3.18.: Avoiding twinning in fast release. Even if the fast release avoids twinning before, the next release on another process (= slow release) can compensate it with the pair of RDMA READ and WRITE.

2. "Copying back" the data to the previous releaser process when the block migrates to avoid eager memory copying.

The pseudocode of checking the ownership in a release operation is shown in Listing 3.1. If the value of a probable owner link is not modified by other processes (via RDMA), it is guaranteed that the stores preceding the atomic fence will be visible to the next owner process. Although the interaction of threading and RDMA is not defined in the language standard, RDMA is usually cache-coherent and this scheme should work in most of the existing hardware.

In order to accelerate the release, it is also important to enable the releaser to avoid twinning the dirty data using `memcpy()`. In the baseline method for MENPS, the release operation always copies the dirty data into the twin area, which is necessary to avoid merging the same writes again. However, if the next owner eventually does the same thing (copy to the twin), most of the redundant memory copies would be avoided because the owner is not migrated in most cases. If the data is not migrated, it enables the releaser process to skip memory copy.

The problem to implement this skipping method is that the modifications may not be merged by the releaser process itself because the releaser process may skip merging to shrink the fence latency. To avoid missing the modifications of skipped merging, we choose a method which "copies back" the data to the previous releaser process when the block migrates. Figure 3.18 shows how the fast release method avoids twinning by copying back. In this method, the dirty data of the previous releaser exposed directly to the application program is first transferred via RDMA READ to the next merging process. Because this area is directly exposed to the application, it may be modified concurrently with the DSM system and its data may be different from the twin of the previous releaser. In order to keep track of the diff in the previous releaser, the merging process writes the transferred data back to the releaser's public data using RDMA WRITE. Although this method wastes the network bandwidth because the same data is transferred twice, the latency in the release fence is significantly reduced because it can proceed without copying the memory immediately.

Combined with the two techniques, both memory copy and global locking can be skipped if the memory block is found not to be migrated, which greatly reduces the latency of the release fence in typical cases. To implement this fast release feature, both the floating home-based method and logical lease-based coherence are necessary. Without the floating home-based method, the releaser process cannot skip the global locking in most cases because the home is infrequently migrated to the latest releaser. Because logical lease-based coherence simplifies the coherence problem into a single scalar timestamp, it is easy to implement avoiding global locks in the fast release by writing the timestamps as in Listing 3.1.

3.5. Implementation of Menps

3.5.1. Implementation overview

From the system side of software DSM, the address space of each process is divided into the globally shared space and the local space. The application is accessing only the global space and the local space is only visible to the DSM system or other functionalities (e.g. language standard libraries) which are not aware of the DSM. Menps divides the global space into separated memory regions (called "segments") and each segment consists of multiple contiguous cache blocks. The users can specify the block size of each segment individually to adapt their different purposes.

As mentioned in Section 3.3, global variables with a special attribute[3] can be managed by the DSM. Internally, the attribute allows the DSM to locate them in a DSM segment separately from other regions. Heap allocation from the DSM is currently implemented as a linear allocator and cannot reuse the deallocated memory. We have tried to implement dynamic deallocation for the heap, but this feature was disabled in the later evaluation due to some unsolved bugs.

3.5.2. Core interface of Menps

Listing 3.2 shows the core API of Menps. When the user needs to allocate memory from DSM, it calls `coll_alloc_seg()` to allocate a new segment. The core API provides synchronization operations including barriers and mutexes. Atomic operations were experimentally introduced to implement mutexes, but they are not used in the later evaluation.

3.5.3. Implementing memory fences and barriers

Because most of OpenMP programs frequently use the barrier directive, we have chosen to implement it first. Implementing the barrier in shared-memory systems is more complex than that of message passing (e.g. `MPI_Barrier()`) because a shared-memory barrier not only synchronizes the execution order but also ensures the coherence of the whole memory. To additionally implement fine-grained synchronizations,

[3]For example, GCC has `section` attribute to put the global variables into the specified section. Later, the section address in the executable binary can be controlled by supplying a linker script at link time.

```
class svm_space_base {
public:
  virtual ~svm_space_base() = default;
  // Segment allocation
  virtual void* coll_alloc_seg(
    size_t seg_size, size_t blk_size) = 0;
  virtual void coll_alloc_global_var_seg(
    size_t seg_size, size_t blk_size, void* start_ptr) = 0;

  // Mutexes
  using mutex_id_t = std::uint32_t;
  virtual mutex_id_t allocate_mutex() = 0;
  virtual void deallocate_mutex(mutex_id_t) = 0;
  virtual void lock_mutex(mutex_id_t) = 0;
  virtual void unlock_mutex(mutex_id_t) = 0;

  // Barrier
  virtual void barrier() = 0;

  // Enable/disable pinning
  virtual void pin(void*, size_t) = 0;
  virtual void unpin(void*, size_t) = 0;

  // Enable/disable segmentation fault handler
  virtual void enable_on_this_thread() = 0;
  virtual void disable_on_this_thread() = 0;

  // Atomic operations (experimental)
  // e.g. atomic for uint32_t
  virtual bool compare_exchange_strong_acquire(
    uint32_t& target, uint32_t& expected, uint32_t desired) = 0;
  virtual void store_release(uint32_t*, uint32_t) = 0;
  virtual uint32_t load_acquire(uint32_t*) = 0;
};
```

Listing 3.2: Core API of MENPS.

Table 3.1.: XOR-based diff merge.

		twin (t) / data (d)			
		00	01	11	10
previous home (h)	0	0	1	0	race
	1	1	race	1	0

Output for the new home $h' = h \oplus t \oplus d$
Race detection $r = (h \oplus t) \cdot (d \oplus t)$

MENPS decomposes the barrier into a release fence and an acquire fence. Both fences can be individually issued at each process.

A release fence applies the diffs of all of the writable blocks. Because MENPS employs an eager protocol (see also Section 6.1.1), every writable block must be merged before stepping out of the fence. When a release fence is issued, merge operations on all of the writable blocks within the process are done in parallel.

An acquire fence takes a signature as its argument to correctly invalidate the caches. This signature is transferred from the releaser process which synchronizes with the acquirer process. As in the release fence, the acquire fence invalidates the caches in parallel by calling `mprotect()` from multiple threads. However, we are suspecting that the performance gain of this parallelization is small because the operating system serializes most of the page management operations as described later in Section 3.8.4.

To implement these fences, each process records a read set and a write set, which track readable and writable cache blocks within the process. In our current implementation, the read set is implemented as a priority queue sorted by read timestamps to efficiently enumerate the invalidated blocks based on their timestamps, and the write set is a list consisting of the IDs of writable blocks. Each process also has a minimum write timestamp to preserve coherence.

A barrier operation first issues a release fence, and then the signature returned by the fence is exchanged with all of the running processes. In the current implementation, `MPI_Allgather()` is used for this all-to-all communication. Because it is not easy to implement a scalable allgather operation in general, we consider that we should implement a communication function to reduce signatures from all of the processes, but we have not tried in this paper. Finally, an acquire fence is issued to invalidate the caches based on the signatures.

3.5.4. Write serialization and diff processing

We implemented a distributed queue locking mechanism for the mutual exclusion of cache blocks. Implementing queue locking in RDMA is not a new idea (e.g. [170]), but our method is different from normal locks because it combines the probable owner method with queue locking in order to decentralize the implementation of home lookup and locking. Although ordinary queue locking (e.g. [129]) consumes the constant space for each lock when unlocked, our method always consumes the storage proportional to the number of processes.

Listing 3.3 shows the pseudocode of our distributed queue locking. Each process has a probable owner value (`links[i]` in the code) for each cache block. Each link value is in three possible states: UNLOCKED, LOCKED, and a link to another process. The link of the first owner process is initialized as UNLOCKED and all of the other processes are pointing to this owner. When a certain process starts locking, it traverses the distributed links using RDMA compare-and-swap. If there are multiple processes are waiting for acquiring the same lock, the current owner process notifies the next waiting process via a send/receive communication.

Table 3.1 shows the XOR-based diff merge method used in MENPS. We know that Ramesh et al. [158] also employed XOR-based diff merging and we added a feature to dynamically detect write-write race conditions. Compared to a naive implementation of merging using conditional statements, the calculations of both the output and race detection can be parallelized using SIMD operations. Because the data modification is

```
const int UNLOCKED=1, LOCKED=2, LINK0=3;
// links[i] is owned by i-th process
int links[N_PROCS];
int lock(int cur /* running process ID */) {
  int l = links[cur];
  if (l == UNLOCKED) {
    l = CAS(&links[cur], UNLOCKED, LOCKED);
    if (l == UNLOCKED) return cur;
  }
  links[cur] = LOCKED;
  int p = l-LINK0;
  while (true) {
    l = CAS(&links[p], UNLOCKED, LINK0+cur);
    if (l == UNLOCKED) return p;
    else if (l == LOCKED) {
      l = CAS(&links[p], LOCKED, LINK0+cur);
      if (l == LOCKED) { Recv(p); return p; }
    }
    if (l >= LINK0) p = l-LINK0;
  }
}
void unlock(int cur) {
  int l = link[cur];
  if (l == LOCKED) {
    l = CAS(&links[cur], LOCKED, UNLOCKED);
    if (l == LOCKED) return;
  }
  Send(l-LINK0);
}
```

Listing 3.3: Pseudocode of our distributed queue locking method combined with probable owners.

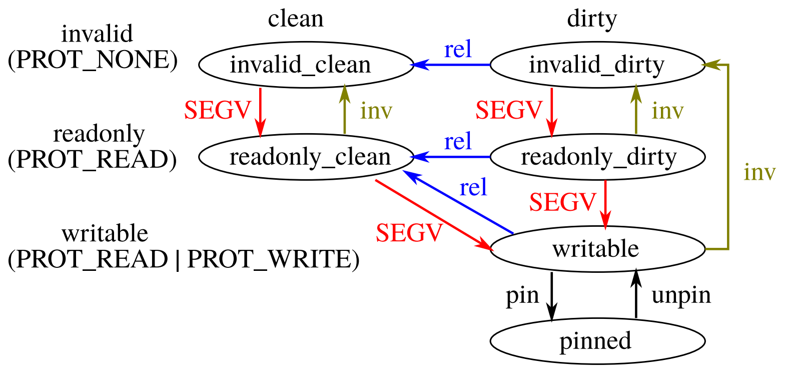


Figure 3.19.: State transitions of Menps for a single cache block.

guarded by the global lock as explained before, it is possible to use SIMD instructions in this case. This feature of race detection enabled us to find the concurrency bugs of both the DSM system and its benchmark programs. Note that read-write data races cannot be captured by this method, and not all of write-write races can be detected (e.g. races between the threads that share the same process).

3.5.5. State transition mechanism

Figure 3.19 shows the state transition rule of our DSM system. Typical shared-memory systems classify the memory states into invalid, readonly, and writable. Our method extends this classification to implement two advanced features. First, our method splits invalid and readonly states into clean or dirty states. This distinction enables the system to delay executing the critical section of an invalidated writable block. This method is called "two-way merging", which is originally proposed in the paper of a software DSM system Cashmere-2L [178]. When a writable block is being invalidated, conventional protocols merge the diff immediately and after that the state transitions to invalid. However, in the same situation, ours only changes the state to invalid_dirty and does not issue the merge at that time. Because the merge is delayed until the release fence, this method can shrink the latency of the acquire fence.

Second, to place a call stack on the DSM, we added a "pinned" state[4] which disables the page protection control of the DSM system. When the upper-level threading system tries to execute a user-level thread that places its call stack on the DSM, it pins the memory area for the call stack.

This feature was introduced to avoid a deadlock problem inside a segmentation fault handler of our DSM. Consider a situation where the call stack of a user-level thread is placed on the DSM and the thread has acquired a mutex of the system (e.g. a communication system or a standard library). Then, one of the pages for the stack transitions to the invalid state due to cache invalidation. If the segmentation fault handler tries to acquire the mutex again, this becomes a deadlock because the same thread acquired the mutex again. In general, it is hard to replace all of the related mutexes with recursive mutexes which can serve multiple lock acquisitions. Therefore, we consider that a practical solution for this problem is to avoid entering the segmentation fault handler to resolve the cache miss of a call stack.

The pinned state is added to disable the page protection of Menps for cache blocks where call stacks are placed. This state can be controlled by the upper layer, for example, the OpenMP layer MEOMP explained next.

3.6. MEOMP: OpenMP runtime on Menps

3.6.1. OpenMP on an everything-shared DSM system

We have chosen OpenMP for the evaluation of Menps because it is common to use OpenMP for writing shared-memory applications. In order to evaluate Menps apart from existing OpenMP implementations, we implemented a simple scheduling layer MEOMP that works as an OpenMP runtime on the DSM system instead of trying to execute existing schedulers transparently.

MEOMP is the runtime library that can run programs with simple OpenMP pragmas on top of Menps. Menps itself is not restricted to the execution model of OpenMP and works as a general coherent caching system for arbitrary shared-memory programs. MEOMP implements a simple scheduling layer for OpenMP programs with the extensive use of Menps' features.

There are existing attempts to implement OpenMP systems on top of DSM systems, which are later described in Section 6.1.4. Most of them are based on a special compiler or a source-to-source translator, but MEOMP requires neither of them because it implements the same ABI functions as the compiler defines. Because Menps is capable of sharing not only heap memory but also call stacks, it enables MEOMP to implement a purely library-based approach for OpenMP. As far as we know, NanosDSM [53] is an "everything-shared" DSM system which enables to implement an OpenMP runtime system without any code transformation specific for DSM. Menps is also one of everything-shared DSM systems, but it is different from NanosDSM because it supports the commodity compilers including GCC and Intel compilers.

Because OpenMP specifies a data-race-free memory consistency model, it is unnecessary to modify the memory model of DSM systems specifically for OpenMP.

[4]This state was named "pinned" because it is similar to what memory pinning of operating systems realizes. This special state fixes the page protection to the writable and avoids downgrading.

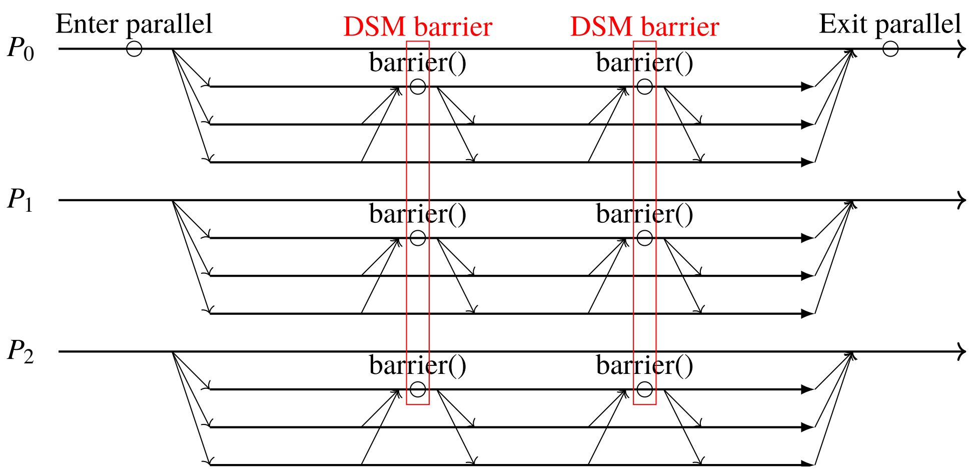


Figure 3.20.: Example of OpenMP runtime behavior. Each process, either master or worker process, has three child worker threads forked from the master thread. In this example, there are nine worker threads in the cluster in total.

3.6.2. Execution flow of MEOMP

MEOMP starts as a normal MPI program with multiple processes. There are the master process, where the MPI rank is zero, and the other worker processes. The current implementation of MEOMP defines the `main` function to set up the environment for the user program. When this main function starts, MEOMP configures the environment for the communication libraries (MPI and UCT) and the underlying user-level threading library. Then, MEOMP initializes the segment of global variables, the segment of call stacks, and the segment of the heap.

Figure 3.20 shows an example execution flow of MEOMP. After the initialization phase, the master process switches to one of the call stacks on the DSM segment and calls a user-defined main function `meomp_main()`:

```
int meomp_main(int argc, char** argv);
```

This main function is invoked only in the master process. The worker processes are idle while the master thread (of OpenMP) on the master process is executing the sequential parts of the program.

When the master thread reaches the beginning of an OpenMP parallel region, the master process distributes a command which instructs other worker processes to execute the same parallel region in parallel. The command specifies the function pointer to the parallel region code, which is automatically separated by the OpenMP compiler, and the arguments for the function. Child worker threads forked by the worker processes invoke the function on top of the call stacks on DSM.

When OpenMP worker threads reach an OpenMP barrier, they first issue a process-local barrier. Then, one of the worker threads in the same process issues a DSM barrier and synchronizes with other processes. To reduce communication between processes, only one thread per process starts the inter-process barrier. Finally, all of the threads issue the process-local barrier again to resume their computation.

MEOMP also provide allocation functions from the heap segment on Menps:

```
void* meomp_malloc(size_t size);
void meomp_free(void* ptr);
```

3.6.3. Scheduling DSM processing

Menps assumes that each process schedules threads via a user-level threading library (Section 2.6.2). User-level threading is useful to implement a DSM system because the fence operations need to process the cache

Table 3.2.: Evaluation environment of Menps (ReedBush-H [182]).

CPU	Intel Xeon E5-2695 v4
	2.1 GHz (max. 3.3 GHz with Turbo boost)
	18 cores × 2 sockets / node
Memory	256GB / node
Interconnect	InfiniBand FDR 4x, 2 links
OS	Red Hat Enterprise Linux 7.2
Compiler	Intel C++ Compiler version 18.1.163
MPI	Intel MPI Library version 2018.1.163

blocks in parallel and latency hiding of inter-node communications is easily expressed with a small runtime cost.

MEOMP provides the functions of the same ABI as existing compilers and their libraries are interfaced. This technique is used by other libraries to run OpenMP programs on different runtime systems (e.g. [122, 95]). This scheduling layer internally uses the features of ComposableThreads (Chapter 5) to implement context switching between an OpenMP worker thread and a background normal thread. Our current implementation serves some primitive functionalities such as distributing a parallel for-loop evenly (specified as "static" scheduling). Dynamic scheduling or other advanced scheduling features are not supported in the current prototype implementation.

There are several limitations to this interfacing method. First, on page-based multiple-writer DSM systems, OpenMP reductions cannot be implemented because some compilers (e.g. GCC) generate the optimized code as the combination of per-thread reductions and atomic instructions. The page protection mechanism cannot distinguish atomic instructions from normal store operations. Second, OpenMP threadprivate variables are simply converted into thread-local storage (TLS). Because Menps runs OpenMP worker threads as user-level threads, TLS may be shared by multiple worker threads. Due to these issues, our system does not support these features currently.

3.7. Evaluation of Menps

The evaluation environment is shown in Table 3.2. This cluster can run programs using up to 32 nodes. We used MassiveThreads[5] [134] as the underlying user-level threading library. We have run Menps using two processes per node in order to avoid NUMA issues, and each process executes 17 worker threads of MassiveThreads (one core is left for the OS). Each process also runs 16 OpenMP worker threads on MassiveThreads and one core is reserved for MPI communication. Therefore, we run 32 OpenMP worker threads per node with Menps in this setting. We also measured the performance in ICC OpenMP or Intel MPI using all of the cores (36 cores per node).

3.7.1. Application benchmark: NAS Parallel Benchmarks

To evaluate Menps, we selected an unofficial version [189] of NAS Parallel Benchmarks (NPB) ported to C and OpenMP because most of the original benchmark programs are written in Fortran (only IS is written in C). Menps is implemented in C++ and mainly assumes C/C++ applications with OpenMP. For the evaluation in MPI, we used the original Fortran codes of NPB 3.3.1. We did not evaluate some applications including NAS LU, SP, and MG due to implementation issues in Menps.

[5] Strictly speaking, we replaced mutexes and barriers of MassiveThreads with those of ComposableThreads. This mode is called CTMTH in Chapter 5.

Table 3.3.: Sequential execution time of NPB (the unofficial C+OpenMP version, OpenMP is disabled).

Name	Class	Time [s]
EP	D	4778.36 ± 1.06
	C	301.03 ± 1.02
CG	D	11619.82 ± 782.61
	C	365.13 ± 16.22
FT	C	357.55 ± 2.04
IS	C	12.30 ± 0.07
BT	A	88.06 ± 2.97

Table 3.4.: Parameters of Menps for NPB.

		DSM heap		DSM global variables	
Name	Class	Total size	Block size	Block size	Length of a WN list
EP	D	1 MiB	32 KiB	32 KiB	1024
	C	16 MiB	32 KiB	32 KiB	1024
CG	D	32 GiB	256 KiB	4 KiB	4096
	C	8 GiB	32 KiB	32 KiB	128
FT	C	4 GiB	1 MiB	32 KiB	128
IS	C	4 GiB	1 MiB	32 KiB	1024
BT	A	1 MiB	32 KiB	128 KiB	1024

Because the applications in NPB depend on some OpenMP features which are not currently supported by Menps, we replaced them with alternatives for the evaluation. Reductions are changed into sequential loops only in the results with the DSM. We also converted threadprivate variables to automatic variables because those variables were temporarily living in a certain scope. These changes are disabled during the evaluation on ICC OpenMP.

We also compared the performance of Argo [100] with Menps using NAS EP and CG. In Argo, we used 36 worker threads for each process. Because Argo does not support OpenMP directives directly, we needed some modifications to the application. We implemented a thin wrapper library that imitates OpenMP directives using Pthreads and converted every directive to a library function call using C++ lambda functions.

We tried to run some other benchmarks in NPB but failed due to these reasons:

- LU uses the OpenMP flush directive to communicate with neighbor processes in a more fine-grained manner than barrier synchronizations. Although we originally designed our runtime to support such fined-grained synchronizations, we are not sure that our implementation can guarantee the semantics of this directive.
- We ported SP and it can complete its execution, but the output result of this benchmark was corrupted when it was executed with multiple processes in our system. We are investigating whether this bug is due to our system or the benchmark program.

Table 3.3 shows the execution time of NPB without parallelization. To measure these values, we disabled the feature of OpenMP by removing the compiler option `-fopenmp`.

Table 3.4 shows the DSM parameters which we used for NPB. We adjusted these values to maximize the performance in multiple nodes. The heap size may be larger than the actual demand of the application because we set some sufficient values.

3.7.2. Application benchmark: matrix multiplication

This benchmark program executes matrix multiplication of two 50000×50000 regular matrices once. It computes $C + A \cdot B$ where A, B, and C are the input dense matrices of double-precision floating-point numbers, and updates C with the result. The main computation first enters an OpenMP parallel region, computes the submatrices divided by the rows of C, and finally exits the parallel region. Before executing the main computation, the benchmark program executes a warm-up phase which executes the same computation to load the matrices.

The benchmark program is written with OpenMP as in NPB. For each thread, the main computation uses BLAS of Intel MKL to calculate the submatrices. The sequential time is measured with the same OpenMP program with OpenMP disabled.

3.7.3. Microbenchmark: memory read on DSM

This microbenchmark measures the performance of memory reading. The benchmark program repeats two phases: a writing phase and a reading phase. In the writing phase, one process writes the memory region with the specified size. In the reading phase, another process reads the whole written memory. OpenMP barrier operations are issued at every phase change.

3.7.4. Microbenchmark: false sharing benchmark

This microbenchmark measures the performance of when false sharing happens. In this microbenchmark, there are multiple writer processes that write on the same cache block in Menps. The cache block size is set to 32 KiB in this experiment. Each process writes the non-overlapping stripes of the cache block with the specified stripe size. For example, when the stripe size is 128 bytes, process 0 writes the range of [0, 127] bytes, process 1 writes [128, 255] bytes, and so on. When this process number reaches the number of processes, it starts cyclically from 0 again.

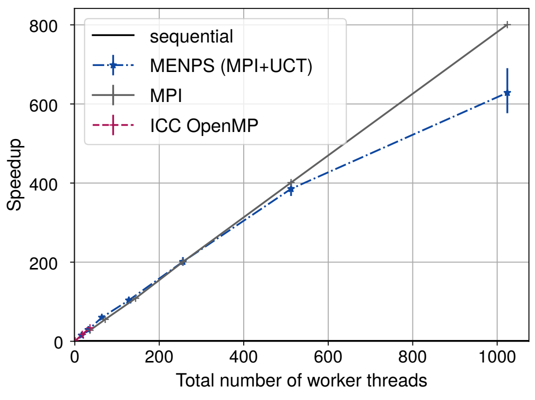
This microbenchmark was conducted both on Menps and Argo. This microbenchmark is artificially designed to see the worst case of the floating home-based method of Menps because it needs to frequently move the whole cache block. For false sharing cases, coherence protocols using fine-grained writes may perform better than the floating home-based method.

3.8. Evaluation Results: NAS Parallel Benchmarks

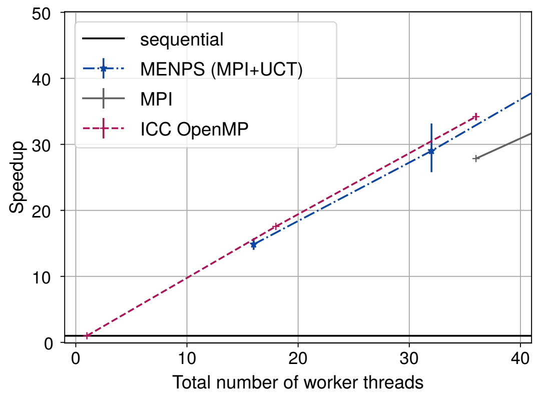
The scalability results of NPB are shown in Figure 3.21. All of the speedups are normalized to the sequential results in Table 3.3. First, it is visible that Menps can scale NAS EP. We note that it is relatively easy to scale this benchmark because no communication is involved during its computation phase as its name Embarrassingly-Parallel suggests. NAS CG is another benchmark program that Menps can perform better in multiple nodes than the single-node OpenMP. The speedup benefit saturates at around 63 times with 128 cores on Menps. On the other hand, the performance improvement in MPI flattens with two to four nodes and then scales better than other systems. It is difficult to determine the reason for this result because there are many differences between ours and MPI's version including the languages.

In NAS CG, we also compared the two conditions using MPI+UCT vs. MPI-only in Menps. When UCT is enabled, RDMA operations are executed in UCT, and when disabled, they call MPI-3 RMA functions instead. Because UCT provides a low-overhead interface to the interconnects, the performance of MPI+UCT is 21% better than MPI-only with 128 cores (four nodes).

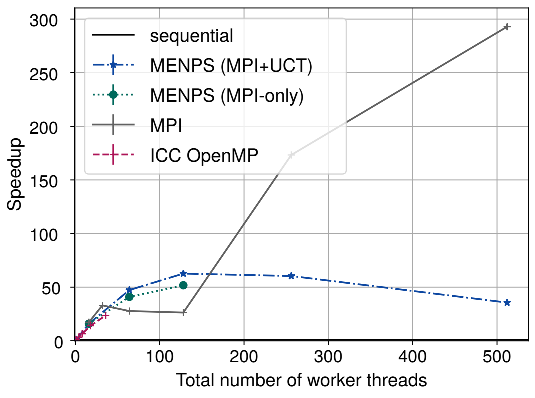
Although Menps could successfully execute the other three benchmarks, there was no performance gain compared to single-node configurations. Several possible reasons prohibit our system from scaling these benchmarks. Because the problem sizes of these benchmarks especially in NAS IS are smaller than EP and



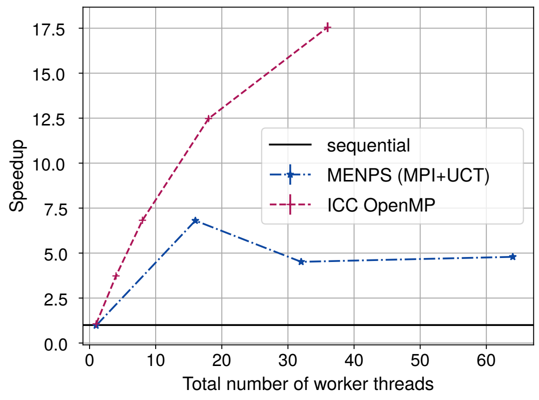
(a) NAS EP (CLASS=D)



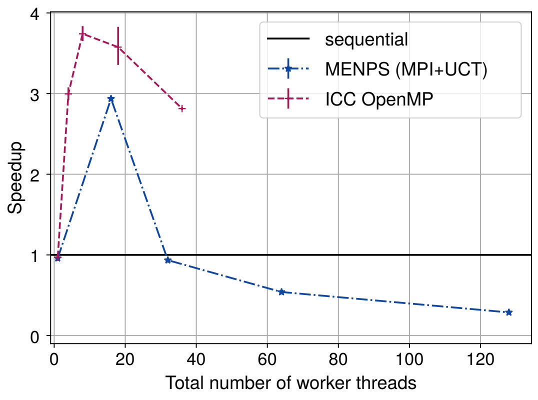
(b) NAS EP (CLASS=D) (zoomed)



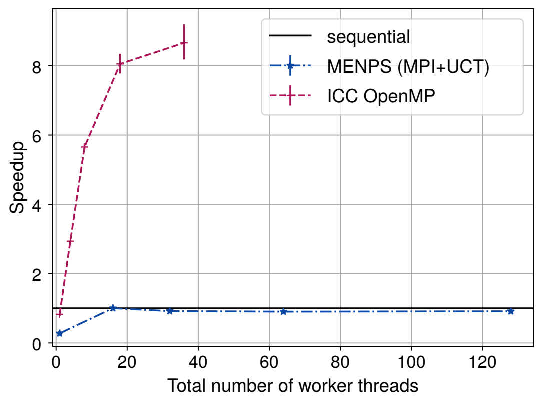
(c) NAS CG (CLASS=D)



(d) NAS FT (CLASS=C)



(e) NAS IS (CLASS=C)



(f) NAS BT (CLASS=A)

Figure 3.21.: Speedup comparisons between Menps, ICC OpenMP and MPI using NPB.

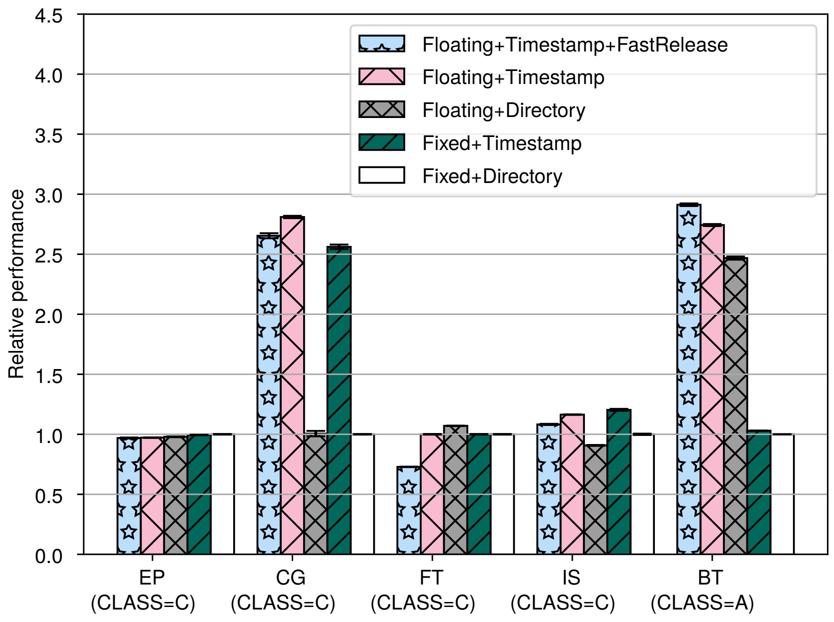


Figure 3.22.: Normalized performance improvement of different home-based methods and invalidation methods with 64 cores (two nodes). All of the values are normalized to the results of the baseline setting Fixed+Directory.

CG, it is hard to get the performance benefit of multiple nodes. Our implementation could not run larger problem sizes in these benchmarks due to the size of the physical memory per node.

3.8.1. Directory, Fixed Home, vs. Floating Home + Timestamp invalidation

Figure 3.22 shows the comparison of our proposed methods with the baseline methods. The floating home-based method is compared with the situation which cyclically assigns the home process based on the block ID (fixed-home method). We note that our implementation of the fixed-home method performs differently from Argo's method because ours serializes the diff merge but Argo merges diffs with distinct RDMA WRITEs (typically incurring a large number of small RDMA WRITE messages). In the experiment on Menps, when the home process is fixed, we observed that BT had a significant slowdown by a factor of 3.2 times. We think that BT continuously repeats the writes to the same blocks on the same process across barriers and the home migration is indeed important to accelerate such a situation.

We compared our timestamp-based invalidation scheme with directory-based coherence on Menps. The performance result of CG using timestamp-based invalidation is 3.1 times better than directory-based coherence, but that of FT becomes 36% worse. Our invalidation method works efficiently if there is enough space for holding the write notices in a signature and logical leases can roughly track the ordering of cache blocks. Otherwise, precise sharer tracking using directories can have a better performance result as in FT.

Overall, the combination of the floating home-based method and the hybrid invalidation method was the most stable approach to run these benchmarks. It also achieves the best performance in NAS CG, which Menps can surpass the single-node OpenMP runtime in the absolute performance.

3.8.2. Impact of Write Notices

To examine the effectiveness of our invalidation method, we examined the impact of changing the length of a write notice list in NAS CG with two nodes. Figure 3.23 shows the execution time with various lengths

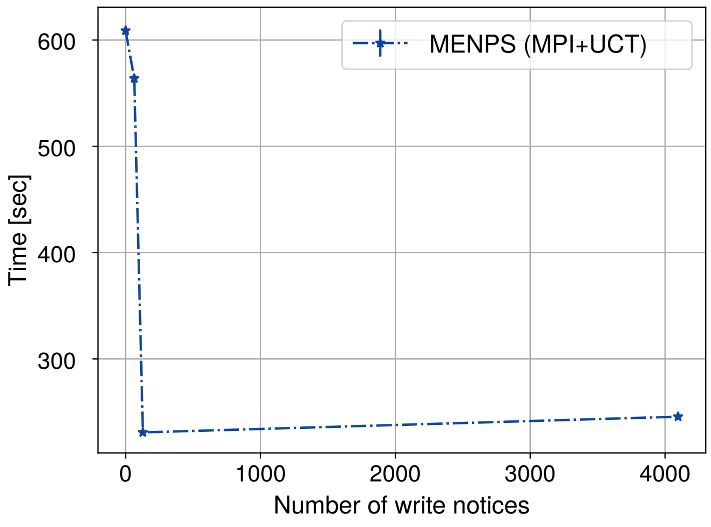


Figure 3.23.: Execution time of NAS CG (CLASS=D) on MENPS using 64 cores (two nodes) with various lengths of a write notice list.

of the list. This result shows that there seems to be a performance peak of around 128. If the number of write notices is zero in our system, it means that cache invalidations are accomplished only with logical leases-based invalidations. We found that using only logical lease-based invalidations would significantly degrade the performance of DSM systems because it frequently enters the critical sections for cache blocks to update timestamps. If there is a sufficient space to hold write notices, write notice invalidations can effectively reduce the number of those critical sections. However, when the write notice list is too large, it adds unnecessary traffic for exchanging the signatures and hence slightly degrades the performance.

3.8.3. MENPS vs. Argo

Figure 3.24 shows the scalability comparison of NAS CG (CLASS=C) with Argo[6]. Although Argo's paper reported a good scalability result with the same application and problem size, we observed that Argo was not scaling with multiple processes in our environment. On the other hand, MENPS performs better than ICC OpenMP with multiple processes. The performance benefit of MENPS is smaller than NAS CG (CLASS=D) (Figure 3.21) because of the insufficient problem size.

We also compared the performance of NAS EP with Argo. Figure 3.25 shows the scalability comparison of NAS EP (CLASS=C) with Argo. MENPS scales NAS EP with less than 238 cores (= 7 nodes). However, with more nodes, it stops scaling due to critical sections for reducing the results. Compared with the CLASS=D result (Figure 3.21), this smaller problem finishes within two seconds with 238 cores, which seems hard to scale. Although the performance of EP on Argo was improved with more nodes, the absolute speedup on Argo did not reach the speedup with single-node OpenMP.

There are several differences between MENPS and Argo. MENPS utilizes user-level threading, but Argo heavily depends on the usage of Pthreads and incurs an overhead in multi-threading. Argo is partially depending on directory-based coherence, which increases the number of coherence operations for each memory operation.

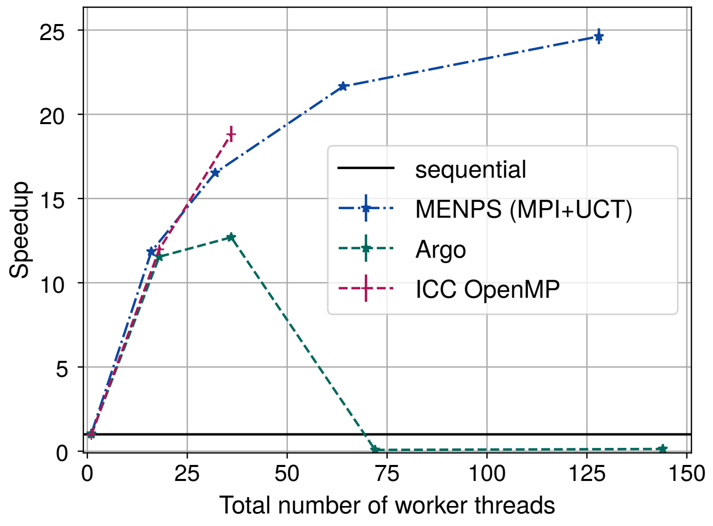


Figure 3.24.: Scalability comparison of NAS CG (CLASS=C) between Menps, Argo, and ICC OpenMP.

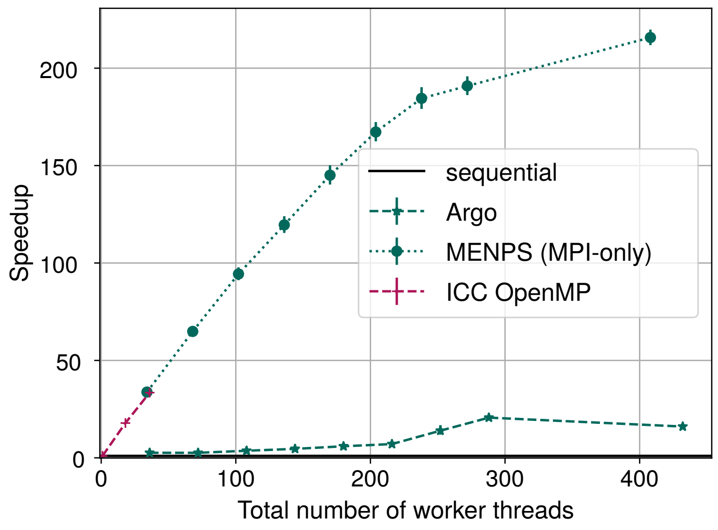
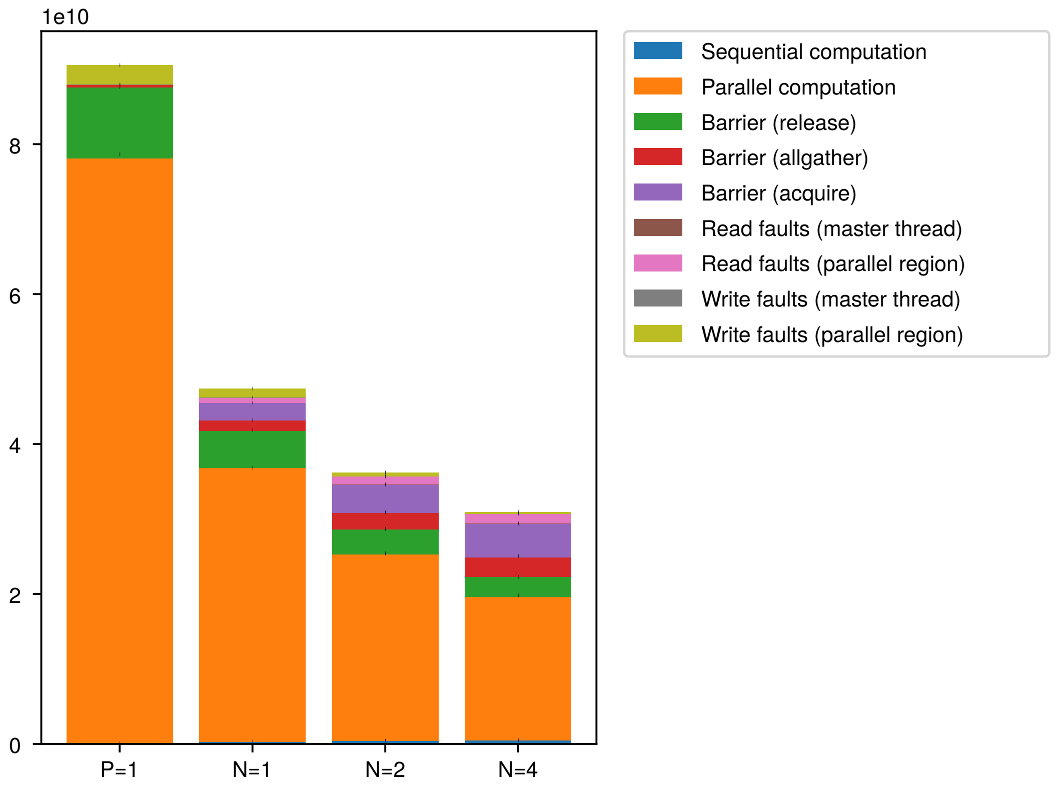
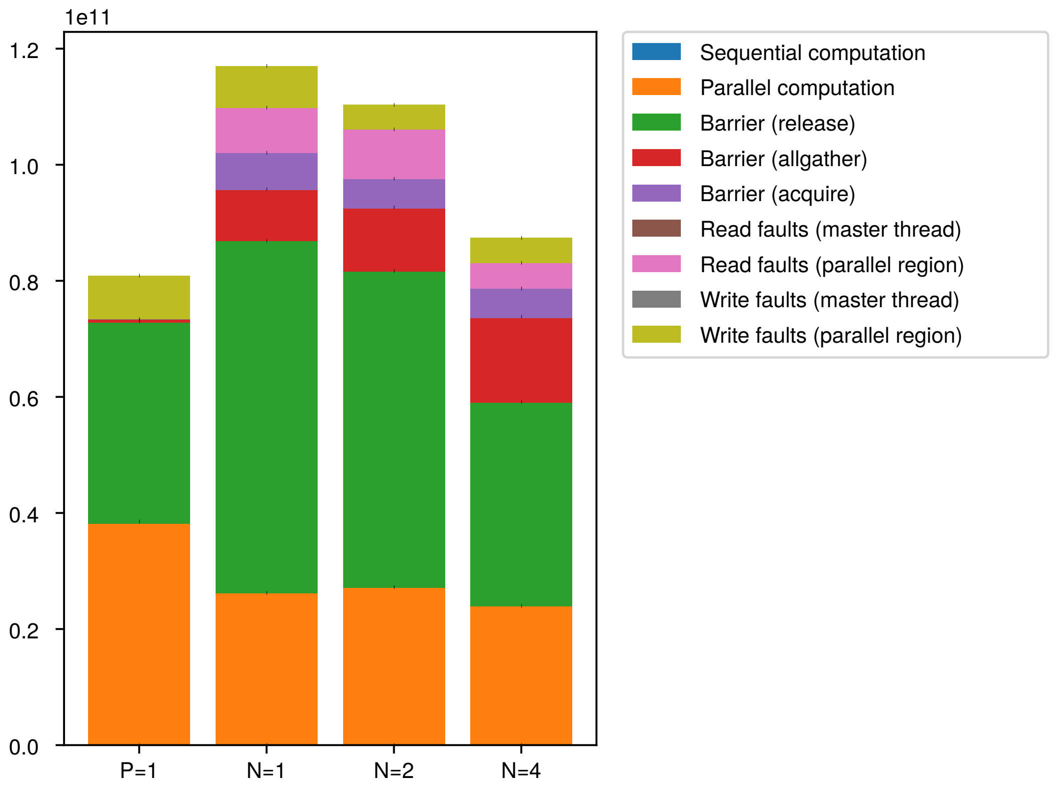


Figure 3.25.: Scalability comparison of NAS EP (CLASS=C) between Menps, Argo, and ICC OpenMP.



(a) NAS CG (CLASS=C)



(b) NAS BT (CLASS=A)

Figure 3.26.: Execution breakdown of NAS.

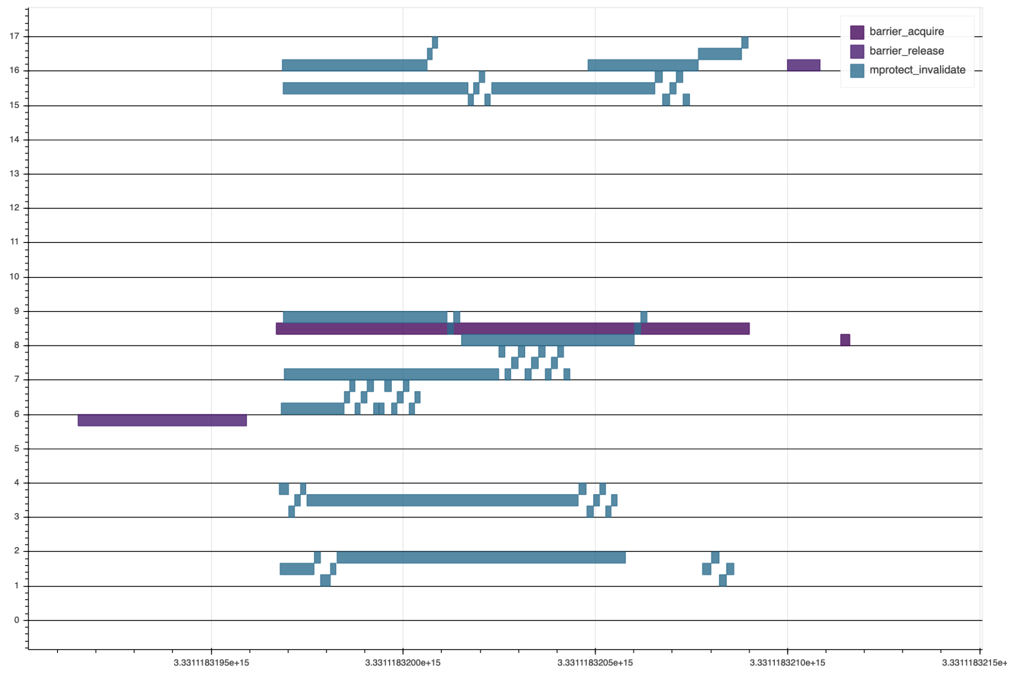


Figure 3.27.: Execution trace of an acquire fence in NAS CG (CLASS=C). X-axis is the execution time in cycles. Y-axis is the worker thread number provided by the user-level thread library.

3.8.4. Execution breakdown of NAS on Menps

Figure 3.26 shows the execution breakdown of NAS CG and NAS BT. The major causes of overhead in these two applications were different. In NAS CG, the main bottleneck with multiple nodes is the acquire fence operation in barriers. The overhead of the acquire fence largely comes from invoking the `mprotect()` system call because the role of the acquire fence is to invalidate the written cache blocks. As mentioned by Hong et al. [84], for example, the paging overhead is crucial for modern multi-threaded DSM systems supposed to be run on CPUs with a large number of cores per node. Although Menps parallelizes the invalidation part of the acquire fence using multiple threads for all of the invalidated cache blocks (via user-level threads), the operating system may sequentially process the page table manipulations inside.

On the other hand, the overhead of NAS BT is largely dominated by the release fence. NAS BT is considered a write-intensive application because the performance of the release fence depends on the number of writable cache blocks within a process. The release fence is parallelized by Menps for all of the writable blocks, but it was not enough to accelerate NAS BT in the overall execution time. It is also observed that the other parts including faults or acquire fences are shrinking with more nodes.

Figure 3.27 shows an execution trace of fence operations in NAS CG. The event kind *barrier_acquire* denotes the execution time of an acquire fence. It can be seen that some of the `mprotect` invocations in the fences take a huge amount of time (over $100,000$ cycles). This slowdown happens inside the operating system.

Figure 3.28 shows an execution trace of fence operations in NAS BT. BT includes many fence operations shown as purple lines and there is a strangely long fence in the middle of this trace. The green line below this fence corresponds to a *single* RDMA operation, which strangely consumed about two seconds.

We think that this problem is caused by the retransmission interval of InfiniBand hardware. When an RDMA operation fails in reliable connections, the network hardware checks its failure and retries the operation. The maximum interval for checking can be changed by the `timeout` value, but it must be no

[6]During this experiment, we found a bug of Menps which rarely made the program stuck in its initialization with multiple processes. Because the output results were correct when the initialization succeeded, we omitted the cases of buggy executions.

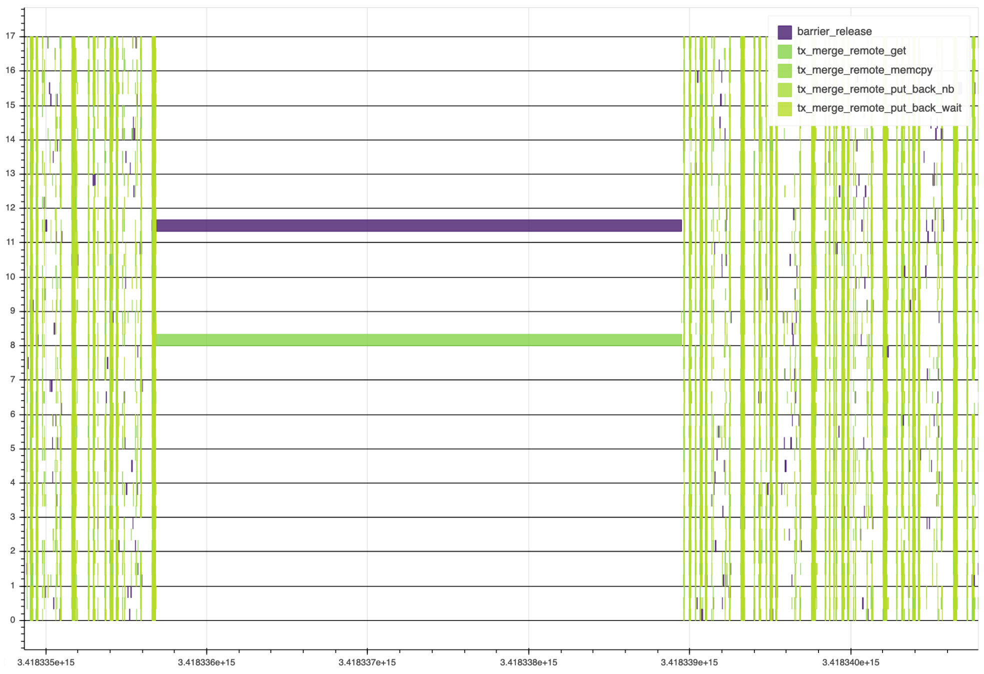


Figure 3.28.: Execution trace of a release fence in NAS BT (CLASS=A). X-axis is the execution time in cycles. Y-axis is the worker thread number provided by the user-level thread library.

less than `local_ca_ack_delay`. In our environment, `local_ca_ack_delay` was 16, which means that the hardware checks in the interval of 0.268 to 1.074 seconds. This value explains why a single RDMA operation sometimes does not return in the ordinary latency but consumes a few seconds.

Both the breakdown and trace data are collected and examined by the performance visualizer tool MassiveLogger [186], which we have developed for Menps and other applications.

3.9. Application Benchmark: matrix multiplication

Figure 3.29 shows the speedup comparison of matrix multiplication. It is still far from the ideal speedup in relatively large scales, but the relative speedup on Menps to the sequential program is 416 times using 1152 threads (32 nodes). The sequential performance of this benchmark program is 35.9 GFLOPS, which was slightly better than the hardware specification (33.6 GFLOPS per core)[7]. From this result, Menps keeps accelerating the performance of BLAS in multi-node cases.

3.10. Microbenchmark results

3.10.1. Microbenchmark results: memory read benchmark

Figure 3.30 shows the performance of the memory read benchmark. It is observed that single-threaded cases (`num_threads=2`) perform better than multi-threaded cases when the size of the copied region is smaller than 1 MB. When Menps is running a single-threaded program with multiple processes, it becomes faster to change the page protection because the page table is owned by a single thread. Multi-threading cases suffer from the overhead of refreshing each core's TLB. With more data transferred, multi-threaded cases perform better than single-threaded cases because they can switch the copying threads and hide the communication latency.

[7] A possible reason of this performance result is the effect of TurboBoost.

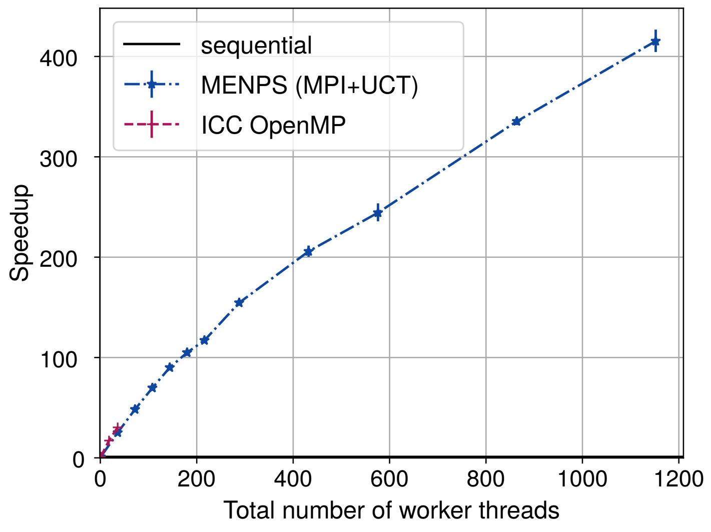


Figure 3.29.: Speedup comparison of matrix multiplication benchmark.

Table 3.5.: Changes on NPB to run on two DSM systems.

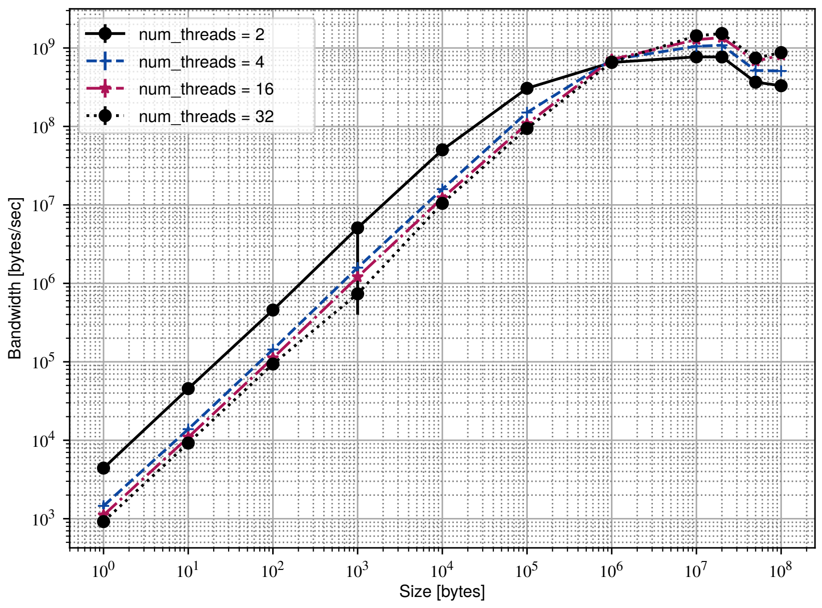
Name	SLOC (original)	Changes for MENPS	Changes for Argo
EP	269	80 insertions(+), 16 deletions(-)	497 insertions(+), 90 deletions(-)
CG	921	141 insertions(+), 33 deletions(-)	945 insertions(+), 444 deletions(-)
FT	1263	166 insertions(+), 24 deletions(-)	
IS	710	115 insertions(+), 34 deletions(-)	
BT	3713	66 insertions(+), 33 deletions(-)	

3.10.2. Microbenchmark results: false sharing benchmark

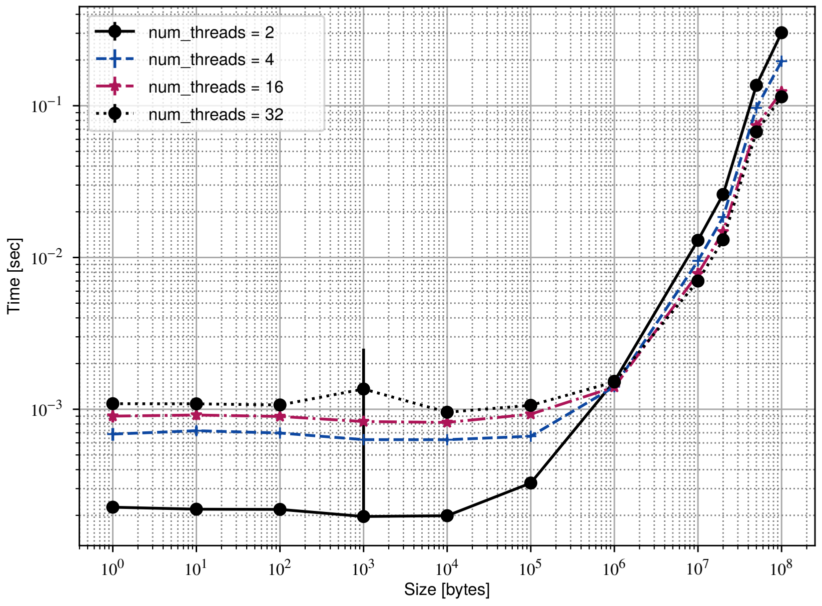
Figure 3.31 shows the latency results of the false sharing benchmark. This latency is the average time of each iteration which consists of writing the corresponding stripes and issuing a barrier. The latency on Argo highly depends on the size of each stripe. When the stripe size is larger than 4 KiB, Argo completes the workload in the short latency because Argo manages caches as 4 KiB pages and the amount of false sharing is reduced. When the stripe size is much smaller than 4 KiB, the latency on Argo is increased because Argo suffers from the overhead of fine-grained writes (Section 3.2).

On the other hand, the latency of this benchmark on MENPS becomes almost flat with varying sizes of stripes. This is because MENPS uses a floating home-based method, which merges the cache blocks in a coarse-grained manner. In MENPS, the latency only depends on the number of writers which write the same cache block because those writers complete the merges in serial.

This benchmark is artificially designed to examine the problem of coarse-grained writes. Even though the fine-grained approaches look much better than coarse-grained approaches for false sharing cases, MENPS still sometimes performs better than Argo when the number of processes is small. It is simply because RDMA cannot efficiently transfer fine-grained writes. We can confirm that the coherence protocol of MENPS which transfers coarse-grained blocks performs almost comparable with fine-grained approaches with false sharing because of its RDMA-based design.



(a) Bandwidth of memory read.



(b) Execution time of memory read benchmark.

Figure 3.30.: Performance results of memory read benchmark. num_threads is the number of total threads on both writer and reader processes. num_threads=2 is the minimum because this benchmark uses two processes.

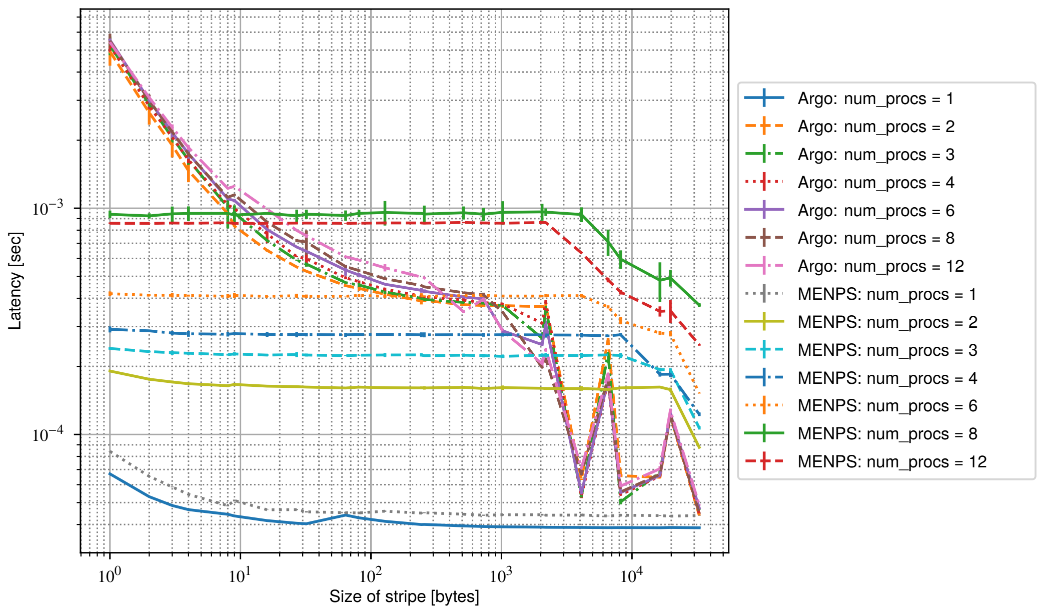


Figure 3.31.: Latency results of false sharing benchmark. num_procs is the number of writer processes which write on the same cache block (32 KiB).

3.11. Measuring Productivity

To examine the productivity of two DSM systems, we measured the changes made to the original programs in SLOC. Table 3.5 shows the changes to NPB to run on two DSM systems. Because Menps supports transparent execution of OpenMP programs, it requires a smaller amount of changes to the OpenMP programs than Argo requires. The large part of changes in Argo comes from our boilerplate code for running on Argo. If the interface of Argo is extended to support the easy translation of OpenMP programs, we think that the changes for Argo can be reduced.

3.12. Summary of Menps

To improve the productivity of distributed-memory programming, we developed a distributed shared memory (DSM) library Menps with several new techniques to realize an efficient DSM system on modern hardware. Menps can transparently execute shared-memory applications with simple OpenMP directives and achieve good application productivity. To implement dynamic data migration using RDMA, we used an idea to float the home process of a cache block. We also noticed a good combination of logical timestamps and write notices for cache invalidation, and provided a hybrid invalidation scheme which can be simply implemented in a scalable manner. These features eliminate the use of remote message handlers and enable DSM to exploit the performance of RDMA.

To show the efficacy of our methods, we evaluated Menps with some programs of NAS Parallel Benchmarks. Our library does not always perform better than a normal OpenMP runtime system even with multiple nodes but can scale NAS EP, CG, and matrix multiplication with almost the same program written for a single node. We also experimentally showed that Menps was more efficient than a DSM library Argo using NAS EP and CG.

3.12.1. Future work

It is still hard to make Menps practical for more complicated applications. Here is the summary of future work for Menps:

Feasibility for complex applications

The significant benefit of distributed shared memory is the freedom to adopt C pointer-based data structures. From the perspective of system development, it enables to reduce the amount of codebase. For application developers, some algorithms can be simply implemented using pointers. For example, graph processing is such an algorithm that depends on a huge amount of indirect references. The feasibility of Menps in such a complex application has not been evaluated, but the current implementation of Menps is ready to do such an experiment with minimal modification of the program.

Implementing remote memory paging

One important problem with our current implementation is the available memory space constrained by the size of the physical memory of each node. Menps is initially intended to speed up shared-memory programs that can be executed in a single node, but it would be beneficial if a DSM system can transparently execute the programs which demand a huge memory space. This concept is also known as remote memory paging, which is described in Section 6.1.6. Implementing remote memory paging with RDMA is not straightforward because RDMA interfaces require memory pinning of transferred data (Section 2.3.1).

Prefetching and background coherence actions

For example, Argo implements both prefetching and write-back threads to accelerate reads and barriers. Because Menps does support these features, the latency of cache misses and release operations have substantial overhead. If it is possible to efficiently implement the background threads or services concurrently with application programs, the latency of those operations can be reduced. If there is no cooperation from the user programs, it is necessary to predict memory access locations in the future. Introducing software prefetching functions may be the first step to further shrink the read latency.

Solving timestamp-based coherence issues

Menps also has a problem with overflowing timestamps, but we think that this problem can be fixed as proposed in TC-Release++ (Section 6.3.3) because overflowing timestamps is rare if the integer size is sufficiently large.

Better broadcasting support

In general, shared memory systems are not good at broadcasting the cache blocks to multiple processes because the data which may frequently vary cannot be simply replicated. As an example, Yu et al. [196] reported *hot spot* phenomena of TreadMarks, in which a single-writer node is responsible for distributing diffs to all of the other nodes. This problem also applies to PGAS systems because the global data only exists in a single node and all of the other nodes need to access the same location. On the other hand, MPI can broadcast the data easily because the application explicitly invokes the broadcast operation.

Implementing a broadcast-like protocol in shared-memory systems is challenging, but we think that it is not an impossible thing. Initially, Menps is designed to be capable of implementing "reader-to-reader transfers" in which a reader process replicates the cache to another reader process again. Timestamp-based coherence is also useful to implement reader-to-reader transfers because the validity of cache replicas can be checked with a tiny cost.

Implementing missing features of OpenMP

There are some missing OpenMP features such as reductions and threadprivate. These features are not supported because GCC expands them as optimized operations such as atomics or Pthreads' TLSs. It is difficult to convert them in a completely transparent way, but there should be a simple mitigation functionality to avoid them. Ideally, it should be possible to run existing OpenMP runtime systems (e.g. BOLT [95]) for hardware shared memory, but there are many possible technical problems including the use of atomic operations.

Reducing paging overhead

Since Menps is a page-based DSM system, it suffers from the paging overhead of both the hardware and the operating system. When the protection of a page shared by multiple cores changes frequently, *TLB shootdown* is problematic for the performance. LATR [108] is an example to accelerate the page protection mechanism by lazily applying the TLB coherence to other cores on the software side. Such a technique may also improve the performance of page-based DSM systems. Because the overhead of changing the memory protection is challenging for other purposes such as security, recent processors started to support Memory Protection Keys (MPK) [150] to efficiently change the protection state in user space.

4. MECOM: a communication library using parallelized software communication offloading

4.1. Introduction

Modern HPC interconnects in supercomputers provide high-performance communication among cluster nodes. Accelerating communications gains the possibility of performance improvements in a variety of parallel applications.

We observe that communication performance is often limited by the software overheads for message processing. For example, injecting a message into the interconnect hardware takes hundreds of cycles for each message due to the cost of PCIe transactions and other computations for message management [99]. Polling the message completion also requires hundreds of cycles. These software overheads limit the message rate (per core) to about 10 million/sec, but the maximum rate of the hardware is over 100 million/sec in the latest equipment.

The software overheads diminish the benefits of latency hiding because they cannot be overlapped with the computation of an application. Since the latency is greatly reduced to about thousands of cycles per message recently, the impact of the software overheads is becoming relatively larger.

To overcome the message rate limit due to the software overheads, multi-core processing is required in recent CPUs and interconnects. Most of the interconnects provide multiple hardware queues (or more conceptually, "endpoints"), and if multiple cores access different hardware queues, they can work in parallel and improve the aggregated message rate per node.

The network software stack should be also able to handle multiple communication resources. Some low-level communication libraries provide RDMA APIs to explicitly specify an endpoint for each message. In order to avoid putting a burden on the users of selecting the endpoint, some other libraries provide another type of APIs that automatically select an appropriate endpoint. We consider that automatic selection is appropriate in terms of both productivity and performance because ordinary programmers do not have enough knowledge about the underlying hardware.

Because the communication pattern is changing in application phases, communication resources should be shared and dynamically distributed among the cores in the same node. Statically assigning the endpoints to the cores becomes a bottleneck of the message rate. The number of hardware queues is limited in today's interconnects and it is not always possible to allocate them for every core due to excessive memory consumption.

For these reasons, we consider that we need a method to automatically schedule communication requests into multiple communication resources using multiple cores efficiently. We are focusing on *software offloading* [190] approaches, in which application threads delegate communication processing to dedicated threads via non-blocking data structures. Software offloading improves both the message rate and the latency hiding capability because the messages are processed in parallel from application threads.

The problem of software offloading is the consumption of CPU resources if the dedicated threads are busy-waiting. As an example of the existing systems, PAMI [110, 111] has a feature to dynamically start and stop offloading threads to reduce busy-waiting, but their method depends on the special feature of POWER8 processors. Instead, we decided to use a user-level thread library to adjust the number of offloading threads efficiently. Because user-level thread libraries are general frameworks, our approach is widely applicable to most of the modern processors.

We developed a low-level communication library that implements software offloading using user-level threads on a high-performance interconnect, InfiniBand [89]. As far as we know, this is the first attempt to tightly integrating user-level threads into low-level communication. We conducted microbenchmarks to measure the metrics and demonstrated that our offloading architecture was beneficial for improving communication performance.

The fundamental problem is: how to match multiple communication resources and multiple computation resources (= threads) efficiently and concurrently? To give one solution for this problem, we have implemented a communication library MECOM (MENPS COMmunication layer). This chapter explains the design and implementation of MECOM.

4.2. Background

4.2.1. InfiniBand Verbs

We first describe the features of InfiniBand, one of the most widely used interconnection networks in current supercomputers. In the InfiniBand architecture, communication requests between nodes are called Work Requests (WR). A WR is posted to a hardware queue called Queue Pair (QP), which is created both on the sender and receiver nodes of communication. To complete a communication request, a hardware queue called Completion Queue (CQ) is checked based on polling. A QP is statically associated with a CQ in its initialization and a single CQ can be shared by multiple QPs. These hardware resources can be accessed by using the standard API called InfiniBand Verbs. We also use the same term "Verbs" for its implementation provided by OpenFabrics [146].

Before posting a new WR, the memory buffer must be registered to the hardware for zero-copy transfers. Memory registration pins the memory buffer to physical memory and returns the IDs called `lkey` and `rkey`. These IDs must be included in a WR to distinguish memory regions. When the post function (`ibv_post_send()`) in Verbs is called with the parameters indicating a QP and a WR, the WR is transferred to the device via either memory-mapped I/O or DMA [99]. The post function then immediately returns back to the caller function, but at that time the submitted request may not be done. The poll function (`ibv_poll_cq()`) is called to get a completion of the request from the corresponding CQ. WRs can be distinguished by the integers called `wr_id`, which are attached in the post function and later retrieved by the poll function.

In Verbs, a single function call of the post function can serve multiple work requests which are connected as a singly-linked list. It is beneficial for reducing the call overhead when there are many requests. We show that this feature improved the message rates in Section 4.4.

InfiniBand offers several service types. In this paper, we focus on Reliable Connection (RC), which supports all of the operations in InfiniBand. The major drawback of RC is memory consumption because it requires a QP for each destination node. Unreliable Datagram (UD) can reduce the number of QPs because a single QP can be used for all of the destinations, but it supports only the SEND operation and does not support the RDMA operations. Dynamically Connected (DC), which was recently introduced by Mellanox, supports all of the operations as RC while it requires only one QP as UD. However, DC adds overhead compared to RC because it is realized based on dynamic reconnections [149].

All of the functions defined in InfiniBand Verbs are thread-safe. To see how this guarantee is implemented, we surveyed the userland codes of the drivers (mlx4/5) and found that each QP had a distinct *spinlock* to serialize API calls. We also found that QPs and CQs were synchronized by different spinlocks. Therefore, different QPs and CQs can be controlled by multiple threads in parallel.

Both the post and poll functions take hundreds of cycles in each call. To reduce this overhead, Accelerated Verbs (AVerbs) [109, 149] is recently developed by OpenFabrics to bypass Verbs. AVerbs uses some optimization techniques including the utilization of SIMD instructions to efficiently communicate with the I/O region. AVerbs significantly reduces the call overhead in both the post and poll functions and improves the message rates. However, we emphasize here that the software overheads in posting and polling a message cannot be eliminated even with AVerbs.

InfiniBand can also have multiple ports for each adapter card. Multiple ports can also be used for improving the aggregated message rate as multiple hardware queues can be.

4.2.2. Multi-threading support in communication systems

The developers of MPI runtime systems [21, 22] are trying to improve the multi-threading performance in MPI. Although MPI defines MPI_THREAD_MULTIPLE for hybrid parallel programs, the support of this level is still immature in most of the MPI implementations. To enhance the performance of hybrid parallel programs, MPI Endpoints [61] was proposed for the MPI standard. With this proposal, the MPI users can create additional ranks at each process and select an endpoint for each MPI call. It seems that endpoints will not appear in the next MPI standard, but this discussion is still active in the MPI community.

Vaidyanathan et al. [190] proposed *software offloading* for efficient multi-threading and latency hiding of MPI. Their idea is to allocate a thread to execute communication processing. When application threads invoke communication calls, they insert commands into a non-blocking queue, which are later consumed by the dedicated thread. Using non-blocking queues controlled by atomic operations is more scalable than guarding the resources with locks. Furthermore, offloading is effective for latency hiding because application threads requesting communication can return to the user code immediately. Application threads only need a command insertion to establish the transfer and the actual communication progress is done by the dedicated thread. Their implementation was carefully evaluated for several performance metrics but the parallelization of offloading could not be implemented because the underlying MPI runtime was not thread-safe.

PAMI [110, 111] is a low-level communication library which implements a software offloading method. PAMI uses a thread pool to parallelize communication progress and can dynamically spawn communication threads if necessary. To reduce spinning of communication threads, PAMI's offloading is depending on a mechanism called wakeup signals of POWER8 processors, which is not available in other CPU architectures.

UCX [172], a recent communication library, is composed of two layers: UCT and UCP. UCT is a low-level abstraction of hardware transports and the users of UCT need to manually select transports. UCP is constructed over UCT and provides a higher-level API than UCT. UCP provides rich functionalities such as transport selection, multi-rail communication, and software protocols which are not directly supported by the hardware.

We also confirmed that libfabric [75] can also select an endpoint for each communication request. It seems that it lacks the interface to automatically select from a set of the endpoints as UCP can. Relatively old libraries such as GASNet and ARMCI are not optimized for multi-threading because they were developed more than a decade ago when multi-core architectures were not popular.

4.2.3. User-level threading for communication libraries

As explained in Section 2.6.2, it is known that user-level threading (ULT) libraries can schedule multi-threading computations efficiently. ULT is beneficial for communication libraries because it can switch the current context into another context and realizes the efficient overlap of computation and communication. For example, Lu et al. [121] combined ULT with MPI to hide communication latency. With ULT, an application thread waiting for the completion of communication can be immediately switched to other threads that can do useful work. Their primary focus was the latency hiding and they did not utilize the parallelism of network resources.

This work is focused on combining a low-level communication library with user-level thread libraries. Since ULT is a general-purpose method for all sorts of computation and gathering attention in the HPC field [42], it is preferable to optimize low-level communication libraries for such a threading library.

```
using process_id_t = /*integer*/;
struct remote_address { size_t offset; /*...*/ };
struct local_address  { size_t offset; /*...*/ };
struct callback { void (*f)(void*); void* d; };
struct read_params {
  process_id_t   src_proc;
  remote_address src_raddr;
  local_address  dest_laddr;
  size_t         size_in_bytes;
  callback       on_complete;
};
void async_read(const read_params&);
```

Listing 4.1: API function of RDMA READ in our library.

4.3. MECOM 1: Communication software offloading on InfiniBand Verbs

In this paper, we focus on hybrid parallel programming models that utilize multi-threading in a computation node because typical user-level thread libraries require a shared address space by all of the cores within the node.

4.3.1. API of MECOM 1

Listing 4.1 shows an API function for RDMA READ in our library. Other RDMA operations such as RDMA WRITE or remote atomic operations are also defined in the same manner. Although our API is similar to those of other communication systems, we note a few minor distinctions of it here:

- All of the API functions are thread-safe. In other words, the underlying runtime is responsible for working efficiently on multi-threading environments.
- There is no "progress" function in our API. The communication progress is automatically handled using multi-threading on user-level threads.
- A completion of communication is notified via a callback function as the prefix `async_` implies. UCX also adopts a similar approach.
- We used a *fat pointer* to represent an address registered as an RDMA region. If only a pointer to the region is passed to the API function, the runtime needs to do the reverse resolution for region IDs (e.g. `rkey` in InfiniBand) and imposes an additional overhead as observed in UCP [149].

4.3.2. Design of parallelized software offloading

Figure 4.1 illustrates the general offloading architecture of our library. Although we describe only the implementation on InfiniBand in this paper, we introduce some general terms because our approach is also applicable to other architectures. There are two sorts of non-blocking queues in our design: "command queues" and "tag queues". A "command" corresponds to a work request and a "tag" corresponds to `wr_id` in InfiniBand. There are three components in the design:

- **Requesters** : Threads requesting a new inter-node data transfer. When application threads invoke communication functions, they directly enter the requester code. Instead of processing communication requests by themselves, requesters insert commands to the command queue.
- **Executors** : Threads that are monitoring the command queue and execute the delegated commands when they can be retrieved.

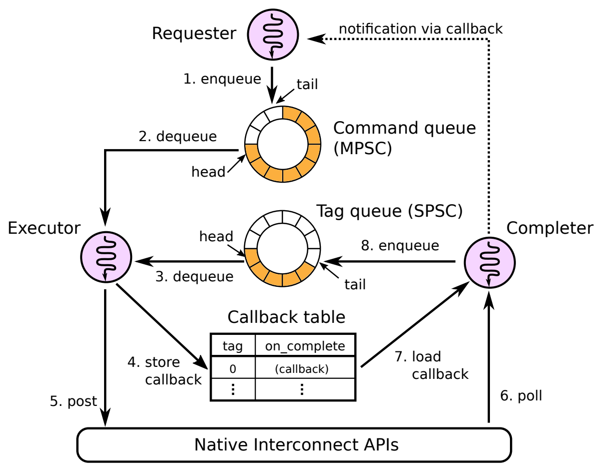


Figure 4.1.: General design of MECOM 1.

- **Completers** : Threads that call the poll function of the interconnect API. When a new completion is polled, the completer executes a callback function, which is configured by the executor.

In our current implementation, each executor has its own command queue and tag queue. Because there might be multiple application threads that are requiring communication calls, command queues are categorized as Multiple-Producer Single-Consumer (MPSC) queues. A completer communicates with an executor via its tag queue. Tag queues can be implemented as Single-Producer Single-Consumer (SPSC) queues. Tags are attached to requests by an executor and collected by a completer.

4.3.3. Implementation using a circular buffer

Listing 4.2 shows the basic data structure of our library. We used fixed-size circular buffers to implement non-blocking queues as in [190] and [110]. Fixed-size circular buffers are composed of two counters (head/tail) and an array. A callback table is also allocated as an array.

Listing 4.3 shows an implementation example of requesters. (We omit "`memory_order_`" to represent memory orders for brevity.) First, it selects an endpoint to which a new message request is posted. Our current implementation uses thread-local storage to store the endpoint ID which will be used next in each core. The endpoints are simply selected in a round-robin fashion in our current implementation.

Requesters work as producers of a command queue. Because there might be multiple requester threads that are concurrently modifying the same command queue, we use atomic compare-and-swap (CAS) to modify the tail counter of the command queue exclusively. The requester not only increments the tail counter but also sets its least significant bit (LSB). This LSB is used to represent the execution state of an executor. When the LSB is 0, it means that the executor is not running. If the requester sets the LSB, it also needs to fork a new (user-level) thread to execute the communication progress later. This behavior is different from ordinary software offloading [190] ssnd is introduced in order to dynamically control the load of communication threads.

The requester also checks that the command queue is not full; otherwise, it calls `ult_yield()` to wait until one or more entries are freed. We consider that dynamically-sized non-blocking queues allow requesters to exit from this function always immediately, but we currently employ the fixed-size scheme because of its simplicity. The flag "`vis`" is required to prevent the executor from reading the command before being written because atomically updating both the tail counter and the queue entry is impossible in most of the current CPU architectures.

We note that it is better to implement a mitigation method to avoid the contentions of CAS such as inserting an exponential backoff. Our current implementation does not support such methods but it is possible to

```
using tag_t = /*integer*/;
const size_t N_PROCS = /*...*/, N_EPS = /*...*/,
  N_CMDS = /*...*/, N_TAGS = /*...*/;

struct command {
  atomic<bool> vis; callback cb;
  // more members for several command types
};
struct endpoint {
  command cmds[N_CMDS];
  atomic<uint64_t> cmd_head, cmd_tail;
  tag_t tags[N_TAGS];
  atomic<uint64_t> tag_tail; uint64_t tag_head;
  atomic<uint64_t> n_ongoing;
  callback cbs[N_TAGS];
  // more members specific to each interconnect
  // (e.g. pointers to QP, CQ)
}
eps[N_PROCS][N_EPS];

thread_local size_t cur_ep[N_PROCS];
```

Listing 4.2: Data structure for offloading.

```
void copy_params(const read_params&, command*);

void async_read(const read_params& params) {
  // Select an endpoint
  size_t* ce = &cur_ep[params.src_proc];
  endpoint* ep = &eps[params.src_proc][*ce];
  *ce = (*ce + 1) % N_EPS;
  // Decide where a command is inserted
  uint64_t ct = ep->cmd_tail.load(relaxed);
  do {
    while ((ct >> 1) - ep->cmd_head.load(acquire) >= N_CMDS) {
      ult_yield(); // The command queue is full
      ct = ep->cmd_tail.load(relaxed);
    }
  } while (!ep->cmd_tail.compare_exchange_weak(
    ct, ((ct + 2) | 1), acquire, relaxed));
  // Place a new command
  command* cmd = ep->cmds[(ct >> 1) % N_CMDS];
  copy_params(params, &cmd);
  cmd->cb = params.on_complete;
  cmd->vis.store(true, release);
  // Fork a new thread if necessary
  if ((ct & 1) == 0) {
    ult_id_t tid = ult_fork(&executor_main, ep);
    ult_detach(tid);
  }
}
```

Listing 4.3: Implementation example of requesters.

```
void execute(endpoint*, tag_t, const command&);

void executor_main(void* p) {
  endpoint* ep = static_cast<endpoint*>(p);
  uint64_t ch = ep->cmd_head.load(relaxed);
  while (true) {
    uint64_t ct = ep->cmd_tail.load(relaxed);
    while (ch == (ct >> 1)) {
      // Try to exit if there is no new command
      if (ep->cmd_tail.compare_exchange_weak(
          ct, (ct & ~1), release, relaxed)) { return; }
      ult_yield();
      ct = ep->cmd_tail.load(relaxed);
    }
    // Wait for the executor and completer
    command* cmd = ep->cmds[ch % N_CMDS];
    while (!cmd->vis.load(acquire)) { ult_yield(); }
    while (ep->tag_head == ep->tag_tail.load(acquire)) { ult_yield(); }
    // Load and execute the command
    tag_t tag = ep->tags[ep->tag_head % N_TAGS];
    cbs[tag] = cmd->cb;
    execute(ep, tag, *cmd);
    // Recycle the command entry
    cmd->vis.store(false, relaxed);
    ep->cmd_head.store(++ch, release);
    // Fork a new thread if necessary
    if (ep->n_ongoing.fetch_add(1, acquire) == 0) {
      ult_id_t tid = ult_fork(&completer_main, ep);
      ult_detach(tid);
    }
  }
}
```

Listing 4.4: Implementation example of executors.

introduce such a method to our algorithm for highly contended situations.

Listing 4.4 shows an implementation example of executors. Because an executor is a single consumer for both the command queue and the tag queue, CAS operations are not required in the executor code. The executor thread monitors the non-blocking queue and transfers communication requests to the underlying API. If no communication request is found, the executor thread tries to reset the tail counter's LSB of the command queue. If it succeeds, the executor thread can exit and stop spinning. Once the LSB is reset by the executor, one of the other successive requesters needs to set it again. This technique requires only one atomic operation in requesters in the best-case scenario while it allows executors to notify the requesters on an exit. `ult_yield()` is also inserted in some cases to reduce the contention.

A completer thread is also forked by the executor thread. Our current implementation uses a fetch-and-add operation to synchronize between the executor and the completer. We consider that this atomic operation can also be removed if the executor does not share the completer thread with other executors.

Listing 4.5 shows an implementation example of completers. A completer thread is repeatedly polling the completion of ongoing messages. When it detects a new completion, it calls the callback function assigned by the executor. It also frees the tag to recycle for future requests. If there are no ongoing requests, the completer thread tries to exit its execution as executors do.

```
bool poll(endpoint*, tag_t*);

void completer_main(void* p) {
  endpoint* ep = static_cast<endpoint*>(p);
  uint64_t tt = ep->tag_tail.load(relaxed);
  while (true) {
    // Poll a completion
    tag_t tag;
    while (!poll(ep, &tag)) { ult_yield(); }
    // Invoke the specified callback
    callback& cb = cbs[tag];
    (*cb.f)(cb.d);
    // Recycle the tag
    ep->tags[tt] = tag;
    ep->tag_tail.store(++tt, release);
    // Exit if there is no ongoing request
    if (ep->n_ongoing.fetch_sub(1, release) == 1) { return; }
  }
}
```

Listing 4.5: Implementation example of completers.

Table 4.1.: Evaluation Environment of MECOM 1.

CPU	Intel® Xeon® E5-2680 v2
	2.80GHz, 2 sockets× 10 cores/node
Memory	16GB/node
Interconnect	Mellanox® Connect-IB® dual port
	InfiniBand FDR 2-port (only 1 port is used)
Driver	Mellanox® OFED 2.4-1.0.4
OS	Red Hat® Enterprise Linux® Server
	release 6.5 (Santiago)
Compiler	GCC 4.4.7 (with the option “-O3”)

4.4. Microbenchmark evaluation of MECOM 1

4.4.1. Evaluation setup and benchmarks

Table 4.1 shows the evaluation environment used for MECOM 1. We measured three performance metrics on this platform: the latency, the overhead, and the message rate. The bandwidth can be derived from multiplying the message rate and the message size. All of the benchmarks use 2 processes (1 process/node) and one process of them runs multiple threads which issue RDMA READ from another process. Our library is constructed on Verbs, not on AVerbs.

Listing 4.6 shows a microbenchmark program to measure the latency and the overhead. Each thread first makes a communication request and then waits for its completion by spinning on a change of the flag. The flag is set by the callback function specified in the communication call and executed by the completer. The latency is the duration $(t_2 - t_0)$ and the overhead is $(t_1 - t_0)$.

Listing 4.7 shows a microbenchmark program to measure the message rate. Each thread is continuously making communication requests and does not wait for their completions when the number of ongoing requests is below the batch size. After a while, we check the number of messages that were completed and convert it to the message rate.

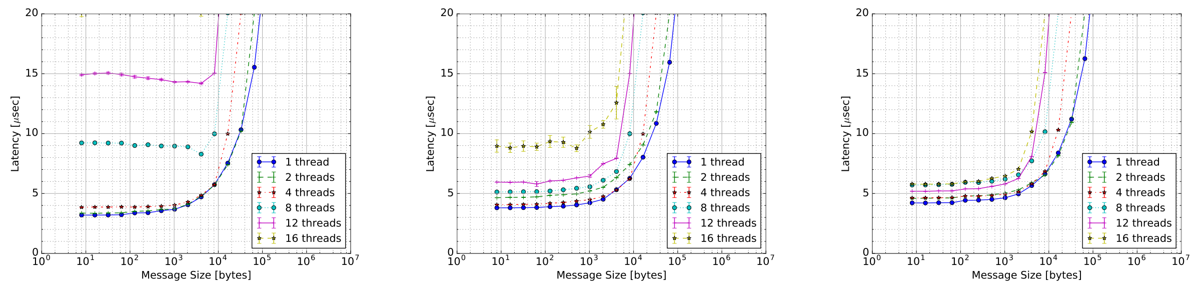
We have tested three conditions using our library:

```
uint64_t get_clock(); // returns CPU clock
atomic<bool> flag;
void callback_func() {
  flag.store(true, release);
}
// ...
while (/*...*/) {
  flag.store(false, relaxed);
  uint64_t t0 = get_clock();
  async_read(/*...*/);
  uint64_t t1 = get_clock();
  while (!flag.load(acquire)) { ult_yield(); }
  uint64_t t2 = get_clock();
}
```

Listing 4.6: Microbenchmark to measure the latency and the overhead.

```
const size_t N_BATCH = 256.
atomic<uint64_t> count;
void callback_func() {
  count.fetch_add(1, release);
}
// ...
size_t established = 0;
while (/*...*/) {
  while (count.load(acquire) - established >= N_BATCH) { ult_yield(); }
  async_read(/*...*/);
  ++established;
}
```

Listing 4.7: Microbenchmark to measure the message rate.



(a) Direct injection (b) Static offloading (c) Dynamic offloading

Figure 4.2.: Latency of RDMA READ with a single QP.

- **Direct injection**: The post function is directly called in application threads. Shared resources are guarded by spinlocks.
- **Static offloading**: There is an executor thread that is spinning on a command queue. Both the executor and the completer threads are always waking up. This corresponds to typical software offloading approaches, which dedicate some cores only for communication.
- **Dynamic offloading**: An executor thread is dynamically spawned from application threads. A completer thread is also spawned by the corresponding executor. Only this condition uses ULT.

In all three conditions, a completer is executed in a different thread from others. Both direct injection and static offloading use Pthreads to spawn threads. `ult_yield()` is replaced with `pthread_yield()` in these conditions.

We used MassiveThreads 0.97 as the underlying ULT library. We changed MassiveThreads to fork a new thread using parent-first scheduling by default in order to return from the requester functions immediately. Although MassiveThreads also has the function `myth_create_ex` to customize the scheduling policy including the usage of parent-first, this function is too slow for our usage. It is easy to add an API function using parent-first scheduling without performance loss in applications using child-first scheduling. To avoid the effects of NUMA, we configured to run only 10 worker threads in MassiveThreads in all of the experiments except for that of Figure 4.7.

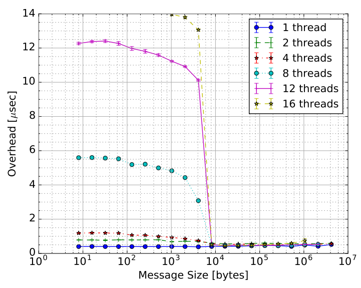
For all of the benchmark results, we measured 32 times for each graph point. Each measurement takes 5 seconds and we averaged the latency and the overhead. Error bars mean 95% confidence intervals assuming normal distributions.

4.4.2. Performance metrics with a single QP and a CQ

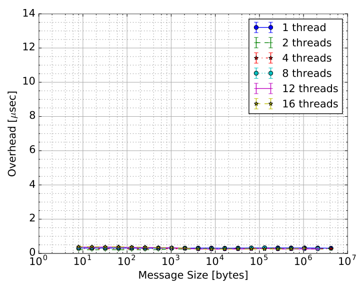
First, we measured the latency using perftest, a benchmark suite provided by OpenFabrics [146]. The latency of RDMA READ using perftest was 2.01 μsec (5628 cycles). Figure 4.2 shows the latency on InfiniBand with our library. The minimum latency with direct injection was 3.197μsec (8951 cycles) when the message size was 8 bytes. One reason for this degradation is that our runtime currently runs a polling thread different from the requester thread. We guess that a large portion of this additional latency comes the communication between intra-node cores.

When we enabled static offloading, the latency increased to 3.804μsec (10652 cycles). This additional overhead (0.670 μsec) is caused due to the cost of offloading because it requires enqueuing and dequeuing communication requests in a command queue for offloading. As observed in separating the polling thread, communication between cores lengthens the critical path of the latency. With dynamic offloading, the latency increases to 4.21μsec (11804 cycles) because it also needs to spawn a new thread for the executor if there is no executor thread at that time.

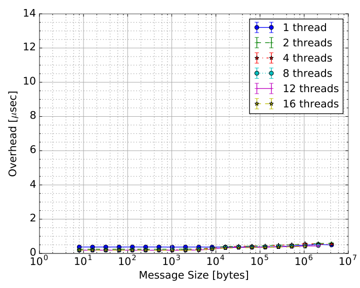
Figure 4.3 shows the overhead on InfiniBand. The minimum overhead with direct injection was 0.399μsec



(a) Direct injection

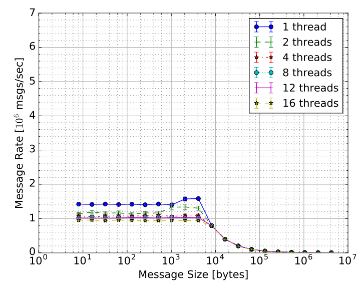


(b) Static offloading

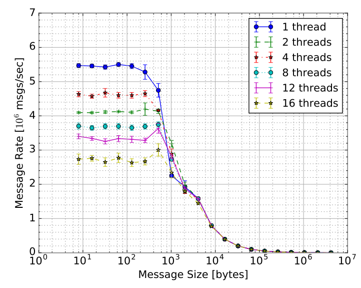


(c) Dynamic offloading

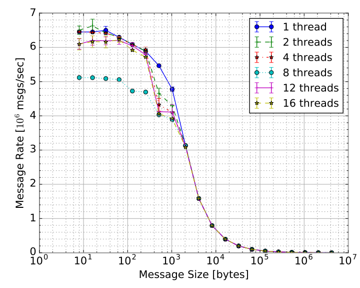
Figure 4.3.: Overhead of message injection of RDMA READ with a single QP.



(a) Direct injection



(b) Static offloading



(c) Dynamic offloading

Figure 4.4.: Message rates of RDMA READ with a single QP.

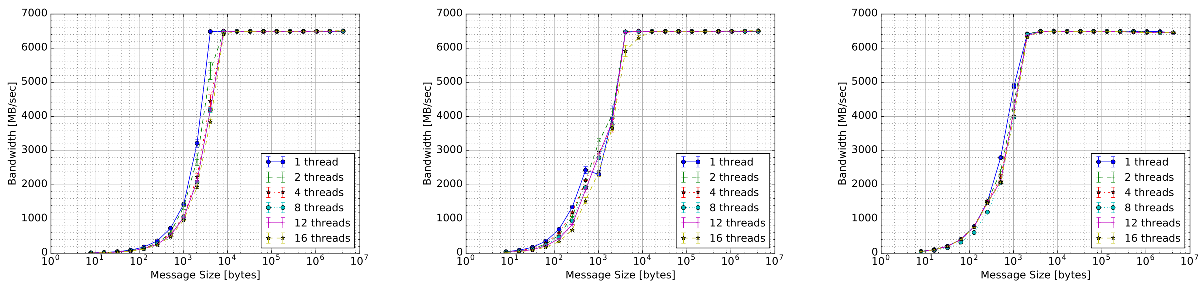
(1120 cycles). This value is comparable to UCT's result with Verbs [149]. It rapidly increases when there are many requester threads because the post function of Verbs is guarded by a spinlock. One simple solution for this is to allocate a set of QPs dominated by each core, but in general, it is not always possible due to the lack of resources.

With offloading, the overhead is almost constant regardless of the number of requester threads because communication requests are simply pushed into the command queues. The overhead was 872 cycles with static offloading and 1040 cycles with dynamic offloading. We consider that our implementation still has room to improve this overhead because it currently includes a virtual function call and other unimportant instructions. As in the latency results, dynamic offloading need more CPU cycles due to spawning a new thread.

Figure 4.4 shows the message rate with a single QP. With direct injection, the message rate was 1.404 million/sec with 8-byte messages and 1 requester thread. This result is because calling the post function of Verbs takes about a thousand cycles and becomes a bottleneck of message rates.

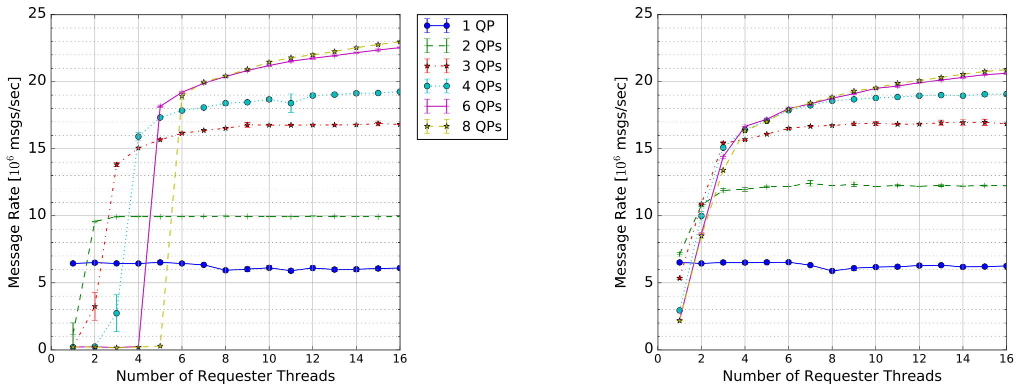
When we enabled static offloading, the message rate was improved to 5.47 million/sec with 8-byte messages. This measured rate is almost identical to that of the UCT's result over Verbs [172]. Because an executor in our system retrieves multiple communication requests from a command queue and puts them in a single call of the post function, the message rate with offloading becomes better than direct injection. UCT can boost the message rate to 14 million/sec with a single core using AVerbs, but our library does not support this acceleration now. The message rate of dynamic offloading is 6.452 million/sec, which is slightly better than the static one, but we could not figure out the exact reason for this difference.

Figure 4.5 shows the bandwidth results. Because our current implementation does not fragment large messages as UCX does [149], the peak bandwidth (6,492 MB/sec) is stable and constant regardless of the implementation methods. The effects of software overheads do not appear in bandwidth benchmarks because of fewer messages. We also checked the bandwidth using perftest was comparable to ours, which was 6,180



(a) Direct injection (b) Static offloading (c) Dynamic offloading

Figure 4.5.: Bandwidth of RDMA READ with a single QP.



(a) Fork (parent-first) (b) Condition variables

Figure 4.6.: Message rates of RDMA READ of 8-byte messages with dynamic offloading using multiple QPs (dynamic offloading).

MB/sec with 65,536-byte messages.

4.4.3. Message rates with multiple QPs and CQs

As mentioned in the previous sections, the message rates are limited by the software overheads to access hardware queues. We tried to overcome this bottleneck by parallelizing calls of both the post and poll function. Because it is not realistic to statically assign many cores for offloading threads, we evaluated this parallelization effect on the dynamic offloading approach.

Figure 4.6 shows the message rates using multiple QPs and CQs with dynamic offloading. We used a distinct CQ for each QP to maximize the parallelism in this benchmark. We prepared two types of implementations for this benchmark. “Fork” means that this implementation forks a new thread when there is no executor or completer as in Listing 4.3 and Listing 4.4. “Condition variables” means that executor and completer threads are created in program startup and wait for a *user-level* condition variable in MassiveThreads.

The results using a single QP (blue lines) are under the same condition as Figure 4.4. The message rate keeps stable with many requester threads because our library uses an efficient non-blocking scheme. With 2 or more QPs, the message rate of Fork is highly degraded when there are less than 6 requester threads. We think that this is because the workers forking new threads are out of thread resources (e.g. call stacks, thread descriptors) managed inside MassiveThreads. MassiveThreads has worker-local pools of thread resources. If the requester thread is continuously forking new threads and other workers are capable of immediately

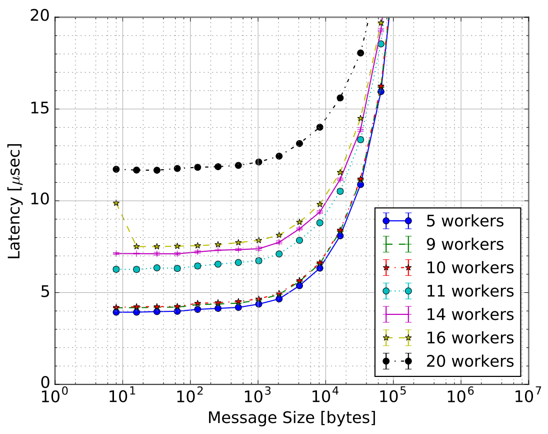


Figure 4.7.: Latency of RDMA READ with different numbers of workers (dynamic offloading).

executing them, the thread resources are collected in the pools of those workers. If a worker is running out the pooled resources, it allocates a new resource with some heavy operations. We consider that this is why the message rates of "Fork" with multiple QPs are highly degraded.

When there are a few requester threads, "Condition variables" performs better than "Fork" because it does not dynamically allocate resources but instead recycle them. We can also observe that "Fork" is more scalable than "Condition variables" when there are many requester threads. We think that thread resources are balanced among workers if there are many requesters. If the worker-local pool has enough resources, forking a thread is efficient because the thread is simply put on the worker-local thread queue.

Because our communication library does synchronization using a non-blocking scheme, the additional synchronization in the mutexes of MassiveThreads to use condition variables is unnecessary. For better scalability of non-blocking algorithms, we consider that we need an additional threading primitive that serves as a wake-up mechanism without synchronization. It is possible to implement such a mechanism in user-level threads because both saving first-class continuations and accessing worker-local thread queues can be done without acquiring locks.

The scalability of parallelizing communication processing is limited to 3 or 4 QPs (and CQs) in our experiments. We are still investigating the actual bottleneck of message rates. The bottleneck may not be the network limit but other resources such as CPUs and memory buses. As an example of other studies, PAMI on POWER8 processors and InfiniBand EDR reported a linear speed-up of message rates up to about 20 cores [109]. They disabled the multi-threading support to measure the message rates, but we focused on the message rate on multi-threading environments. Our library is capable of dynamically managing communication resources.

4.4.4. Effects of NUMA nodes

Figure 4.7 shows the latency with different numbers of ULT workers. The latency results with up to 10 workers are almost the same (about 4μsec with 8-byte messages) because they fit in a single NUMA domain. However, with 11 workers, the latency jumps to 6.268μsec. Using all of the cores from two NUMA domains degrades the latency to 11.72μsec because it allows frequent thread migrations between different NUMA nodes. When work-stealing happens, MassiveThreads picks up a victim worker evenly at random from all of the cores including those in different NUMA domains.

We are currently considering that it is not easy to solve this problem in general. Although it may be possible to implement some locality-aware work stealing methods that perform well in a certain communication system, it may degrade the performance of ordinary applications. Another solution is to prevent communication threads from being stolen by the cores in other NUMA domains. However, this may prohibit

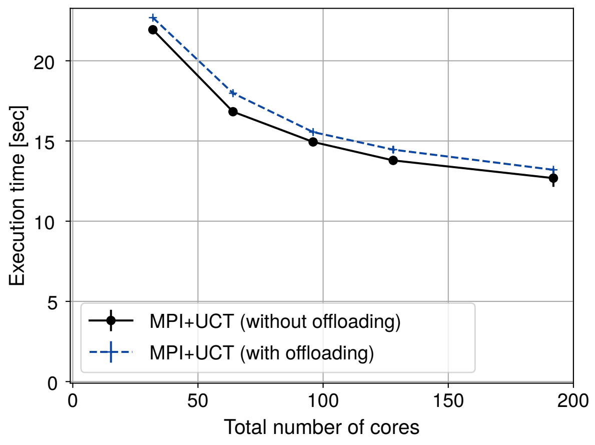


Figure 4.8.: Comparison of execution time of NAS CG (CLASS=C) with or without software offloading.

the parallel executions of communication requests and lose the opportunity to accelerate the message rates.

4.5. MECOM 2: Communication software offloading on MPI and UCX

Menps (Chapter 3) is based on MECOM 2, which is the next version of MECOM 1. The basic concepts of MECOM 1 and MECOM 2 are similar: both implement software communication offloading for multi-threading environments. Here, the technical difference between these versions is briefly described.

MECOM 1 was implemented both on MPI and InfiniBand Verbs. MECOM 2 implemented based both on MPI and UCT (the low-level module in UCX [172]). In the design of MECOM 2, we used UCT instead of Verbs in order to exploit the performance of AVerbs.

To issue communication in UCT it is necessary to allocate UCT *workers*, which manage internal communication resources including QPs and CQs. A UCT worker has a set of QPs and CQs, but it is not allowed for multiple threads to simultaneously access the QPs or CQs of a single worker. MECOM 2 creates multiple UCT workers to parallelize communications and allocate a user-level thread for every worker to execute dedicated communications as in MECOM 1. The offloading architecture of MECOM 2 is similar to that of MECOM 1, but it is implemented as the feature of ComposableThreads, which is described later in Chapter 5.

4.5.1. Evaluation of parallelized software offloading with Menps

To see the effect of parallelized software offloading, we also measured the performance of NAS CG (CLASS=C) on Menps with or without software offloading. The evaluation environment is the same as Section 3.7 and we used UCT to parallelize RDMA communications. Without offloading, every worker thread (of the user-level threading library) has its own communication resource guarded by a simple mutex. With offloading, the number of communication resources remains the same, but each resource also has a user-level thread for dynamic offloading. Figure 4.8 shows the comparison of execution time. The software offloading method presented in this chapter did not improve the performance but rather slightly degraded it. We think that the advantage of software offloading — the increase of message rates — was not affecting the major bottleneck of the DSM system. The benefit of using MECOM 2 and UCT for Menps is mentioned in Section 3.8.

4.6. Summary of MECOM

We developed a low-level communication library MECOM that implements a parallelized offloading scheme with user-level threads to achieve high-performance inter-node communication in multi-threading cluster environments. We evaluated our library using microbenchmarks and found that our method can reduce the overhead and improve the aggregated message rates using multiple hardware queues in multi-threading executions.

We also showed the problems with our current implementation. Particularly, the latency increase is a major problem of our method, but its seriousness depends on the latency tolerance of each application [125]. We also pointed out the problems of our current usage of a user-level thread library MassiveThreads, especially in NUMA environments. However, we think that dynamic offloading approaches with user-level threads are promising to solve the scheduling problems in communication systems because of its flexibility.

Our future work includes evaluations using benchmark applications. We are working on interfacing our method with higher-level programming models such as MPI. The effort of MECOM was partly utilized by Fukuoka et al. [73] to accelerate MPI communications (also described in Section 6.4.2). There are opportunities of further reduction of the latency and the software overhead.

5. ComposableThreads: a user-level threading library with compile-time parametricity

5.1. Introduction

In Chapter 3 and Chapter 4, we have seen that user-level threading (Section 2.6.2) is beneficial to develop both a DSM library and a communication library. It can be used to implement either implementing OpenMP workers on the DSM memory space or software communication offloading. However, it is difficult to make use of existing user-level thread schedulers (see also Section 6.5) because most of them are not focused on system prototyping but on accelerating HPC applications on hardware shared-memory architectures.

In this chapter, a new user-level threading library *ComposableThreads* is introduced. ComposableThreads is a header-only library written in C++ which extensively uses template meta-programming for customizability. ComposableThreads is "composable" because the users of this library can connect, insert, or replace its components for their own purposes. The zero-overhead abstraction of C++ enables us to create customizable threading libraries with minimal overhead. Each component of ComposableThreads is also designed to statically check the types utilizing C++'s features.

5.1.1. Why we need composability for threading?

We summarize here why we need to accomplish composability in the design of user-level threading libraries:

Customized locking schemes

As explained in Section 2.5, there are many variants as the implementation methods of mutexes. Instead of pursuing the universal mutex implementation, we have tried to provide the fundamental tools to build locking schemes suitable for individual purposes.

Customized scheduling strategies

One of the advantages of Argobots [171] is the customizability of scheduling policies because Argobots was not focused on efficient work stealing. Following this idea, we intend to decompose the scheduler so that it can be run with or without standard work stealing with minimal overhead.

Distributed work stealing

The customization of scheduling strategies can be further extended into distributed-memory programming. Although the current implementation of Menps does not support *distributed work stealing* (Section 6.5.3) which dynamically schedules threads on distributed-memory architecture, we have intended to port ComposableThreads into distributed work stealing.

We have been concerned about the fact that MassiveThreads/DM (or Uni-Address Threads [12]), which influenced the initial design of Menps, was a distributed work stealing scheduler written from scratch apart from MassiveThreads. This is because existing thread libraries lack composability at their design level. In general, it is hard to use those libraries to pick up some features for special purposes because the components are strongly connected with each other.

Inserting aspects

There are several cases where it is required to change the threading library behavior like aspect-oriented programming. A typical use case is event tracing, which records the whole program execution for performance analysis of task-parallel programs.

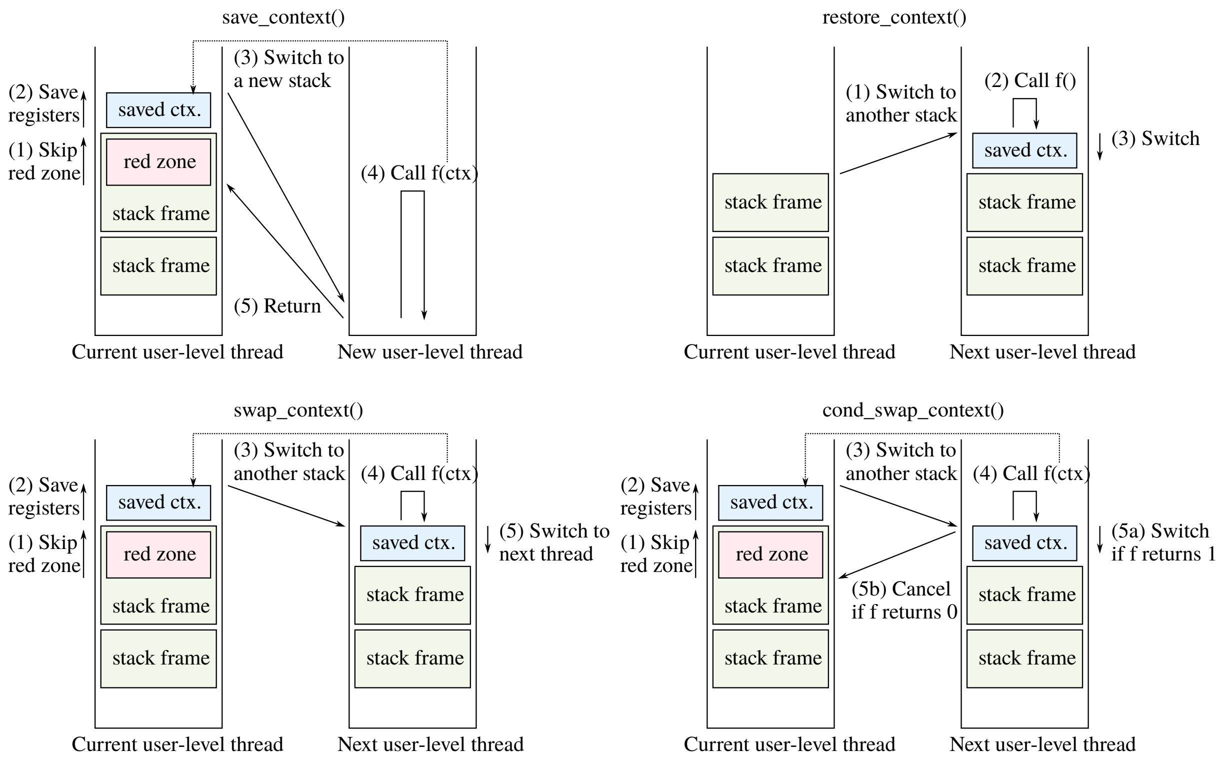


Figure 5.1.: The context switching primitives of ComposableThreads.

Customized execution contexts

Users may need to save additional registers. For example, ucontext of POSIX [185] saves the signal mask and FPU state on the x86-64 architecture for compatibility, but MassiveThreads [134] or other ULT libraries sometimes ignore them to increase the performance. Ideally, this problem should be solved at the language level, but it can be mitigated by a better library design.

5.2. Design of a composable threading library

5.2.1. Design of execution contexts

ComposableThreads has its own context switching layer. Currently, it only supports the x86-64 architecture, but because it is separated from the other parts, it can be ported to other architectures with minimal effort.

Figure 5.1 shows the context switching of ComposableThreads. The important difference from ordinary context switching implementations is that these functions *always* take a user-defined function called on top of the call stack of another thread. This is similar to `callcc()` (*call with current continuation*) of Scheme programming language [169], but this context switching is a more low-level functionality for efficient threading. The idea itself, calling the function on another user-level context in C/C++, is not a novel thing (e.g. `ontop_fcontext()` in Boost.Context [106]), but in our observations, it is *unnecessary* to implement the switching functions which do not take the function argument. This is because context switching (= taking the current continuation) and other operations with the continuation should be atomically done. Although it is possible to guard the resources for saving continuations via mutexes, for example, such a method degrades the performance because it requires hardware atomic instructions by nature.

In detail, there are five context switching primitives:

- `save_context()` saves the current context and calls the specified user-defined function with the saved

Table 5.1.: Support of ontop context switching functions in user-level threading systems.

ComposableThreads	`save_context()`	`swap_context()`	`restore_context()`
MassiveThreads [134]	Not supported[1]	`myth_swap_context_withcall()`	`myth_set_context_withcall()`
Boost.Context [106]	Not supported	`ontop_fcontext()`	Not supported
Argobots [171]	`init_and_call_fcontext()`	Not supported	Not supported
libfibre [25]	`stackDirect()`	`stackSwitch()`	Not supported

Table 5.2.: Support of non-ontop context switching functions in user-level threading systems.

ComposableThreads	`make_context()`	`restore_context()` + empty func.	`swap_context()` + empty func.
MassiveThreads [134]	`myth_make_context_voidcall()`	`myth_set_context()`	`myth_swap_context()`
Boost.Context [106]	`make_fcontext()`	Not supported	`jump_fcontext()`
Argobots [171]	`make_fcontext()`	`take_fcontext()`	`jump_fcontext()`
libfibre [25]	`stackInit()`	Not supported	Not supported

context on a new call stack area. If this function normally returns, the previous context is resumed. This function is mainly used to implement thread creation.

- `restore_context()` abandons the current context and calls the specified user-defined function on top of the next thread's context.
- `swap_context()` saves the current context and calls the user-defined function with the saved context on top of the next stack frame.
- `cond_swap_context()` saves the current context and calls the user-defined function as in `swap_context` first. The difference is that it can change the next execution path by the return value of the user-defined function. In other words, this function *conditionally* swaps the context depending on the function result.
- `make_context()` (missing in Figure 5.1) prepares a new context for the resumption. This function does not switch the current context by itself and returns the newly created context.

Table 5.1 and Table 5.2 show the comparison of context switching functionalities in different threading systems. For "ontop" functions, which take user-defined functions executed on top of another call stack, other libraries may or may not support the functions provided by ComposableThreads. In addition, as far as we know, no other threading systems support `cond_swap_context()`, which is useful for implementing mutexes on user-level threading described later. ComposableThreads does not support non-ontop functions without a function argument, but they can be emulated by passing a function which does nothing in its body.

5.2.2. Design of workers

Based on the context switching primitives, *workers* are introduced. A worker is a component that executes runnable tasks. A worker is associated with its own real thread (e.g. a Pthreads' thread) which is running the scheduler loop function.

Internally, a worker class is composed of four data members:

- **A task deque** to hold runnable tasks.
- **A running task** that is currently being run by the worker itself. When a worker class is initialized, this running task is set to a pseudo task representing the scheduler context.
- **A scheduler context** that represents the context of the scheduler loop function.
- **A (real) thread-local storage** to get the current worker. Most of the methods of the worker class need to be called from the same thread and their instances can be checked at runtime. There are also functions called remotely including the thief function to steal threads.

[1]This is achieved via myth_make_context_empty() + myth_swap_context_withcall(). This functionality is important for MassiveThreads because it employs a work-first policy.

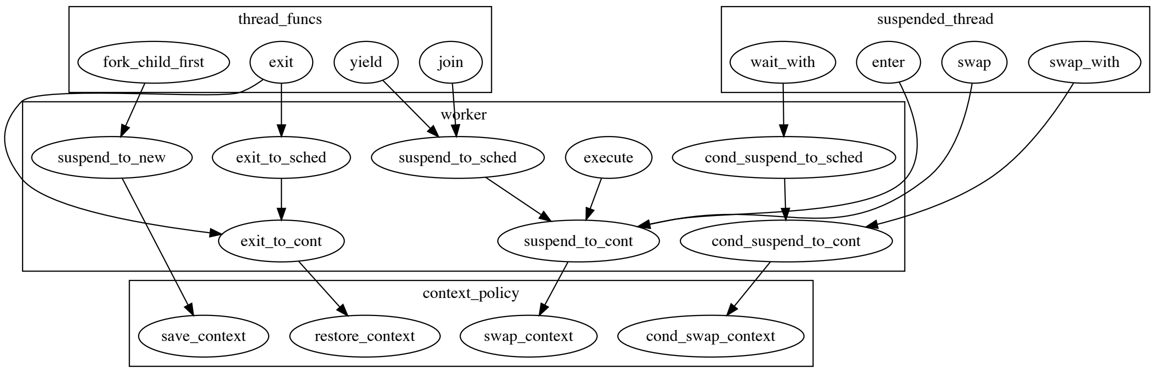


Figure 5.2.: Dependency graph of ComposableThreads' functions using context switching.

Because the worker itself does not need to aware of the underlying real thread, it is not included as a data member.

Figure 5.2 shows the dependency graph of the functions using context switching. A set of context switching primitives of Figure 5.1 is provided as `context_policy`, which can be replaced with a different "policy" class for customized register saving strategies. With these primitive functions, the worker class implements several functions:

- `wk.suspend_to_new()` suspends the current context and calls the specified function at the specified call stack. The saved continuation is passed as an argument of the specified function. This call stack is converted to a thread, which is registered as the running task of the worker. This method internally calls `save_context()`.
- `wk.exit_to_cont()` abandons the current context and calls the specified function at the specified continuation. After the specified function returns, the thread of the specified continuation is resumed and registered to the worker as its running task. Because this function does not save the current context, it is mapped to `restore_context()`.
- `wk.suspend_to_cont()` suspends the current context and calls the specified function at the specified continuation. This is implemented on `swap_context()`.
- `wk.cond_suspend_to_cont()` suspends the current context, calls the specified function at the continuation, and then conditionally resumes to the continuation. This method uses `cond_swap_context()` for conditional context switching.

These function calls can be used for implementing the other functions:

- `wk.execute()` resumes the specified continuation from the scheduler thread. This method calls `wk.suspend_to_cont()` with the function which stores the previous context to the scheduler context.
- `wk.*_to_sched()` first removes the scheduler context and calls `wk.*_to_sched()` with the continuation of the scheduler context.

The basic features of user-level threading, including `fork`, `exit`, `yield`, and `join` can be implemented with these primitive functions. Because the worker implements the core of user-level threading cleanly, these derived functions can be implemented as simple wrapper functions.

5.2.3. Design of suspended threads

Suspended threads is a low-level functionality of user-level threading which we have introduced. A suspended thread is a first-class move-only object and it can be, for example, passed as a function argument. The default constructor initializes it as the empty state, which is not associated with any thread. When the method of a suspended thread saves the continuation of the current thread, it transitions to a non-empty state associated with the continuation.

The methods of a suspended thread `sth` are described as follows:

- `sth.wait_with()` suspends the current thread and saves its continuation to the suspended thread. It takes a user-defined function which is invoked on top of the call stack of the resumed thread with the specified pointer arguments. The returned boolean from the function instructs whether the worker actually saves the continuation or cancels it. If it is canceled, the suspended thread remains empty.
- `sth.notify()` resumes the continuation associated with the suspended thread. It will put the continuation into one of the worker deques, which is probably owned by the caller's worker. `sth.enter()` also resumes the continuation and works similarly as `sth.notify()`, but it differs in the actual behavior because it saves the continuation of the current thread, put it into the deque, and then resumes the continuation associated with `sth`. Both `sth.notify()` and `sth.enter()` are interchangeable if the performance is not concerned.
- `sth.swap_with()` first suspends the current thread and saves its continuation as `sth.wait_with()`. It also resumes the continuation associated with the suspended thread. The suspension can be canceled in the same way as defined in `sth.wait_with()`. `sth.swap()` is an alternative of `sth.swap_with()` without the specified function.

The observer function returns the state of the suspended thread. If a continuation is associated with the suspended thread, the observer returns `true`.

5.2.4. Design of mutexes and condition variables

In ComposableThreads, mutexes and condition variables have been implemented on suspended threads. In addition to ordinary methods such as `lock()` and `unlock()`, ComposableThreads' mutexes introduce `unlock_and_wait()` which saves the continuation to a suspended thread while it also unlocks the mutex.

5.2.5. Design of lock delegation

Chapter 4 discussed that user-level threading is beneficial for the implementation of communication software offloading. ComposableThreads generalizes this idea as a lock delegation class that can be applied to different purposes. A delegator is a class that synchronizes between producer and consumer threads for a shared resource.

The producer methods of a delegator `d` are described as follows:

- `d.lock_or_delegate()` tries to lock the delegator first. If it succeeds, it will immediately return to the caller returning `true`. Otherwise, it calls `delegate_func` to delegate a function to the consumer and returns `false`. `delegate_func` can return a pointer to a suspended thread where the continuation of the caller thread is saved. If this returned pointer is null, the continuation is not saved.
  If lock acquisition succeeds, the caller thread needs to unlock the same delegator after it completes its critical section. If not, because the caller thread is not locking the delegator, it shall not unlock it either.
- `d.execute_or_delegate()` first tries to lock the delegator as `d.lock_or_delegate()` does. If it succeeds, it executes `imm_exec_func` on the same thread, unlocks the delegator, and returns `true`. If not, `delegate_func` is called for delegation and returns `false`. This method is convenient when a caller can encapsulate the critical section as a function object.

- `d.lock()` waits for the lock acquisition of a delegator as in a normal mutex and `d.unlock()` releases the lock. `d.unlock_and_wait()` works in the same way described in the mutex interface. With these methods, it is apparent that delegators simply extend the interface of mutexes.

For a consumer `c`, a consumer thread can be started by `c.start_consumer()` and stopped by `c.stop_consumer()`. `c.start_consumer()` takes a consumer object constrained by the concept `delegate_consumer`. It is required for the consumer object to having the following methods:

- `c.is_active()`: This method returns `true` if the delegator needs to invoke the progress of the consumer.
- `c.progress()`: The progress function of the consumer. The return value of this method is a suspended thread which is resumed by the delegator later. Returning this suspended thread may improve the performance because the delegator thread can directly switch to the returned thread.
- `c.execute()`: In this method, the consumer executes the delegated function specified as the argument. The first element of the return value is `true` if the delegated function is executed by calling `c.execute()`. Otherwise, the delegator will try to execute the delegated function again later. The second element of the return value means the same as described `c.progress()`.

5.3. Implementation of ComposableThreads

The current implementation of ComposableThreads is divided into multiple levels. Basically, the classes at a low level do not have the dependencies on the upper levels.

- Level 0: Spinlocks, atomics
- Level 1: Task descriptors, context switching
- Level 2: Worker deques
- Level 3: Workers
- Level 4: Memory pools
- Level 5: Schedulers, threads, suspended threads
- Level 6: Mutexes, condition variables, delegators, parallel for-loops
- Level 7: Barriers

5.3.1. Implementation of mutexes and condition variables

ComposableThreads implements mutexes as queue-based locks (Section 2.5). Currently, it only supports MCS locks [129], but it can be extended to support other locking schemes. Barriers are currently implemented on top of condition variables.

5.3.2. Implementation of delegators

The main difference of software offloading technique in Chapter 4 from other similar work was introducing a user-level thread for each resource (= mutex) as a helper to execute critical sections. It is better to use a helper thread when lock acquisition is contended because:

1. If the next acquirer thread needs to execute the critical section, it always needs to wait for the current lock owner.
2. If the owner thread needs to execute the critical sections of other succeeding threads, the owner thread cannot return to its execution and causes starvation.

Introducing a helper thread naturally solves this problem by invoking polling on the helper thread. This approach is feasible in the environments with user-level threading (or cooperative schedulers) because it is

Table 5.3.: Evaluation environment for ComposableThreads.

CPU	Intel Xeon CPU E5-2699 v3
	2.30 GHz (max. 3.60 GHz with Turbo boost)
	2 threads × 18 cores × 2 sockets
Memory	661 GiB
OS	Ubuntu 18.04.3
Compiler	GCC 7.4.0

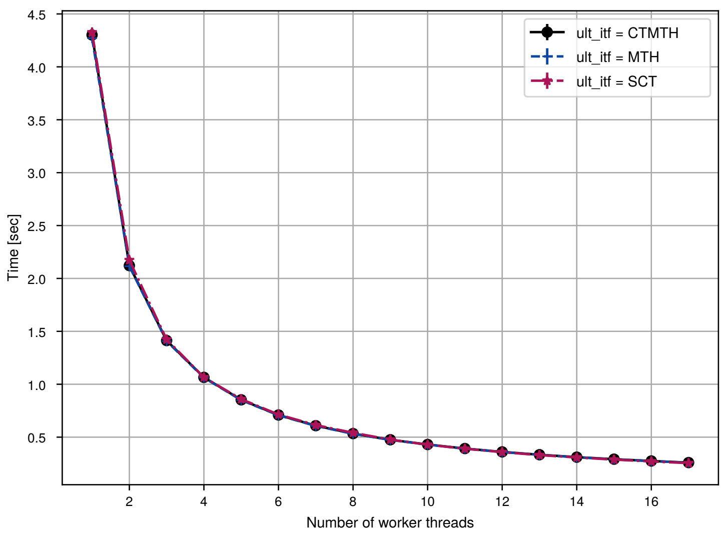


Figure 5.3.: Execution time of fib(37).

based on oversubscription of threads.

The implementation of ComposableThreads' delegators is derived from the software offloading technique of MECOM 1, but is different from MECOM 1 in terms of the following technical points:

1. It can select one of two implementations: list-based (queue-based) or circular buffer-based implementations. Queue-based implementations can support fair lock arbitration as mentioned in Section 2.5. It is also possible to switch to circular buffers which were used in Section 4.3.3.
2. If the lock acquisition does not contend, the critical section is executed immediately. In MECOM 1, it always delegates a communication request regardless of the lock state. This frequent delegation causes too many inter-core communications.

5.4. Microbenchmark Evaluation

In the evaluation, we have compared three instances of threading libraries: SCT, MTH, and CTMTH. SCT is the ComposableThreads-only mode (Shared-memory ComposableThreads). MTH means MassiveThreads. CTMTH is almost MassiveThreads, but the implementation of both mutexes and barriers are replaced with those of ComposableThreads (this is one of the technical strengths of this library; it can be composed on top of the existing system). Table 5.3 shows the evaluation environment for ComposableThreads.

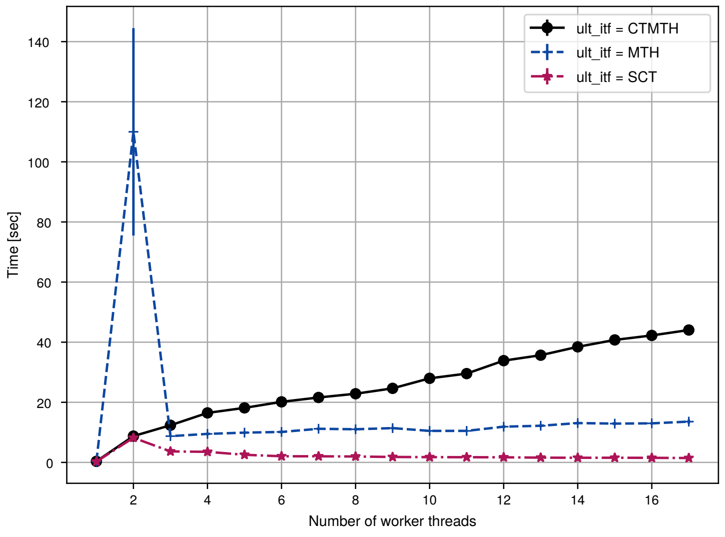


Figure 5.4.: Execution time of mutex benchmark.

5.4.1. Overheads of thread creation

The *fib(n)* microbenchmark calculates the n-th Fibonacci number in parallel to measure the performance of the primitive features of task schedulers: task creation and destruction. Figure 5.3 shows the performance comparison of fib(37) using ComposableThreads with other configurations. It has been confirmed that the evaluation results of all of these three libraries have almost the same.

5.4.2. Mutexes and barriers

Figure 5.4 compares the execution time of the benchmark to measure the performance of a contended mutex. In this microbenchmark, there are 50 user-level threads that are repeating locking and unlocking the same mutex. The benchmark finishes when all of the threads executed the critical section 200,000 times.

From this evaluation result, SCT performs the best with varying numbers of threads. The performance of original MassiveThreads' mutexes strangely works slowly with two threads, which is due to its spinlock-based implementation. There is still an unknown issue in this figure; the performance of CTMTH is worse than SCT. The ComposableThreads' mutex is implemented on suspended threads and suspended threads of CTMTH are implemented on *uncond* objects (see also Section 6.4.2). The current mutex implementation is optimized for the SCT's suspended threads, but may not be for uncond objects through CTMTH's interface.

Figure 5.5 compares the execution time of a barrier benchmark. In this microbenchmark, 50 user-level threads repeat the barrier 300,000 times. SCT generally performs better than MTH because of its queue-based locking approach. However, with small counts of threads executing barriers, MTH and CTMTH perform better than SCT. The performance of CTMTH is not comparable to MTH with many threads due to the current issue of the mutex implementation.

5.5. Summary of ComposableThreads

ComposableThreads is a user-level threading library with compile-time parametricity. ComposableThreads is implemented as a C++ header-only minimal library that extensively uses zero-overhead abstraction and static type checking for threading. In this work, we focused on how we can decompose the design of threading libraries so that the users can extract the components of this library. This library was initially designed to implement MENPS and MECOM, but it can be generalized for other purposes which need the

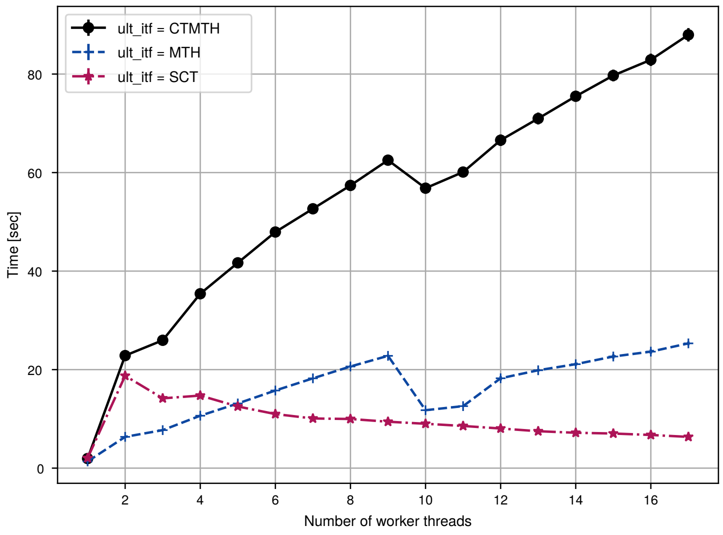


Figure 5.5.: Execution time of barrier benchmark.

flexibility of threading. In the microbenchmark evaluation, it has been confirmed that the performance of ComposableThreads was comparable to that of MassiveThreads.

Because ComposableThreads is a research prototype system developed for Menps, it implements only a small set of user-level threading features. To this library feasible for other purposes, it is required to implement other missing features and evaluate with other applications. Implementing tasklets (e.g. [171, 94]) or other acceleration techniques with the primitives of ComposableThreads may also be interesting future work.

6. Related work

6.1. Distributed Shared Memory (DSM)

6.1.1. Traditional DSM systems

There had been a number of studies related to DSM systems until the 1990s. Many DSM systems were developed either on hardware or software (or both), but they could achieve the limited scalability (roughly less than 100 cores). Traditional DSM systems could ignore the complexities of today's computing environments such as multi-core architectures (hybrid parallelization) and low-latency interconnects. In contrast, Menps assumes the modern hardware architectures in its design and can utilize such resources for accelerating coherence actions.

The first software DSM system is Ivy [119, 120], which introduced page-based access control to handle cache misses in software (Section 3.4.1). The idea of probable owners was also introduced in Ivy. At that moment, because the research of consistency models was in an early stage, Ivy used sequential consistency as its consistency model, which prohibits memory reordering for latency hiding (Section 2.4.1).

The concept of release consistency was first published in a hardware DSM system DASH (Directory Architecture for SHared memory) [74, 118]. In DASH, the main motivation of introducing release consistency instead of sequential consistency was to reduce the effects of write latency. On memory writes, the DASH protocol pipelines write messages to the shared data (the shared cache) and it does not wait for their completions until synchronization points (= release fences), which can effectively hide the write latency.

The first software-based implementation of release consistency is Munin [41]. It is not realistic to implement the coherence protocol of DASH on page-based DSM systems because DASH issues memory transfers for every write operation. In page-based DSM systems, it is significantly costly to detect all of the writes issued by an application program due to the inherent overhead of the page protection mechanism. Munin alleviates this problem by delaying the transfers and aggregating the write messages until synchronization points. In order to delay the writes, Munin first introduced the diff approach (Section 3.4.1) to merge the writes from multiple processes. On a synchronization point, Munin broadcasts the diffs to all of the sharer processes and the diffs are applied on the sharers. Figure 6.1 shows how the Munin's protocol broadcasts diffs to all of the sharers at fences. Compared with the home-based protocol (Figure 3.7), it is observed that the same diffs are applied in multiple processes.

TreadMarks [102, 103] is a famous example of traditional software-based DSM systems. The protocol of TreadMarks is named lazy release consistency (LRC). Lazy release consistency itself is *not* a consistency model but the implementation of cache coherence because its consistency model is exactly equal to release

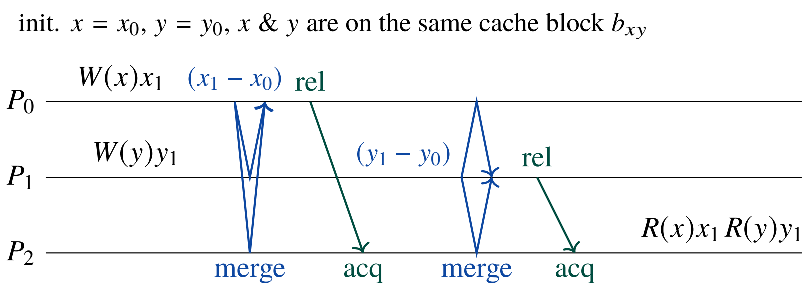


Figure 6.1.: A false sharing situation in the homeless eager protocol employed in Munin.

consistency. The protocol of Munin is named Eager Release Consistency (ERC) in the TreadMarks' paper focusing on the difference of when the diffs are applied. ERC executes the diff application until a release fence, but LRC delays until the next load operation which actually requests the cache block. LRC can reduce the write latency but lengthens the read latency because the reader processes need to accumulate all of the latest diffs from the writers on cache misses.

It is hard to implement lazy release consistency protocols using RDMA because the reader process cannot request the releaser to apply diffs via a message. The core idea of lazy diff application cannot exactly capture the benefits of coarse-grained zero-copy memory transfers because the emergence of readable memory blocks is as delayed as possible. Due to these problems, Menps is designed as an eager protocol for RDMA-based coherence and is rather focused on processing diffs locally and immediately.

TreadMarks can also be characterized by other features related to its coherence actions. As described in Section 3.4.4, TreadMarks introduced write notices, which is considered as a directory-less invalidation method. Menps effectively uses write-notice invalidation to implement the fast read method to fetch the memory block with a single RDMA operation (Section 3.4.5). TreadMarks used vector timestamps [24] to correctly track the partial ordering between processes and manage the internal data such as write notices and diffs. Vector timestamps are useful to reduce the communication traffic in DSM systems, but they come with a large storage cost proportional to the number of processes [47]. On the other hand, logical timestamps (or Lamport clocks [115]) is a fixed-size small data structure and can be used to roughly order the coherence actions. One of the major problems of TreadMarks is the necessity of global garbage collection, but this problem is solved in Menps by its decentralized design.

Zhou et al. [204] introduced home-based lazy release consistency (HLRC) to process diffs. The idea of home-based multiple-writer protocols has been described in Section 3.4.1. The study of HLRC is motivated by the protocol called Automatic Update Release Consistency [88] which used the hardware assistance of the SHRIMP multiprocessor to propagate diffs. AURC has several advantages over the previous work including LRC mainly because it allocates a home node for each cache page to aggregate diffs. AURC does not use diff processing and the storage costs for diffs can be eliminated. The home node can directly access the cached pages without page faults and other nodes can fetch the pages in a single round-trip message. However, AURC depended on the special hardware support and degrades portability. HLRC behaves similarly to AURC, but HLRC creates "temporary" diffs when the succeeding acquire operation happens, transfers them to the home node, and discards them immediately. HLRC suffers from the software diff processing overhead, but it can mitigate the storage problem of LRC. Menps is considered as home-based protocols because of these advantages, but its floating home-based approach is different from the original HLRC because Menps does *not* transfer run-length encoded diffs and rather transfer the whole block to exploit the performance of RDMA.

Chung et al. [52] proposed a coherence protocol called Moving Home-based Lazy Release Consistency (MHLRC). The concept of MHLRC is close to that of the floating home-based method of Menps because MHLRC proactively migrates the home nodes to reduce the diff transfer overhead. Because MHLRC can choose whether the home is moved or not, it seems that the floating home-based method simply limits the existing coherence protocol and suffers from the overhead of frequent home migration. However, as shown in Section 3.10.2, frequent page migration does not become a significant bottleneck in the current hardware setting. Compared with MHLRC, the floating home-based approach still takes advantage of moving the home nodes but can be purely implemented in RDMA without message handlers because of its simplicity.

SCASH [80] is a page-based DSM system based on Release Consistency. SCASH is a home-based DSM system similar to JIAJIA and employs directory-based coherence (Section 3.4.3). SCASH also supported a dynamic home reallocation mechanism (= home migration) which can dynamically change home nodes to adapt application behaviors. As explained in Section 3.4.1, if the DSM system can migrate a home node during execution, the metadata for the home node placement must be carefully kept coherent in all of the nodes. SCASH introduced a *base* node that always knows the latest home node and is notified when the home is reallocated. They have reported that their proposed home reallocation method improved the performance of SPLASH-2 LU compared with the default round-robin strategy.

Table 6.1.: Comparison of software DSM systems.

System	Consistency	SW/MW	Home	Eager/Lazy	Home migration
Ivy [119, 120]	Sequential	SW	N/A	N/A	N/A
Munin [41]	Release	MW	Homeless	Eager	N/A
TreadMarks [102, 103]	Release	MW	Homeless	Lazy	N/A
HLRC [204]	Release	MW	Home-based	Lazy	No
JIAJIA [85]	Scope	MW	Home-based	Lazy	No
SCASH [80]	Release	MW	Home-based	Eager	Yes
Orion [136]	Release	MW	Home-based	Eager	Yes
Samhita [158]	Regional	MW	Home-based	Eager	No
Argo [100]	Data-race-free	MW	Home-based	Eager	No
MENPS	Data-race-free	MW	Home-based	Eager	Yes

The authors of SCASH's paper did not explain clearly how SCASH determines whether the cached metadata which may point to the old nodes is still valid. Even though SCASH supported home migration, it is apparent that SCASH was based on centralized data structures such as cache directories for sharer tracking and base nodes for home tracking. In contrast, MENPS employs the decentralized methods for both cache invalidation and home migration.

Table 6.1 shows the comparison with existing software DSM systems. It can be observed that relatively recent systems (downward in the table) are converging to home-based eager protocols as in MENPS.

6.1.2. RDMA-based DSM systems

The first attempt to implement a DSM system using RDMA (Section 2.3.1) was done by Rangarajan et al. [159]. They implemented home-based lazy release consistency (Section 6.1.1) on top of Virtual Interface Architecture (VIA) [63], which was an available interconnect architecture with RDMA at that time. Because the real VIA implementation did not support RDMA READ operations, they only focused on exploiting RDMA WRITE for software DSM. Although their implementation used RDMA WRITE operations to accelerate the transfers of reply/response messages, their method still heavily depended on message-based communications.

Iosevich et al. [90, 91, 92] have compared SC/MV (sequential consistency / multiple view [93]) and home-based lazy release consistency. Their DSM system was also implemented over VIA. Their implementation method is based on *bins* [31], which accumulate write notices in a coarse-grained manner using vector timestamps. Bins can be transferred either via messaging or RDMA. Diffs can be transferred either in one of the two ordinary methods: PackDiff and DiscontigousWrite (described in Section 3.2). They have reported that RDMA accelerated the transfers of bins in most of the benchmark applications, but does not improve diff processing. This result corresponds to the result of Section 3.2, which indicates that PackDiff generally outperforms DiscontiguousWrite.

NEWGENDSM [141, 143] is a software DSM system that utilized RDMA for acceleration in the early age of InfiniBand. NEWGENDSM is based on two implementation methods: ARDMAR (Atomic and RDMA Read) and DRAW (Diff with RDMA Write). ARDMAR is an RDMA-based method to assign the home process of a cache block. Although ARDMAR has similarity to the floating home-based approach in MENPS such as the use of RDMA atomic operations, ARDMAR can only decide the initial home placement dynamically and does not implement dynamic home migration. On the other hand, the floating home-based approach solves a harder problem than ARDMAR because it can dynamically migrate the home and avoid centralization due to the fixed home. DRAW is the same method as DiscontiguousWrite in Section 3.2, which we have avoided to employ in the protocol design of MENPS due to its inherent problem of software overhead.

The similar approach to NEWGENDSM has been conducted by Eichner et al. [65]. The former work of NEWGENDSM [142] proposed a protocol called PIPE, which implements PackDiff (described in Section 3.2) using RDMA WRITE operations.

ViSMI (Virtual Shared Memory for InfiniBand clusters) [147] is also a software DSM system that utilized InfiniBand in the 2000s. Although ViSMI uses RDMA for several operations in its coherence protocol, ViSMI uses a request/reply model (see also Section 2.1.2) and is not a fully RDMA-based system. ViSMI is based on hardware multicast of InfiniBand, which is only available in SEND/RECV operations with the UD service type. Compared with ViSMI, to fully exploit the performance of RDMA, Menps does neither depend on messaging nor multicasting in the design of its coherence protocol.

Argo [100] is a rare example of recent DSM systems and has inspired the design of Menps. Argo is implemented as a pure RDMA-based DSM without message handlers. In Section 3.8.3, the performance between Argo and Menps has been compared. Menps has several important distinctions from their proposal. Although Argo implements an invalidation method called P/S3 classification, their method still depends on centralized directory structures. Argo is implemented on MPI (particularly on MPI-3 RMA), which is not always mapped to the actual RDMA operations [174]. Menps is based on the decentralized schemes and each communication is exactly mapped to a single RDMA operation.

Hong et al. [84] implemented an RDMA-based multithreaded software DSM system MAGI, which employs sequential consistency as its consistency model. They implemented a single-writer protocol (Section 3.4.1) with directory-based coherence (Section 3.4.3). As explained in Section 3.4.1, single-writer protocols cannot be implemented without the assistance of remote CPUs on previous writer nodes. Because their method cannot be implemented purely with RDMA, it runs a protocol thread on each node to process requests from remote nodes.

The authors of MAGI proposed several methods to improve the original protocol of Ivy [119, 120]: speculative faults, batched TLB shootdown, polling-based request handling, and protocol bypassing. Speculative faults is a similar technique to prefetching. Batched TLB shootdown aggregates system calls to reduce the effect of TLB shootdown. Polling-based request handling is a similar idea to software offloading (Chapter 4), which dedicates cores for communication handling. Protocol bypassing adds an API to directly read the remote memory ignoring the requirement of sequential consistency. All of these techniques may be applied to the MENPS' protocol because they are orthogonal.

Exploiting the RDMA performance in DSM systems is one possible form of hardware communication acceleration. As an example of hardware acceleration, Bianchini et al. [29] proposed a method to use a programmable protocol controller to accelerate diff processing in DSM systems. Automatic Update Release Consistency (AURC) [88], which affected the design of home-based lazy release consistency [204], used coprocessors to propagate diffs. The idea to program communication hardware still exists even in modern studies [83, 60]. Because Menps is designed to match the current hardware limitations, such hardware customization is orthogonal to this work, but some of the proposed methods of Menps may be replaced with customized hardware accelerations.

6.1.3. Object-based consistency models

Release Consistency (Section 2.4.2) is one of the relaxed consistency models that can both effectively relax the memory ordering for performance and provide a memory view to be easily understood. However, the constraints of release consistency are sometimes "too conservative" because it requires synchronization of the whole memory. To reduce the number of the synchronized variables, *region-based* or *object-based consistency* models have been studied. In these consistency models, the memory system synchronizes only the words, variables, or objects which are associated with a synchronization object (e.g. a mutex). They can reduce the number of invalidation requests or write notices (Section 3.4.4) compared with release consistency, which synchronizes the whole memory space based on happens-before partial ordering. In other words, memory access in region-based consistency is associated with a synchronization object, but memory access in release consistency is associated with a processor (or a process).

Midway [28] is an example of object-based DSM systems that introduced the consistency model called *entry consistency*. Entry consistency requires explicit binding of variables to synchronization variables beforehand, and at synchronization events, only the specified variables are guaranteed to be consistent. C Region Library (CRL) [97] is another example of object-based DSM systems similar to Midway, which is implemented entirely in software ("all-software DSM") because it does not depend on page-based access control. All of these variants require explicit annotations from programmers and degrade the application productivity even though they can provide some performance improvements because of reducing unnecessary invalidations.

Scope consistency [87] is a consistency model derived from both release consistency and entry consistency. Scope consistency implicitly binds objects to synchronization variables and reduces unnecessary invalidation. Scope consistency does not require explicit annotations in entry consistency because the binding is implicitly detected by the DSM system using a page-based solution.

In Menps, we did not try to further relax the memory model of existing data-race-free models because we considered that the relaxation of consistency models is not the major problem for current DSM systems. The coherence methods of Menps are orthogonal to these further relaxed models. For example, although scope consistency can reduce the number of write notices because only writes to associated variables are managed, it still cannot solve the problem of increasing write notices infinitely. The decentralized coherence actions of Menps will be beneficial for these models if applied.

DAG Consistency [33] is a consistency model based on task-based programming models (Section 2.6.2). DAG Consistency is conceptually similar to what scope consistency or other object-based approaches tried to solve because DAG Consistency implicitly binds variables to tasks rather than to processors. SilkRoad [151, 153, 152] is a DSM system that combined both release consistency and DAG Consistency. These two models can be naturally combined because they are based on partial ordering between threads.

6.1.4. Compiler-based DSM systems

As described in Section 3.4.1, Menps is classified as a page-based DSM system in terms of software-based transparent access control. Using the page protection is one of the available choices. The advantage of page-based DSM systems is the elimination of overheads on cache hits. However, they suffer from kernel overheads on cache misses because controlling the page protection requires system calls.

Another approach to transparently control memory accesses is compiler-based DSM [168], which replaces memory accesses by a special compiler. Compiler-based DSM uses compilation techniques to replace the memory accesses of the original program in order to embed access control codes. Because it is difficult to implement compiler-based DSM due to the necessity of compiler modifications, this dissertation focused on the page-based approach. However, with the help of LLVM [6], a new compiler-based system may be easily implemented into modern compiler infrastructure.

Hu et al. [86] implemented a translator that converts OpenMP directives to appropriate calls of the modified version of TreadMarks. Their system was the first attempt to execute OpenMP programs on software DSM systems. They implemented a translator from OpenMP programs into programs using TreadMarks. They reported that the shared-memory support of TreadMarks eased the implementation of their OpenMP translator.

Omni-compiler [165, 166, 145] is an OpenMP compiler which translates OpenMP programs to the programs executable on SCASH [80] (Section 6.1.1). Because Omni-compiler transforms every global variable declaration for SCASH, it is unnecessary to annotate the global variables of the user programs.

Menps is a pure library without code transformation and resides at the same level as SCASH in the software stack. Using code transformation for better transparency support is orthogonal to the main contribution of this dissertation.

ParADE [101, 96] is an OpenMP environment composed of both a source-to-source translator and a runtime system. The ParADE translator is based on Omni-compiler. The runtime system is a multi-threaded

DSM system based on a home-based lazy release consistency protocol. The authors are focused on their hybrid execution model which provides both message-passing primitives and shared-memory constructs.

FDSM [126] is a page-based DSM system designed for executing OpenMP programs. The authors did not implement a compiler for FDSM, but they assumed Omni-compiler to evaluate OpenMP programs on their DSM runtime system. FDSM can analyze the access pattern of a loop at its first iteration and calculate the *communication set* that indicates which variables are modified in a basic block. The communication set captures more fine-grained information than ordinary write notices because FDSM tracks and records each write instruction with its instruction emulation method. They showed that FDSM outperformed SCASH in the performance of NAS CG because the communication set eliminates the overhead of processing twins and diffs. Their acceleration method, which exploits the iterative characteristics of applications, is orthogonal to the proposals of Menps.

OMPi [154] is a source-to-source translator for C programs with OpenMP pragmas which produces the programs running on DSM systems. This system is characterized by the fact that it supports multiple threading systems and multiple DSM systems.

Kwon et al. [113] proposed a fully-automated compiler-runtime system to run OpenMP programs transparently on clusters. They provided the experimental results of their system close to the results with MPI. Compared to the DSM approaches, their method seems to be highly specific to array-based computational patterns which are typical in HPC applications. Because Menps is a page-based DSM library, array indexing looks like ordinary pointer dereferencing, which loses the application-level information about the data layouts. The additional information gathered at compile time may also be beneficial for DSM-based systems, but it does not solve the whole problem of coherent caches described in Section 3.4, which is inevitable in complex applications in general.

NanosDSM [53] is an "everything-shared" software DSM system that can execute OpenMP programs. To compile OpenMP programs for NanosDSM, the authors used a research compiler called Nanos Mercurium Compiler [23]. NanosDSM is different from other OpenMP systems on DSM because it can handle sharing call stacks as Menps can do (see also Section 3.6.3). The program code for NanosDSM is also shared among processes so that the initial node only needs to have the pages for the code. NanosDSM employs sequential consistency as its consistency model to encourage the cooperation between the DSM system and the OpenMP runtime system. As mentioned in Section 3.6.3, MEOMP is based on commodity OpenMP compilers while NanosDSM was implemented on the research compiler. When NanosDSM or other OpenMP systems on DSM were developed, GCC or other compilers did not support OpenMP.

6.1.5. DSM systems with operating systems support

ScaleMP [167] is an example of distributed shared memory systems in the industry. It is generally hard to discuss the difference between Menps and ScaleMP because the implementation detail of their systems is not published. ScaleMP seems to be a kernel extension and requires (at least) the installation of a kernel module, which is not practically applicable for the users of the current supercomputing systems who do not have the root privilege.

`libMPNode` [122] is an OpenMP runtime system designed for non-cache-coherent domains. Their runtime is based on a Popcorn Linux [123], which implements distributed shared memory at the kernel level. This kernel enables applications to migrate between heterogeneous-ISA processors.

6.1.6. Remote memory paging

The design of Menps focused on accelerating shared-memory applications by exploiting the CPU parallelism of multiple nodes. Another possible advantage of distributed shared memory systems is the increase in the overall available memory space. Most of the DSM systems including TreadMarks had memory copies in all of the nodes. Because Menps also does it for simplicity, it suffers from the capacity problem.

Table 6.2.: Comparison of existing PGAS systems.

Name	Global-view	Library-based	Base language
Co-array Fortran [144]	No	No	Fortran
UPC [66]	Yes	No	C
Global Arrays [137]	Yes	Yes	C
X10 [46]	Yes	No	Java
Chapel [43]	Yes	No	Original
OpenSHMEM [44]	No	Yes	C
XcalableMP [133]	No	No	Fortran or C
Grappa [135]	Yes	Yes	C++
UPC++ [58, 19]	Yes	Yes	C++

There have been a number of attempts to provide large memory space as software DSM systems. Cashmere-VLM (Very Large Memory) [64] is one of DSM systems which is focused on supporting large memory space in order to avoid swapping. JIAJIA [85] is a home-based page-based DSM system that implements remote memory paging. JIAJIA is one of the widely used DSM systems and employs Scope Consistency (Section 6.1.3) as its consistency model.

Infiniswap [76] is recently proposed as a tool to expand the application's memory space using the memory modules of other nodes in a cluster. Infiniswap is not a DSM system but a memory disaggregation system because a single application using Infiniswap only uses the CPU cores in a single node. With this limitation, Infiniswap can avoid many inherent issues around coherent caches that Menps tries to solve. Although Infiniswap requires the installation of a kernel module, it provides a truly transparent memory interface effectively utilizing the whole memory space of a cluster.

DLM [130] is also a remote paging system that divides the compute node from other memory servers. Their method is implemented as a user-level system and does not require kernel modification.

6.2. Partitioned Global Address Space (PGAS) systems

PGAS (Section 2.2.3) is a memory interface derived from DSM which provides global addressing while avoiding coherent caches. PGAS systems usually scale better than DSM systems because PGAS can be implemented as a thin system layer and application programmers of PGAS are aware of communication costs. Although PGAS is considered more productive than message-passing programming, PGAS is not as productive as "true" shared memory (= DSM) because it cannot automatically manage caches and require the applications to invoke explicit communication calls.

Table 6.2 shows the comparison of existing PGAS systems including both library-based and language-based systems. Early PGAS systems such as Unified Parallel C [66] and Global Arrays [137] appeared in the 2000s, and after that, the research on global address space systems started to be dominated by PGAS instead of DSM. These systems usually scale better than DSM systems because the application programs for PGAS are aware of communication overhead and optimized with some engineering efforts.

As argued in Section 1.1, even though there have been a number of PGAS systems over two decades, they do not have a standard interface that eases programmers to port their applications from one system to another. OpenSHMEM [44] is a recent effort to standardize the PGAS interface, but its interface is based on local-view PGAS, which is far from ordinary shared-memory programming.

Grappa [135] is recently published as a "DSM" system, but because it only supports a new programming model using global address space *without coherent caching*, it should be categorized as a PGAS system in common terminology. Some of the ideas in this dissertation can also be found in Grappa including the extensive use of user-level threading and SC-for-DRF consistency. They implemented a method to delegate computations to the data locations rather than moving the data as Menps does. Menps is focused on the data migration scheme, but it should be possible to migrate both tasks and data for generality.

Menps is different from these PGAS systems because it provides genuine coherent caching on top of distributed-memory systems.

GAM [40] is a recent example of a PGAS system *with coherent caching*. It means that its interface is based on the get/put functions like PGAS, but those function calls will utilize the local caches like DSM. The cache coherence protocol of GAM is based on directory-based coherence. GAM is focused on exploiting the RDMA performance as Menps, but not all of its communications are mapped to RDMA because it depends on the control channels using SEND operations (Section 2.3.1). Most of the evaluated benchmarks for GAM are variants of transaction processing (e.g. TPC-C [188]), which are simpler than the benchmarks evaluated in Menps. The authors of GAM's paper compares GAM with Argo [100]. Empirically, Argo sometimes performs better than GAM, but it happens mostly when the entire working set is cached in every node.

6.3. Hardware coherent caches

6.3.1. Self-invalidation techniques

As explained in Section 3.4.3, the standard method for cache invalidation is directory-based coherence, which is more scalable than snoopy coherence. There has been a large amount of research about snoopy or directory-based cache coherence protocols. For invalidation-based protocols, the cache invalidation problem is considered as *calculating the set of invalidated cache blocks* from the set of valid caches, which is shown in Section A.1.

To solve the problems of directory-based invalidations, there are many proposals to accomplish directory-less cache invalidations. *Self-invalidation* [157] is a concept in which readers spontaneously invalidate the caches rather than waiting for the writer-initiated invalidations. Many of those approaches assume multi-core coherence and small cache granularity and depend on page-level data classifications which require centralized data structures in reality. Instead, we have sought other approaches which could decentralize the coherence protocol.

DeNovo [51, 181, 180, 179] is a shared-memory architecture based on *disciplined parallelism*, which is a concept to limit "wild behaviors" of shared-memory systems and provide deterministic results. The concept of DeNovo derives from Deterministic Parallel Java (DPJ) [35], an experimental language extension to Java. Internally, the compiler for DeNovo inserts self-invalidation fence operations before synchronization points in the application code. This idea can be conceptually categorized as a compiler-based DSM system (Section 6.1.4) with additional hardware support. On the other hand, Menps is based on a purely library-based approach without any help from compilers and hardware, which enables rapid prototyping on existing real systems. The authors of DeNovo's papers are aware of the fact that their approaches including signature-based invalidation (see also Section 3.4.5) are close to the ideas which were originally invented in software DSM systems.

VIPS (or VIPS-M) [162] is a "directory-less" coherence protocol using self-invalidation with private-shared classification (P/S classification). In VIPS, every cache line is classified into either private or shared modes, which are managed by the operating systems at the page level. When cache lines belong to private pages, memory accesses to those lines can bypass the check for invalidation. When cache lines are shared, VIPS simply *self-invalidates* them without additional confirmation.

Esteve et al. [67] proposed a mechanism for TLB-based private-shared classification, which VIPS is also based on. They summarized that this classification strategy could be found in other self-invalidation techniques [161, 81, 104] which intended to accelerate coherence actions or reduce energy. In software DSM, however, using coarse-grained classification is less meaningful because the minimum cache granularity must be the page size on page-based DSM. Assuming the page-level classification from operating systems hides the hardware coherence problem into the software side, but this assumption is not applicable to fully software-based DSM systems. The page-level classification simply centralizes the coherence protocol to page tables which are managed by the software side (= operating systems).

Argo [100] (see also Section 3.2 or Section 6.1.2) is a software DSM system based on the self-invalidation technique named “P/S3 classification.” To implement P/S3 classification, Argo is still based on centralized directories, which seems contradictory to its directory-less concept. This is mainly because software page-based DSM systems cannot assume fallback paths such as operating systems. On the other hand, MENPS is designed as an entirely decentralized DSM system that does not depend on centralized directory information.

6.3.2. Reducing directory storage

The scalability problem of cache directories has been largely discussed for multi-core coherence. Many studies focused on reducing the directory storage because hardware directory structures increase the amount of circuit, which simply leads to huge energy consumption. However, on software DSM, it is less important to reduce directory storage because cache directories are normal memory regions that do not waste a significant amount of energy.

The simplest method to record a set of sharers is the full bit vector scheme, which associates a bit vector for each cache block. This scheme is not acceptable for hardware coherence with large memory because it uses the directory storage proportional to the size of the total memory and thus consumes a huge amount of energy. The other simple method is the limited pointer scheme [10], in which a directory records a limited number of processor IDs as integers. The limited pointer scheme can reduce the storage if the directories contain a small number of pointers. However, when the actual number of sharers exceeds the capacity of limited pointers, it needs to resort broadcasting as in snoopy coherence and degrades the performance.

Sparse directory [78] is a method to share a directory structure for multiple cache blocks to reduce the directory storage. In hardware coherent caches, because the cache size is much smaller than the main memory size, most of the cache blocks are not existent and do not require sharer tracking. When too many cache blocks conflicts in the same sparse directory, the previous cache block sharing the directory is invalidated (or evicted), which is called *directory-induced invalidation*. Sparse directory is often used as a standard method for hardware cache directories.

There is another basic directory-based method using a doubly-linked list proposed in the Scalable Coherent Interface (SCI) [79]. This method maintains links in private caches for each cache block as a distributed doubly-linked list. Both adding and removing a processor from the list can be done in constant time. Invalidation with this method needs to follow the distributed list and takes longer latency than centralized directories. The probable owner method (Section 3.4.1) has a similarity to this invalidation method in their data structures because both of them depend on distributed linked lists over private caches.

Duplicate-tag directory [26, 200] is a method to replicate the tag information on private caches into a data structure sorted by addresses. The duplicated tags can be searched in parallel because it is implemented in hardware. This method increases the energy proportional to the associativity.

SCD [163] is a proposal to organize the directory structures in a scalable manner. SCD dynamically switches several data structures for sharer tracking sets based on the number of sharers for each cache line. When a cache block has a small set of sharers, SCD uses the limited pointer scheme for the block’s directory. When it exceeds the capacity of the limited pointer, it switches to hierarchical bit vectors. In general, implementing complex data structures for directories is more suitable for software-based caching mechanisms, but it does not solve the fundamental problem of sharer tracking via cache directories.

Because SCD is proposed to scale the directory storage, it is supposed to slightly degrade the performance as a cost, but the evaluation of SCD reported that SCD “accelerated” one of the benchmark applications (canneal from PARSEC [30]). This is because directory-induced invalidations had the same effect of self-invalidation (Section 6.3.1).

6.3.3. Timestamp-based invalidations

It has been popular to utilize timestamps for cache coherence since the early DSM systems. For example, TreadMarks used vector timestamps to synchronize data structures updated concurrently. However, timestamp-based coherence (or lease-based coherence) is a different concept from them as it introduces a *lifetime* of cache blocks and manages the invalidations based on the predicted lifetime.

Most of timestamp-based coherence protocols [173, 176, 194] depend on *physical* timestamps, which are incremented at certain intervals. Because the synchronization of physical timestamps requires a global barrier, the scalability of those methods is limited. Another issue is that a writer needs to wait for the timestamp expiration in physical time, which leads to significant performance degradation in each write.

The cache invalidation method of Menps is largely affected by Tardis [198, 199], which invalidates cache blocks with logical timestamps. Compared with physical timestamps, logical timestamps do neither require periodic broadcasting nor a centralized coordinator, which enable decentralized coherence. Although Tardis is simulated as a multi-core hardware system, Menps is a software system evaluated on the real cluster system. The cache invalidation of Menps is different from Tardis because it effectively combines write-notice invalidation. The updated version [199] of Tardis implements a dynamic prediction method of the best lease value, which is orthogonal to and can be applied in Menps.

Yao et al. [195, 194] implemented a hybrid approach called TC-Release++ with both physical timestamps and bloom filters in order to shrink the write latency due to the matter of physical clocks. A bloom filter can work as an approximate set structure carrying invalidated cache IDs, while a write notice array can precisely invalidate caches and enables the system to transfer the data with a single RDMA READ operation. As explained in Section 3.4.5, Menps borrowed this idea to combine both timestamp-based coherence and precise write sets. However, the detailed protocol of Menps is different from TC-Release++ because Menps combined logical timestamp-based coherence and write notices. Employing physical timestamps requires a centralized coordinator for synchronizing clocks. Because bloom filters will lose the information of owners, it is hard to implement the "fast read" of Menps using bloom filters.

6.3.4. Hardware-based multiple-writer protocols

Recent work also presents multiple-writer protocols on hardware cache systems which share the same idea as the diff approach of software DSM systems. For example, Protozoa [202] is a hardware coherence protocol that can precisely track fine-grained writes sharing the same cache line. Protozoa is based on Amoeba-Cache [112], which handles different granularities for cache lines. The data structure used in Protozoa is similar to that of the diff approach as mentioned by the authors.

VIPS-M [162] (see also Section 6.3.1) is also classified as a multiple-writer protocol because it merges the writes from multiple cores using per-word dirty bits. The core idea of VIPS-M is the same as that of multiple-writer DSM systems.

6.4. Communication libraries for interconnects

Although InfiniBand [89] is widely adopted in the current supercomputers, there are other interconnection mechanisms such as uGNI [14], BlueGene/Q [49], Omni-Path [32], and Tofu [11]. Instead of directly using low-level APIs such as InfiniBand Verbs (IBV), low-level communication libraries are developed to achieve the portability of higher-level systems such as MPI and PGAS systems.

6.4.1. Low-level communication libraries

GASNet [38, 36] is widely known as a low-level communication library for PGAS and used in Unified Parallel C (UPC) [66] and other PGAS systems [203, 43, 144]. GASNet is internally divided into two parts: the core API based on Active Messages (Section 2.1.2) and the extended API for remote memory access and collective operations. This layering approach is adopted to achieve portability because RDMA or other networking facilities are not always available in various network architectures.

When GASNet was originally developed, RDMA was still an experimental feature of network hardware. Because RDMA has been commoditized over decades, the design of MECOM is focused on exploiting the performance of RDMA rather than supporting the hardware without RDMA.

GASNet-EX [37] is recently published as the next version of GASNet to redesign and extend the interface for modern interconnect hardware. One of the main changes is the enhanced RMA support to exploit the performance of RDMA. The interface of GASNet-EX supports multiple endpoints for various purposes.

ARMCI [138] is also a low-level communication library developed for a PGAS library Global Arrays [137]. The basic features of ARMCI are RMA and collective functions, which are similar to those of GASNet, but AM is not supported in ARMCI. Recently, the low-level communication layer for Global Arrays is replaced with ComEx [57], which is focused on interoperability with MPI.

Recently, two new libraries are emerging as low-level communication systems: Unified Communication X (UCX) [172] and libfabric [75]. UCX is a low-level communication library composed of the lower-level layer UCT and the higher-level layer UCP. MECOM 2 (Section 4.5) is implemented on top of UCT in order to accelerate RMA communications. libfabric is a sub-project of OpenFabrics [146], a project providing the InfiniBand drivers. The performance results in multi-threading executions are not examined in these new systems.

Papadopoulou et al. [149] analyzed the performance of UCX. They have first measured the latency and bandwidth of the original UCX runtime using microbenchmark programs. In their evaluation, UCT generally performs better than UCP because UCP incurs additional overhead to UCT due to function calls for supporting multiple UCT device types (similar to virtual function calls in C++). They further analyzed the instruction breakdown of these layers and presented several methods to optimize the implementation. Eliminating the specific parts of polling has reduced the latency and increased the bandwidth in the microbenchmarks.

6.4.2. Multi-threading performance of communication systems

As explained in Section 4.2.2, communication software offloading [190] has been implemented on top of MPI or other communication libraries to accelerate multi-threaded communications. MECOM (Chapter 4) is a communication library with software offloading using dedicated threads.

Because most of MPI runtime systems have coarse-grained locks to guard the communication resources, it is hard to parallelize the communication processing of MPI implementations. Amer et al. [16] analyzed the issues of lock arbitration (or fairness, described in Section 2.5) of MPICH, one of the famous MPI implementation. They used ticket locking for improving fairness. Dang et al. [59] improved the work of Amer et al. using queue-based locking. Recently, Amer et al. [15] focused on software combining to accelerate MPI communications in multi-threading.

Fukuoka et al. [73] proposed a system named MPI+myth, which implements a software offloading layer for MPI using a user-level threading library MassiveThreads [134]. They used a feature named *uncondition variables (uncond)* which efficiently suspends and resumes user-level threads to suspend application tasks. Suspended threads (Section 5.2.3) of ComposableThreads is derived from this uncond feature.

6.4.3. Performance characteristics of RDMA

In database research, there has been an active debate on whether RDMA is useful for building transactional processing. There are examples which used RDMA for transactional processing: FaRM [62], DrTM [192], DrTM+R [50]. However, other proposals rely on two-sided primitives in InfiniBand (SEND operations) such as HERD [98] and FaSST [99] criticizing the scalability of RDMA. Most of these systems are optimized for executing transactions in remote nodes, which is a different workload from what DSM systems assume. Mixing different communication models may provide better performance but complicates the system design.

6.5. User-level threading and task scheduling

6.5.1. User-level threading on shared-memory architecture

There are many libraries which implement user-level threading in C or C++ languages: MassiveThreads [134], Qthreads [193], Argobots [171], TBB [2], Nanos++ [4], and Boost.Fiber [107]. Most of these libraries are designed for efficiently scheduling HPC applications. ComposableThreads (Chapter 5) is also categorized as a variant of these libraries, but it is focused on decomposing the library itself mainly for system programming research rather than tuning the performance of typical HPC applications.

There are also language extensions for task parallelism such as Cilk [72], Cilk Plus [1], and OpenMP 3.0 (or higher) [5]. Some other programming languages [184, 156] natively support user-level threading at the language level.

6.5.2. Stackful or stackless coroutines

User-level threading is an extension of *stackful coroutines* into multi-threaded parallel execution. Coroutines are functions which can be suspended and resumed during their execution. Coroutines can be considered as a generalization of subroutines, that are functions without suspension. *Stackful* means that every coroutine has its own call stack to save the continuation for suspension. Boost.Context [106] is a stackful coroutine library for context switching for C++, which was developed for a user-level threading library Boost.Fiber [107].

Stackless coroutines [140, 139] are coroutines which do not save the call stacks. This feature will be included in C++20, the next standard version of C++. Stackless coroutines is a promising core language feature for implementing tasking appearing in this dissertation, but until now, stable implementations have not been available.

Split stacks [183] is a compiler feature that allows discontiguous call stacks. It can be used as a mitigation method to prevent allocating the whole call stack for each coroutine. Split stacks is enabled with a special compiler option which changes the code generation to allocate the memory for stack frames on demand. This feature adds the overhead on each function invocation because it checks whether the current stack chunk size is overflowing or not.

6.5.3. Work-stealing schedulers

The implementation of Cilk [72] used an implementation method called THE protocol for implementing work-stealing deques which hold runnable threads in work-stealing schedulers. ABP work-stealing [18] implements worker deques in a non-blocking manner, which means that it eliminated the use of spinlocks associated with the deques. Chase et al. [48] improved this deque structure by supporting a dynamic extension of deque lengths, which has been called Chase-Lev deque. Chase-Lev deque is the current state-of-the-art implementation of work-stealing deques. Lê et al. [117] proposed an implementation method of Chase-Lev deque on weak memory models.

It is also possible to implement work-stealing schedulers on distributed-memory architectures. Iso-address [17] is a method to migrate threads (or stackful coroutines) between processes. In C or C++, because call stacks may include pointers to stack frames, it is impossible to migrate coroutines by simply copying a call stack from one process to another. In iso-address, because every coroutine is guaranteed to be allocated in different addresses, it can be migrated to another process by copying to the same address.

Uni-address [12] is a similar method to iso-address, but it allocates “all” of call stacks to the same address region. Assuming that each process executes a task as a single thread, the number of running tasks is at most one, enables uni-address to reduce the amount of virtual address space and reduce the registered regions for RDMA. They have shown that their threading system uni-address threads runs on distributed memory at the speed comparable to MassiveThreads.

6.5.4. Implementations of mutual exclusion

There are many variations to implement mutual exclusion on shared-memory systems. For example, Guerraoui et al. [77] gave a detailed analysis on many locking schemes including TTAS locks, ticket locks, queue locks, and so on. They revealed many observations; for example, the storage cost of locks is often ignored by developers, but it also affects the performance in some scenarios with many lock variables. They concluded that no single lock is systematically the best and programmers need to choose the locks based on their purposes.

As mentioned in Section 5.2.5, communication offloading is an instance of the lock delegation problem. Apart from communication or I/O processing, lock delegation has been largely discussed as a variant of mutual exclusion problems. For example, Klaftenegger et al. [105] proposed queue delegation locking to implement lock delegation based on MCS locks and circular buffers. MECOM 1 (Chapter 4) also used circular buffers to delegate communication requests.

ffwd [160] is a lock delegation method that uses dedicated threads for executing critical sections. ffwd is focused on increasing the throughput of critical sections. The performance results of ffwd were, in general, much faster than conventional methods using spinlocks, queue-based locks, software transactional memory, and so on. This is because ffwd does *not* depend on any atomic operations for mutual exclusion but on the round-robin monitoring on dedicated spinning threads. Their implementation method may be translated into software communication offloading. MECOM is designed to avoid spinning of dedicated threads because our work is focused on a more complicated workload which is observed in MENPS.

7. Conclusions

In this dissertation, we have presented the methods to implement a decentralized software distributed shared memory (DSM) for modern cluster computing systems. There have been an enormously large number of trials around distributed shared memory, but this dissertation has several distinctions from the existing work. First, we have tried to develop portable methods which can get the benefit of page-based DSM systems. Second, this dissertation explored the ways to exploit the low-level features of the existing system components, especially the interconnect hardware and multi-core architecture. Both of them have enabled this work to enlighten the study of DSM again from a different perspective.

To demonstrate our ideas about DSM, we have implemented the prototype DSM system MENPS from scratch (Chapter 3). MENPS has many differences from the existing systems, but the most important point is the design policy to adapt its coherence actions to the restrictions of Remote Direct Memory Access (RDMA) (Section 2.3.1). Although it has been already known that it is possible to build a DSM system purely on RDMA (e.g. [100]), the existing work simply translated the standard methods for implementing shared-memory systems, e.g. directory-based coherence (Section 3.4.3), into RDMA-based communications. We noticed that implementing RDMA-based DSM systems requires completely different methods to synchronize and transfer the coherent caches between nodes (Section 3.2). Because the RDMA operations are usually point-to-point communications, it is important to design decentralized protocols without broadcasting or multicasting, but the directory-based coherence is not appropriate for RDMA-based DSM systems because it requires multicasting for cache invalidation by nature. We have shown that the decentralized protocol of MENPS can be implemented only with point-to-point communications. While it is decentralized by nature in the protocol level, it provides comparable or better in the absolute performance because it does not depend on extreme unrealistic solutions such as self-invalidating all of the caches to achieve decentralization.

As a method for write management, a new scheme called floating home-based method (Section 3.4.2) is presented to exploit the performance of RDMA. As shown in Figure 3.1, it is significantly important to place the transferred data contiguously as an RDMA-friendly protocol. Typical home-based approaches in other DSM systems need to send multiple fine-grained RDMA WRITE messages to merge the writes, but in the floating home-based approach, the communications are always coarse-grained. The floating home-based method depends on a trick that uses “RDMA READ operations for processing memory writes” so that it can proceed without the remote CPUs’ involvement which cannot be supported by RDMA normally.

To implement cache invalidation for coherence, MENPS introduced a hybrid approach of logical lease-based coherence and write notices (Section 3.4.5). The combination of those two methods mitigates their problems, over self-invalidation in lease-based coherence and infinite storage consumption in write notices, with each other. As far as we know, this is the first attempt to employ logical lease-based coherence into software distributed shared memory. These two methods are suitable for RDMA because the best case of memory reads only issues RDMA READs exactly once because of the fast read method, which matches the zero-copying concept of RDMA.

Although it is possible to design an RDMA-based DSM system based on these two methods (the floating home-based method and the hybrid cache invalidation method) for processing memory writes and reads, we have found the performance bottleneck in DSM release fences which appears in each OpenMP barrier to shrink the latency of the release fences. we have introduced a new feature called fast release (Section 3.4.6) so that most of the heavy communications and computations can be delayed and aggregated until memory blocks need to be migrated.

In the evaluation of MENPS, we have shown the advantages of the proposed methods for building an RDMA-based DSM system. The evaluation of the proposed methods of MENPS (Section 3.8.1) showed that they improved the performance results against the traditional methods including the fixed-home method or directory-based coherence.

Although the design of Menps is decentralized by nature, the evaluation results of Menps show that only simple applications can scale well with hundreds of threads. The results of Menps indicate the inherent problems of page-based DSM systems, which suffer from the paging overheads both by the operating system and the architecture.

To implement Menps, we have also implemented a communication library MECOM (Chapter 4) which wraps the low-level features of interconnects. Because Menps depends on the communication performance of RDMA on multi-threading systems, it was critical for MECOM to accelerate multi-threaded communications. MECOM employs software communication offloading based on user-level threading to process communication requests efficiently on multi-threading environments.

The evaluation of MECOM shows that the aggregated message rates can be improved using multiple communication resources without the explicit handling of communication progress. It is also shown that the current method for software offloading has the problem of increased latency on NUMA environments. This is because normal user-level threading schedulers frequently migrate the communication threads per NUMA domains.

A user-level threading library ComposableThreads is also proposed in this dissertation (Chapter 5). We also noticed that the method of software offloading implemented in MECOM could be generalized to different purposes and provided a new interface for general lock delegations in ComposableThreads. In a few microbenchmark results, we have shown that the performance of ComposableThreads was comparable to that of the existing user-level thread library MassiveThreads.

The concluding remark of this dissertation is that building a software shared-memory system like Menps from scratch is meaningful because it gives us many important observations about how we can understand and should design parallel computer architecture. Looking at the existing computing environments from a different perspective, we can testify the common knowledge of parallel computing using real applications or hardware. The study of Menps explored a variety of research topics such as cache coherence, memory consistency, parallel programming languages, memory management, hardware communication acceleration, thread scheduling, mutual exclusion, and so on. We believe that any of the ideas or observations of this dissertation which span those research topics will encourage the readers to flourish their new ideas for more productive parallel computing in the future.

7.1. Future work

There are many possible research directions to extend the work of Menps:

Performance comparison with other programming models

This dissertation explained the strengths and weaknesses of different programming interfaces including DSM, PGAS, and message passing. Although Section 3.8 compared the performance of Menps with MPI, for example, the detailed analysis of those programming interfaces should be made with the real performance results.

Combining compiler-based approaches into Menps

Menps is a purely library-based DSM system which is free from compiler-based techniques, but compiler-based approaches (Section 6.1.4) are useful to exploit the application-level information for accelerating the performance. For example, avoiding the false sharing problem is easier than page-based DSM. Compared to when DSM had been actively studied in the 1990s, we can nowadays use LLVM [6] to implement code transformations relatively easily. Page-based DSM including Menps still has advantages for working as the base layer of higher-level components.

Implementing Menps' coherence as a kernel module

Because Menps is entirely implemented as a user-level runtime system, it has limitations from the implementation of operating systems. If it is allowed to modify the kernel of the clusters, it becomes easier to implement software DSM without modifying the binary executable files because the whole runtime system can be trapped at the kernel level. This approach may be perhaps attempted by other systems including ScaleMP [167] (Section 6.1.4).

Translating MENPS' coherence to many-core architecture

The coherence protocol of MENPS is originally developed to accelerate software DSM systems. However, some features of its coherence protocol can be applied to hardware-based coherence. For instance, logical lease-based coherence [198, 199, 197], which is combined with MENPS's invalidation method (Section 3.4.4), was first proposed for hardware coherence. The hybrid approach of MENPS with write-notice invalidation can be also implemented on hardware, which is useful to mitigate the drawback of lease-based coherence.

Debugging concurrency bugs of shared-memory programs

As explained in Section 3.5.4, MENPS is capable of detecting write-write races because it merges writes in its multiple-writer coherence protocol and this merge process can include race detection. This method can be further extended to effectively debug shared-memory programs. Because MENPS is a software shared-memory system that can change home selection or cache invalidation, it may be beneficial to detect such a bug in different configurations provided by MENPS.

A. Appendix

A.1. Cache invalidation problem

Section 3.4.5 described the cache invalidation mechanism of MENPS. More generally, the cache invalidation problem can be formalized as a set calculation between processes. This appendix section presents a formal description of this cache invalidation problem.

In normal caching systems, the entire memory is divided into cache blocks. Assume that there are four kinds of memory operations as in release consistency: read, write, acquire, and release. Additionally, the system may evict some blocks independently from the operational semantics. With these settings, define the variables as follows:

- p : a process with a distinct address space
- i : an interval on a certain process
- $\mathbf{V}(p,i) = (V(p,i,0),\ldots,V(p,i,P-1))$: a vector timestamp of a process p at an interval i
  - $\forall p, \forall i, V(p,i,p) = i$
- $R(p,i), W(p,i)$: sets of blocks which cause read/write misses on a process p at an interval i
- $E(p,i)$: a set of evicted blocks on a process p at an interval i
- $C(p,i)$: a set of cached blocks on a process p at an interval i

On a release on a process p_r at an interval i_r, the vector timestamp is updated as:

$$V(p_r,i_r,p) = \begin{cases} V(p_r,i_r-1,p_r)+1 & \text{if } p = p_r \\ V(p_r,i_r-1,p) & \text{otherwise} \end{cases} \tag{A.1}$$

On an acquire on a process p_a at an interval i_a which is paired with a process p_r at an interval i_r, the vector timestamp is updated as:

$$V(p_a,i_a,p) = \max(V(p_a,i_a-1,p_a)+1, V(p_r,i_r,p)) \tag{A.2}$$

The acquirer p_a needs to invalidate old blocks on that process. The set of blocks newly written by a process p is defined as:

$$W_a(p_a,i_a,p) = \bigcup_{\substack{i \\ V(p_a,i_a-1,p) \,\le\, i \,<\, V(p_r,i_r,p)}} W(p,i) \tag{A.3}$$

The acquirer needs to invalidate blocks written by all the other processes. The set of blocks newly written is defined as:

$$W_{a\cup}(p_a,i_a) = \bigcup_{\substack{p \\ p \neq p_a}} W_a(p_a,i_a,p) \tag{A.4}$$

The blocks invalidated on the acquire is:

$$I_a(p_a,i_a) = W_{a\cup}(p_a,i_a) \cap C(p_a,i_a) \tag{A.5}$$

The set of cached blocks is updated as:

$$
\begin{aligned}
C(p_a, i_a + 1) &= ((C(p_a, i_a) \setminus I_a(p_a, i_a)) \cup R(p_a, i_a)) \setminus E(p_a, i_a) && \text{(A.6)} \\
&= ((C(p_a, i_a) \setminus W_{a\cup}(p_a, i_a)) \cup R(p_a, i_a)) \setminus E(p_a, i_a) && \text{(A.7)}
\end{aligned}
$$

$I_a(p_a, i_a)$ is an intersection of the sets of written and cached blocks. As discussed in implementing `JOIN` operations in RDBMS, there are two ways to calculate a set intersection $A \cap B$:

1. Loop A and check the existence in B. The time complexity is $O(|A|s(B))$ where $s(X)$ is the time spent to search an element in X.
2. Loop B and check the existence in A. The time complexity is $O(|B|s(A))$.

If these two ways are adaptively selected, then the complexity is $O(\max(|A|s(B), |B|s(A)))$.

The complexity of calculating I_a is $O(\max(|I_a|s(C), |C|s(I_a)))$. If the search time is a constant ($s(X) = c$. A bitset, a hash table (on average)), it is $O(\max(|I_a|, |C|))$.

List of Publications

International conferences:

- Wataru Endo, Kenjiro Taura: Parallelized Software Offloading of Low-Level Communication with User-Level Threads. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPC Asia 2018). Tokyo, Japan. pp. 289–298.
- (unrefereed presentation) Wataru Endo, Kenjiro Taura: A Distributed Shared Memory Library with Global-View Tasks on High-Performance Interconnects. SIAM-PP 2018 Contributed Presentation. Tokyo, Japan. March 2018.

Domestic conferences / workshops in Japan:

- Wataru Endo, Kenjiro Taura: Parallelized Software Offloading in Low-Level Communication Layer. xSIG 2017. April, 2017. Best M2 Student Award.
- (poster) Wataru Endo, Kenjiro Taura: Parallelized Software Offloading in a Low-Level Communication Layer. xSIG 2017. April, 2017.
- (unrefereed) Wataru Endo, Kenjiro Taura: 論理タイムスタンプに基づく分散共有メモリライブラリの実装 (English translation: Implementation of a Distributed Shared Memory Library Based on Logical Timestamps). SWoPP2018. July, 2018.
- (unrefereed) Wataru Endo, Kenjiro Taura: 分散スレッドスケジューラと協調する分散共有メモリ処理系の初期評価 (English translation: Initial Evaluation of a Distributed Shared Memory System Cooperating with a Distributed Thread Scheduler). SWoPP2017. July, 2017.

Bibliography

[1] Intel Cilk Plus. https://www.cilkplus.org/.

[2] Intel Threading Building Blocks. https://www.threadingbuildingblocks.org/.

[3] MPICH. http://www.mpich.org/.

[4] Nanos++. https://pm.bsc.es/nanox.

[5] OpenMP. https://www.openmp.org/.

[6] The LLVM Compiler Infrastructure. http://llvm.org/.

[7] PVM: Parallel virtual machine: A users' guide and tutorial for networked parallel computing. *Computers & Mathematics with Applications*, 30(9):122, nov 1995.

[8] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting data races on weak memory systems. *[1991] Proceedings. The 18th Annual International Symposium on Computer Architecture*, pages 1–10, 1991.

[9] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, volume 18, pages 2–14, may 1990.

[10] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*, pages 280–289, 1988.

[11] Y. Ajima, Y. Takagi, T. Inoue, S. Hiramoto, and T. Shimizu. The Tofu Interconnect. In *HOTI '11: Proceedings of IEEE 19th Annual Symposium on High Performance Interconnects*, pages 87–94, aug 2011.

[12] S. Akiyama and K. Taura. Uni-Address Threads: Scalable Thread Management for RDMA-Based Work Stealing. In *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 15–26. ACM Press, 2015.

[13] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long messages into the LogP Model. *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, 1995.

[14] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. In *HOTI '10: Proceedings of the 18th IEEE Symposium on High Performance Interconnects*, pages 83–87. IEEE, aug 2010.

[15] A. Amer, C. Archer, M. Blocksome, C. Cao, M. Chuvelev, H. Fujita, M. Garzaran, Y. Guo, J. R. Hammond, S. Iwasaki, K. J. Raffenetti, M. Shiryaev, M. Si, K. Taura, S. Thapaliya, and P. Balaji. Software combining to mitigate multithreaded MPI contention. *Proceedings of the International Conference on Supercomputing*, pages 367–379, 2019.

[16] A. Amer, H. Lu, Y. Wei, P. Balaji, and S. Matsuoka. MPI+Threads: Runtime Contention and Remedies. In *PPoPP '15: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 239–248, 2015.

[17] G. Antoniu, L. Bougé, and R. Namyst. An efficient and transparent thread migration scheme in the PM2 runtime system. In *RTSPP '99: Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming*, volume 1586, pages 497–510, 1999.

[18] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.

[19] J. Bachan, S. B. Baden, S. Hofmeyr, M. Jacquelin, A. Kamil, D. Bonachea, P. H. Hargrove, and H. Ahmed. UPC++: A High-Performance Communication Framework for Asynchronous Computation. *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 963–973, 2019.

[20] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS Parallel Benchmarks. *The International Journal of High Performance Computing Applications (IJHPCA)*, 5(3):63–73, sep 1991.

[21] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Toward Efficient Support for Multithreaded MPI Communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 120–129. Springer Berlin Heidelberg, 2008.

[22] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *International Journal of High Performance Computing Applications*, 24(1):49–57, 2010.

[23] J. Balart and A. Duran. Nanos mercurium: a research compiler for openmp. *European Workshop on OpenMP (EWOMP'04)*, pages 103–109, 2004.

[24] R. Baldoni, M. Raynal, and M. Klusch. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2):—-, 2002.

[25] S. Barghi. *Improving the Performance of User-level Runtime Systems for Concurrent Applications*. PhD thesis, University of Waterloo, 2018.

[26] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, (June):282–293, 2000.

[27] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ Concurrency. In *POPL '11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 55, 2011.

[28] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. Technical report, 1993.

[29] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. De Maria, M. Abud, and C. L. Amorim. Hiding communication latency and coherence overhead in software DSMs. *Operating Systems Review (ACM)*, 30(5):198–209, 1996.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, (January):72–81, 2008.

[31] A. Bilas, L. Iftode, R. Samanta, and J. P. Singh. Supporting a Coherent Shared Address Space Across SMP Nodes: An Application-Driven Investigation. (December):19–59, 1999.

[32] M. S. Birrittella, M. Debbage, R. Huggahalli, J. Kunz, T. Lovett, T. Rimmer, K. D. Underwood, and R. C. Zak. Intel® Omni-Path Architecture: Enabling Scalable, High Performance Fabrics. In *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 1–9, aug 2015.

[33] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. Dag-Consistent Distributed Shared Memory. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 132–141, 1996.

[34] R. L. Bocchino, V. S. Adve, S. V. Adve, M. Snir, and R. L. B. Jr. Parallel Programming Must Be Deterministic by Default. *Proceedings of the First USENIX conference on Hot topics in parallelism*, 22(1):4, 2009.

[35] R. L. Bocchino, M. Vakilian, V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, and H. Sung. A type and effect system for deterministic parallel Java. *ACM SIGPLAN Notices*, 44(10):97, oct 2009.

[36] D. Bonachea. GASNet Specification Version 1.8. Technical report, EECS Department, University of California, Berkeley, 2006.

[37] D. Bonachea and P. H. Hargrove. GASNet-EX: A high-performance, portable communication library for exascale. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11882 LNCS:138–158, 2019.

[38] D. Bonachea and J. Jeong. GASNet: A Portable High-Performance Communication Layer for Global Address-Space Languages. *CS258 Parallel Computer Architecture Project*, pages 1–27, 2002.

[39] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. *Ottawa Linux Symposium (OLS)*, pages 1–12, 2012.

[40] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang. Efficient distributed memory management with RDMA and caching. *Proceedings of the VLDB Endowment*, 11(11):1604–1617, 2018.

[41] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *SOSP '91: Proceedings of the 13th ACM Symposium on Operating Systems Principles*, volume 25, pages 152–164, 1991.

[42] A. Castello, A. J. Pena, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Orti. A Review of Lightweight Thread Approaches for High Performance Computing. In *CLUSTER 2016: Proceedings of 2016 IEEE International Conference on Cluster Computing*, number 1, pages 471–480, sep 2016.

[43] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21:291–312, 2007.

[44] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM. In *PGAS '10: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, number c, pages 1–3, New York, New York, USA, 2010. ACM Press.

[45] B. Chapman, G. Jost, and R. V. D. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press, 2007.

[46] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 40, pages 519–538, 2005.

[47] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991.

[48] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the 17th annual ACM symposium on Parallelism in algorithms and architectures*, number c, page 21. ACM Press, 2005.

[49] D. Chen, J. J. Parker, N. a. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, and B. Steinmacher-Burow. The IBM Blue Gene/Q Interconnection Network and Message Unit. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 1, 2011.

[50] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and General Distributed Transactions Using RDMA and HTM. *EuroSys*, pages 1–17, 2016.

[51] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-t. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT '11: Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 155–166, oct 2011.

[52] J. W. Chung, B. H. Seong, K. H. Park, and D. Park. Moving Home-Based Lazy Release Consistency for Shared Virtual Memory Systems. *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 282, 1999.

[53] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running OpenMP applications efficiently on an everything-shared SDSM. *Journal of Parallel and Distributed Computing*, 66(5):647–658, 2006.

[54] T. S. Craig. Building FIFO and priorityqueuing spin locks from atomic swap. Technical report, 1993.

[55] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. *PPoPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 28(7):1–12, 1993.

[56] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. apr 1999.

[57] J. Daily, A. Vishnu, B. Palmer, H. van Dam, and D. Kerbyson. On the Suitability of MPI as a PGAS Runtime. In *HiPC '14: The 21st annual IEEE International Conference on High Performance Computing*, pages 1–10, dec 2014.

[58] B. Dalton, G. Tanase, M. Alvanos, G. Almási, and E. Tiotto. Memory Management Techniques for Exploiting RDMA in PGAS Languages. In *LCPC '14: Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing*, pages 193–207, 2015.

[59] H. V. Dang, S. Seo, A. Amer, and P. Balaji. Advanced Thread Synchronization for Multithreaded MPI Implementations. *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, pages 314–324, 2017.

[60] S. Di Girolamo, T. Hoefler, K. Taranov, A. Kurth, M. Schaffner, T. Schneider, J. Beránek, M. Besta, L. Benini, and D. Roweth. Network-accelerated non-contiguous memory transfers. In *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, New York, New York, USA, 2019. ACM Press.

[61] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. *EuroMPI '13: Proceedings of the 20th European MPI Users' Group Meeting*, page 13, 2013.

[62] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast Remote Memory. *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[63] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–76, 1998.

[64] S. Dwarkadas, N. Hardavellas, L. Kontothanassis, R. Nikhil, and R. Stets. Cashmere-VLM: Remote memory paging for software distributed shared memory. In *Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999*, pages 153–159. IEEE Comput. Soc, 1999.

[65] H. Eichner, C. Trinitis, and T. Klug. Implementation of a DSM-system on top of InfiniBand. *Proceedings - 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2006*, 2006:178–183, 2006.

[66] T. El-Ghazawi and F. Cantonnet. UPC Performance and Potential: A NPB Experimental Study. In *SC '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, page 26, 2002.

[67] A. Esteve, A. Ros, M. E. Gomez, A. Robles, and J. Duato. Efficient TLB-Based Detection of Private Pages in Chip Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):748–761, mar 2016.

[68] M. Farreras, G. Almási, C. Caşcaval, and T. Cortes. Scalable RDMA performance in PGAS languages. In *IPDPS '09: Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium*, 2009.

[69] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *HPCA '11: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 169–180, 2011.

[70] S. Fortune and J. Wyllie. Parallelism in random access machines. *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[71] P. W. Frey and G. Alonso. Minimizing the Hidden Cost of RDMA. In *ICDCS '09: Proceedings of 29th IEEE International Conference on Distributed Computing Systems*, pages 553–560, 2009.

[72] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. *ACM SIGPLAN Notices*, 33(5):212–223, may 1998.

[73] T. Fukuoka, W. Endo, and K. Taura. An Efficient Inter-Node Communication System with Lightweight-Thread Scheduling. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 687–696. IEEE, aug 2019.

[74] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[75] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres. A Brief Introduction to the OpenFabrics Interfaces - A New Network API for Maximizing High Performance Application Efficiency. In *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 34–39, aug 2015.

[76] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin. Efficient Memory Disaggregation with Infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, 2017.

[77] R. Guerraoui, H. Guiroux, R. Lachaize, V. Quéma, and V. Trigonakis. Lock – Unlock: Is That All? A Pragmatic Analysis of Locking in Software Systems. *ACM Transactions on Computer Systems*, 36(1):1–149, 2019.

[78] A. Gupta, W. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. *ICPP '90: Proceedings of the International Conference on Parallel Processing*, 1(August):1–10, 1990.

[79] D. B. Gustavson. The Scalable Coherent Interface and related standards projects, 1992.

[80] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic home node reallocation on software distributed shared memory. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, pages 158–163 vol.1. IEEE, 2000.

[81] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, (June):184–195, 2009.

[82] M. Herlihy. The art of multiprocessor programming. 2006.

[83] T. Hoefler, S. Di Girolamo, K. Taranov, R. E. Grant, and R. Brightwell. sPIN: High-performance streaming Processing in the Network. 2017.

[84] Y. Hong, Y. Zheng, F. Yang, B. Y. Zang, H. B. Guan, and H. B. Chen. Scaling out NUMA-Aware Applications with RDMA-Based Distributed Shared Memory. *Journal of Computer Science and Technology*, 34(1):94–112, 2019.

[85] W. Hu, W. Shi, Z. Tang, Z. Zhou, and M. R. Eskicioglu. JIAJIA: An SVM System Based on a New Cache Coherence Protocol. (980001), 1998.

[86] Y. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, dec 2000.

[87] L. Iftode. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, volume 31, pages 451–473, 1998.

[88] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving release-consistent shared virtual memory using automatic update. *IEEE High-Performance Computer Architecture Symposium Proceedings*, pages 14–25, 1996.

[89] InfiniBand Trade Association. InfiniBand Architecture Specification, 2007.

[90] V. Iosevich and A. Schuster. A comparison of sequential consistency with home-based lazy release consistency for software Distributed Shared Memory. *Proceedings of the International Conference on Supercomputing*, (Mv):306–315, 2004.

[91] V. Iosevich and A. Schuster. Multithreaded home-based lazy release consistency over VIA. *Proceedings - International Parallel and Distributed Processing Symposium, IPDPS 2004 (Abstracts and CD-ROM)*, 18:831–840, 2004.

[92] V. Iosevich and A. Schuster. Software Distributed Shared Memory: A VIA-based implementation and comparison of sequential consistency with home-based lazy release consistency. *Software - Practice and Experience*, 35(8):755–786, 2005.

[93] A. Itzkovitz and A. Schuster. Multiview and millipage-fine-grain sharing in page-based DSMs. *Operating systems review*, 1998.

[94] S. Iwasaki, A. Amer, K. Taura, and P. Balaji. Lessons Learned from Analyzing Dynamic Promotion for User-Level Threading. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 293–304. IEEE, nov 2018.

[95] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji. BOLT: Optimizing OpenMP Parallel Regions with User-Level Threads. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, volume 2019-Septe, pages 29–42. IEEE, sep 2019.

[96] W. C. Jeun, Y. S. Kee, S. Ha, and C. Kee. Overcoming performance bottlenecks in using OpenMP on SMP clusters. *Parallel Computing*, 34(10):570–592, 2008.

[97] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-performance all-software distributed shared memory. *Proceedings of the 15th ACM Symposium on Operating Systems Principles, SOSP 1995*, pages 213–228, 1995.

[98] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proceedings of the 2014 ACM conference on SIGCOMM - SIGCOMM '14*, pages 295–306, New York, New York, USA, 2014. ACM Press.

[99] A. Kalia, M. Kaminsky, and D. G. Andersen. Design Guidelines for High Performance RDMA Systems. *USENIX ATC '16: Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, pages 437–450, 2016.

[100] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas. Turning Centralized Coherence and Distributed Critical-Section Execution on their Head. In *HPDC '15: Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM Press, 2015.

[101] Y. S. Kee, J. S. Kim, and S. Ha. ParADE: An OpenMP programming environment for SMP cluster systems. *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC 2003*, 2003.

[102] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA '92: Proceedings the 19th Annual International Symposium on Computer Architecture*, pages 13–21. IEEE, 1992.

[103] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks : Distributed Shared Memory on Standard Workstations and Operating Systems. In *WTEC '94: Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–132, 1994.

[104] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace snooping. In *PACT '10: Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, page 111, New York, New York, USA, 2010. ACM Press.

[105] D. Klaftenegger, K. Sagonas, and K. Winblad. Queue Delegation Locking. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):687–704, mar 2018.

[106] O. Kowalke. Boost.Context. http://www.boost.org/doc/libs/1_63_0/libs/context/doc/html/index.html.

[107] O. Kowalke. Boost.Fiber. http://www.boost.org/doc/libs/release/libs/fiber.

[108] M. K. Kumar, S. Maass, S. Kashyap, J. Veselý, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna. LATR: Lazy Translation Coherence. *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '18*, pages 651–664, 2018.

[109] S. Kumar, R. Blackmore, S. Sharkawi, N. J. K. A., A. Mamidala, and T. J. C. Ward. Optimization of Message Passing Services on POWER8 InfiniBand Clusters. In *EuroMPI 2016: Proceedings of the 23rd European MPI Users' Group Meeting*, pages 158–166, 2016.

[110] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. *IPDPS '12: Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 763–773, 2012.

[111] S. Kumar, Y. Sun, and L. V. Kale. Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q. In *IPDPS '13: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 689–699. IEEE, may 2013.

[112] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-Cache: Adaptive blocks for eliminating waste in the memory hierarchy. *MICRO-45: Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 376–388, 2012.

[113] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff. A hybrid approach of OpenMP for clusters. In *PPoPP '12: Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, volume 47, page 75, sep 2012.

[114] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, 1974.

[115] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[116] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, 1979.

[117] N. M. Lê, A. Pop, A. Cohen, and F. Zappa Nardelli. Correct and Efficient Work-Stealing for Weak Memory Models. In *PPoPP '13: Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, number 1, page 69, New York, New York, USA, 2013. ACM Press.

[118] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *ISCA '90: Proceedings of the 17th annual international symposium on Computer Architecture*, pages 148–159. IEEE Comput. Soc. Press, 1990.

[119] K. Li. IVY: a shared virtual memory system for parallel computing. In *ICPP '88: Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, 1988.

[120] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, nov 1989.

[121] H. Lu, S. Seo, and P. Balaji. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In *HPCC '15: IEEE 17th International Conference on High Performance Computing and Communications*, pages 444–454, aug 2015.

[122] R. Lyerly, S. H. Kim, and B. Ravindran. LibMPNode: An OpenMP runtime for parallel processing across incoherent domains. *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM 2019*, pages 81–90, 2019.

[123] R. F. Lyerly. *Popcorn Linux: A Compiler and Runtime for State Transformation Between Heterogeneous-ISA Architectures*. 2016.

[124] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7):78, jul 2012.

[125] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*, 25(2):85–97, may 1997.

[126] H. Matsuba and I. Yutaka. OpenMP on the FDSM software distributed shared memory. *the Fifth European Workshop on OpenMP*, 2003.

[127] Mellanox Technologies. RDMA Aware Networks Programming User Manual rev 1.7, 2015.

[128] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, 1991.

[129] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *ASPLOS IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, volume 19, pages 269–278, New York, New York, USA, 1991. ACM Press.

[130] H. Midorikawa, K. Saito, M. Sato, and T. Boku. Using a cluster as a memory resource: A fast and large virtual memory on MPI. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, pages 1–10, 2009.

[131] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox InfiniBand Software Stack. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4128 LNCS, pages 124–133. 2006.

[132] MPI Forum. MPI: A Message-Passing Interface Standard Version 3.1. 2015.

[133] M. Nakao, J. Lee, T. Boku, and M. Sato. Productivity and performance of global-view programming with XcalableMP PGAS language. In *Proceedings - 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012*, 2012.

[134] J. Nakashima and K. Taura. MassiveThreads: A Thread Library for High Productivity Languages. *Concurrent Objects and Beyond*, 8665:222–238, 2014.

[135] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin. Grappa : A Latency-Tolerant Runtime for Large-Scale Irregular Applications. Technical report, 2014.

[136] M. C. Ng and W. F. Wong. ORION: an adaptive home-based software distributed shared memory system. *Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on*, pages 187–194, 2000.

[137] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *International Journal of High Performance Computing Applications*, 20(2):203–231, may 2006.

[138] J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High Performance Remote Memory Access Communication: The Armci Approach. *International Journal of High Performance Computing Applications*, 20(6):233–253, 2006.

[139] G. Nishanov, J. Maurer, R. Smith, and Daveed Vandevoorde. P0057R7: Wording for Coroutines. Technical report, 2015.

[140] G. Nishanov and J. Radigan. N4402: Resumable Functions (revision 4). Technical report, 2015.

[141] R. Noronha and D. Panda. Designing high performance DSM systems using infiniband features. In *IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pages 467–474. IEEE, 2004.

[142] R. Noronha and D. K. Panda. Reducing Diff Overhead in Software DSM Systems using RDMA Operations in InfiniBand. In *RAIT 2004: Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies*, pages 1–8, 2004.

[143] R. Noronha and D. K. Panda. Can high performance software DSM systems designed with InfiniBand features benefit from PCI-express? *2005 IEEE International Symposium on Cluster Computing and the Grid, CCGrid 2005*, 2:945–952, 2005.

[144] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum*, 17(2):1–31, aug 1998.

[145] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa. Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system. *Proceedings - CCGrid 2003: 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 450–456, 2003.

[146] OpenFabrics Alliance. OpenFabrics. https://www.openfabrics.org/.

[147] C. Osendorfer, J. Tao, C. Trinitis, and M. Mairandres. ViSMI: Software distributed shared memory for infiniBand clusters. *Proceedings - Third IEEE International Symposium on Network Computing and Applications, NCA 2004*, pages 185–191, 2004.

[148] Y. Oyama, K. Taura, and A. Yonezawa. Executing Parallel Programs With Synchronization Bottlenecks Efficiently. *(PDSIA '99: Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications.*

[149] N. Papadopolou, L. Oden, and P. Balaji. A Performance Study of UCX over InfiniBand. In *CCGrid '17: Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 345–354, may 2017.

[150] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software Abstraction for Intel Memory Protection Keys. 2018.

[151] L. Peng, W. Wong, M. Feng, and C. Yuen. SilkRoad: a multithreaded runtime system with software distributed shared memory for SMP clusters. In *CLUSTER '00: Proceedings of IEEE International Conference on Cluster Computing*, pages 243–249, 2000.

[152] L. Peng, W. F. Wong, and C. K. Yuen. SilkRoad II: Mixed paradigm cluster computing with RC_dag consistency. *Parallel Computing*, 29:1091–1115, 2003.

[153] L. P. L. Peng, W. F. W. W. F. Wong, and C. K. Y. C. K. Yuen. SilkRoad II: a multi-paradigm runtime system for cluster computing. In *Proceedings of IEEE International Conference on Cluster Computing*, pages 443 – 444, 2002.

[154] G. C. Philos, V. V. Dimakopoulos, and P. E. Hadjidoukas. A runtime system architecture for ubiquitous support of OpenMP. *Proceedings of the 7th International Symposium on Parallel and Distributed Computing, ISPDC 2008*, pages 189–196, 2008.

[155] Plataformatec. Elixir. https://elixir-lang.org/.

[156] Python Software Foundation. Welcome to Python.org. https://www.python.org/.

[157] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors. In *ISCA '95: Proceedings of 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.

[158] B. Ramesh. Samhita: Virtual Shared Memory for Non-Cache-Coherent Systems, 2013.

[159] M. Rangarajan and L. Iftode. Software Distributed Shared Memory over Virtual Interface Architecture. *ALS'00: Proceedings of the 4th annual Linux Showcase & Conference - Volume 4*, 2000.

[160] S. Roghanchi, J. Eriksson, and N. Basu. ffwd: delegation is (much) faster than you think. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, pages 342–358, New York, New York, USA, 2017. ACM Press.

[161] A. Ros, B. Cuesta, M. E. Gomez, A. Robles, and J. Duato. Temporal-Aware Mechanism to Detect Private Data in Chip Multiprocessors. In *ICPP '13: 42nd International Conference on Parallel Processing*, pages 562–571. IEEE, oct 2013.

[162] A. Ros and S. Kaxiras. Complexity-Effective Multicore Coherence. In *PACT '12: Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, page 241. ACM Press, 2012.

[163] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *HPCA '12: Proceedings of the 18th International Symposium on High Performance Computer Architecture*, pages 1–12. IEEE, feb 2012.

[164] V. a. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A Theory of Memory Models. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 161–172, 2007.

[165] M. Sato, H. Harada, A. Hasegawa, and Y. Ishikawa. Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 2001.

[166] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. *Proc. of the 1st European Workshop on OpenMP*, pages 32–39, 1999.

[167] ScaleMP, Inc. Scalemp. https://www.scalemp.com/.

[168] D. J. Scales, K. Gharachorloo, and C. a. Thekkath. Shasta: a low overhead, software-only approach for supporting fine-grain shared memory. *ACM SIGOPS Operating Systems Review*, 30(5):174–185, 1996.

[169] Scheme Reports. Scheme Reports. http://www.scheme-reports.org/.

[170] P. Schmid, M. Besta, and T. Hoefler. High-Performance Distributed RMA Locks. *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing - HPDC '16*, (Dc):19–30, 2016.

[171] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castello, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kale, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman. Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[172] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *HOTI '15: Proceedings of IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, aug 2015.

[173] K. S. Shim, M. H. Cho, M. Lis, O. Khan, and S. Devadas. Library Cache Coherence. Technical report, MIT-CSAIL-TR-2011-027, 2011.

[174] M. Si, A. J. Pena, J. Hammond, P. Balaji, M. Takagi, and Y. Ishikawa. Casper: An Asynchronous Progress Model for MPI RMA on Many-Core Architectures. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 665–676. IEEE, may 2015.

[175] M. D. Sinclair, J. Alsop, and S. V. Adve. Chasing Away RAts : Semantics and Evaluation for Relaxed Atomics on Heterogeneous Systems. *Isca'17*, pages 161–174, 2017.

[176] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for GPU architectures. *IEEE Micro*, 34(3):69–79, 2014.

[177] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*, volume 6. 2011.

[178] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. *Operating Systems Review (ACM)*, 31(5):170–183, 1997.

[179] H. Sung. *DeNovo: rethinking the memory hierarchy for disciplined parallelism.* PhD thesis, 2015.

[180] H. Sung and S. V. Adve. DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations. *ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 545–559, 2015.

[181] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism. In *ASPLOS '13: Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, number 3, page 13, New York, New York, USA, 2013. ACM Press.

[182] Supercomputing Division, Information Technology Center, The University of Tokyo. Introduction to the Reedbush Supercomputer System. `https://www.cc.u-tokyo.ac.jp/en/supercomputer/reedbush/system.php`.

[183] I. L. Taylor. Split Stacks in GCC. `https://gcc.gnu.org/wiki/SplitStacks`, 2011.

[184] The Go Project. The Go Programming Language. `https://golang.org/`.

[185] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, 2008.

[186] The MassiveLogger Project. MassiveLogger. https://github.com/massivethreads/massivelogger/.

[187] The Open MPI Project. Open MPI. https://www.open-mpi.org/.

[188] Transaction Processing Performance Council (TPC). TPC-C. http://www.tpc.org/tpcc/.

[189] University of Versailles Saint Quentin en Yvlines. NAS-C-OpenMP3.0. http://benchmark-subsetting.github.io/cNPB.

[190] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó. Improving Concurrency and Asynchrony in Multithreaded MPI Applications using Software Offloading. In *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 30:1–30:12, 2015.

[191] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 256–266, 1992.

[192] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and RTM. *Symposium on Operating Systems Principles (SOSP)*, pages 87–104, 2015.

[193] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *IPDPS '08: Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium*, 2008.

[194] Y. Yao, W. Chen, T. Mitra, and Y. Xiang. TC-Release++: An Efficient Timestamp-Based Coherence Protocol for Many-Core Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3313–3327, 2017.

[195] Y. Yao, G. Wang, Z. Ge, T. Mitra, W. Chen, and N. Zhang. Efficient Timestamp-Based Cache Coherence Protocol for Many-Core Architectures. In *ICS '16: Proceedings of the 2016 International Conference on Supercomputing*, pages 1–13, New York, New York, USA, 2016. ACM Press.

[196] B.-H. Yu, Z. Huang, S. Cranefield, and M. Purvis. Homeless and home-based lazy release consistency protocols on distributed shared memory. In *ACSC '04: Proceedings of the 27th Australasian Conference on Computer Science*, volume 26, pages 117–123, 2004.

[197] X. Yu. *Logical Leases : Scalable Hardware and Software Systems through Time Traveling*. PhD thesis, Massachusetts Institute of Technology, 2017.

[198] X. Yu and S. Devadas. Tardis: Time Traveling Coherence Algorithm for Distributed Shared Memory. In *PACT '15: Proceedings of the 2015 International Conference on Parallel Architecture and Compilation*, pages 227–240. IEEE, oct 2015.

[199] X. Yu, H. Liu, E. Zou, and S. Devadas. Tardis 2.0: Optimized Time Traveling Coherence for Relaxed Consistency Models. In *PACT '16: Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pages 261–274. ACM Press, 2016.

[200] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *MICRO-42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, page 423, 2009.

[201] J. Zhang, B. Behzad, and M. Snir. Optimizing the Barnes-Hut algorithm in UPC. *SC '11: Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2011.

[202] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: Adaptive Granularity Cache Coherence. *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 547–558, 2013.

[203] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS Extension for C++. *IPDPS '14: Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114, may 2014.

[204] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. In *OSDI '96: Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 75–88, New York, New York, USA, 1996. ACM Press.