Université de Montréal

YADL: A General Purpose SDSM System

par

Jean-François Gagné

Département d'informatique et
de recherche opérationnelle

Faculté des arts et sciences

Mémoire présenté à la la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M. Sc.)
en informatique

Avril 2002

Université de Montréal

Faculté des études supérieures

Ce mémoire intitulé:

YADL: A General Purpose SDSM System

présenté par:

Jean-François Gagné

a été évalué par un jury composé des personnes suivantes:

Brigitte Jaumard

(présidente-rapporteure)

Marc Feeley

(directeur de maîtrise)

Jean Vaucher

(membre du jury)

Mémoire accepté le 16 juillet 2002

Abstract

Keywords: DSM, shared memory, parallel programming, NOW, cluster, programming library, memory consistency models.

Software distributed shared memory (SDSM) is an attractive tool to develop parallel applications. However, this facility is not widely used because of its low availability. Moreover, most of the available systems are either unstable or not flexible enough to allow widespread use. This research addresses these problems by proposing a new SDSM system, called YADL, that is flexible and extensible.

YADL supports multiple memory consistency models and the bag of tasks partitioning method. We explain how the utilization of this partitioning method can enhance performance of applications by achieving better load balancing in some situations. Moreover, this partitioning method allows the number of workers to change during the computation, which was forbidden by most previous SDSM implementations.

YADL also provides the static partitioning method traditionally found in SDSM systems. Integrating both partitioning methods in a single system and adapting the algorithms to support both methods is covered by this research.

After having briefly introduced the field of parallel programming and of SDSM systems, we present the architecture, the implementation and the use of YADL. Some benchmark results are also presented to illustrate the performance of the system.

French Abstract

Mots clés: DSM, mémoire partagée, programmation parallèle, NOW, grappe d'ordinateurs, librairie de programmation, modèle de consistance de mémoire.

Une mémoire partagée virtuelle (SDSM pour software distributed shared memory) est un outil facilitant la programmation d'applications parallèles. Malheureusement, cet outil est rarement utilisé à cause de sa faible disponibilité. De plus, la plupart des SDSMs disponibles sont soit instables ou ne comblent pas tous les besoins des programmeurs. Cette recherche aborde ces problèmes en proposant une nouvelle SDSM flexible et extensible nommée YADL.

Les aspects particuliers de YADL sont le support pour plusieurs modèles de consistance de mémoire et un modèle de programmation basé sur le "pool" de tâches. Nous expliquons comment l'utilisation de ce modèle de programmation peut contribuer, dans certains cas, à une meilleure utilisation des ressources. De plus, ce modèle permet d'ajouter et d'enlever des ressources de calculs durant l'exécution d'un programme, ce qui était impossible avec la plupart des SDSMs antérieures.

YADL fournit aussi un modèle de programmation statique comme celui traditionnellement inclus dans les SDSMs. L'intégration des deux modèles de programmation est couvert par cette recherche.

Après avoir brièvement introduit la programmation parallèle et les SDSMs, nous présentons l'architecture, les algorithmes d'implantation et l'utilisation de YADL. Une évaluation basée sur l'exécution de quelques programmes est aussi présentée afin de montrer son efficacité.

Contents

List of Figures

List of Tables

List of Abbreviations

Acronym	Description	First Use
API	Application Programing Interface	13
BER	Backward Error Recovery	38
DSM	Distributed Shared Memory	1
DVSM	Distributed Virtual Shared Memory	12
ERAHWU	Eager Release Annotated Home WU	61
ERVHWI	Eager Release VMH Home WI	62
ERVHWU	Eager Release VMH Home WU	61
HDLC	High Level Data Link Control	52
IP	Internet Protocol	52
NOW	Network of Workstations	1
OS	Operating System	2
SDSM	Software Distributed Shared Memory	1
SHL	Standard Homeless	64
TCP	Transmission Control Protocol	52
UDN	Universal Datagram Network	52
UDP	User Datagram Protocol	52
VM	Virtual Memory	22
VMH	Virtual Memory Hardware	21
WI	Write-Invalidate	23
WOAH	Write-Once Annotated Home	56
WOVH	Write-Once VMH Home	57
WU	Write-Update	23
YADL	Yet Another DSM Library	2

To all the people who taught me something.

Acknowledgments

First, I would like to thank my parents for supporting me in all my occupations, especially my studies. Since the beginning, they have always shown interest for my education. My father, Michel, taught me very early how to use computers by helping me learning my addition and multiplication tables with Lotus 1-2-3. He also introduced me to programming concepts when I was still young, keeping in mind that I needed to have fun, by making me draw castles and brick walls with Logo. My mother, Francine, devoted a lot of effort and showed lots of patience by assisting me learning my lessons at the end of elementary school and at the beginning of high school when I had a hard times with my studies. She has also always been available to help and support me when I had difficulties in my life. Thank to both of you who have greatly contributed to the success of my studies and of my life.

I would also like to thank my advisor, Marc Feeley, who provided me with a favorable research environment in which I had the freedom to explore different avenues of research. He also introduced me to SDSM, a topic I would probably not know without him.

All the people who contribute to this document also deserve to be thanked, especially the draft readers who contribute to the enhancement of the text. These readers are my advisor Marc Feeley, Éric Lesage, Michel Gagné and Étienne Bergeron.

Finally, I would like to salute my laboratory coworkers and friends with whom I spent lot of time in the past two and a half years. In particular Étienne Bergeron and Éric Lesage who shared most of my hobbies such as rock climbing, hiking, camping, scuba diving and others.

Funding for this work has been provided by the Natural Sciences and Engineering Research Council of Canada (NSERC), "Le Fond pour la Formation de Chercheurs et d'Aide à la Recherche" (FCAR), and Ericsson Canada.

Chapter 1

Introduction

When confronted with performance problems, programmers can use faster computers or can optimize their code. Once these methods no longer help, they can use parallelism to try to obtain faster solutions. However, developing a parallel program is usually much harder than developing a sequential program, restraining the use of parallelism.

Many parallel computer architectures are available. Among them, multiprocessor and network of workstations (NOW) are becoming popular. Prior to the development of a parallel application, programmers must choose their target architecture. This choice is guided by the program requirements and by the suitability of each architecture.

A distributed shared memory (DSM) is a tool making NOWs more suited for the development of parallel applications. While conserving their low cost, a DSM reduces the development costs on NOWs by providing a programming model similar to the one available on multiprocessors. With a DSM, a programmer can quickly develop a more elegant parallel program on NOW than when using traditional methods.

1.1 Goals

This work has been done in the context of software DSM (SDSM). No special hardware components are needed to operate a SDSM, allowing them to be used on almost any computer. The

goal of this work was to design and implement a SDSM system that could be used on the Linux operating system (OS), and possibly on all other UNIX-flavor OSes.

Other SDSM systems have been developed. However, even in applications where they could perform well, they are not widely used. This low use could be explained by the ignorance of their existence owing to their low availability. Moreover, currently available SDSM systems are either expensive, unstable research prototypes, incompatible with commonly available computer systems and OSes, or not distributed by their authors. Furthermore, no available SDSM system fits most user needs as they have been designed without concern toward general usability.

The main goal of this work is to address this lack of concern. The resulting SDSM should be able to address most current needs and should also be easily extensible to accommodate future requirements. To achieve this goal, the proposed SDSM must support a general-purpose interface and be easily extensible. Moreover, to be as flexible as possible, it should support many known and useful SDSM key concepts already presented in the literature.

The result of this work is a SDSM system, called YADL, that can be used to develop parallel applications. This name was chosen as a shortcut for "Yet Another DSM Library". YADL features support for multiple memory consistency models and the bag of tasks partitioning. The software package of this system is available on the author's web page (http://www.iro.umontreal.ca/~gagnjea/).

1.2 Contents

The first part of this document, composed of Chapters 2 and 3, consists of the technical foundations for the rest of the document. Chapter 2 presents the parallel programming models and architectures. Chapter 3 introduces DSMs, with emphasis on SDSMs.

The second part of the document, composed of Chapters 4 and 5, presents the SDSM system YADL. Chapter 4 explains the specification, implementation and utilization of YADL. Chapter 5 exposes the results of the evaluation of the system and presents future development.

Finally, the third and last part of the document, the Chapter 6, concludes by presenting the contribution of our work to the SDSM field and by discussing further avenues of research.

Chapter 2

Parallel Applications

A *parallel application* is composed of many abstract computation *instances* running in parallel. These instances can be implemented using system-specific threads or processes. To achieve the parallelization of a computation, the computation must be partitioned into tasks that are performed by the instances. Instances normally need to communicate. Instances must also be executed on a specific parallel computer architecture. These three topics, computation partitioning, inter-instance communication and parallel computer architecture, are discussed in the following sections.

2.1 Partitioning the Computation

The partition of the computation is an important part of the parallelization of an application. Without partitioning a computation, its parallelization cannot be achieved. Moreover, partitioning has an effect on execution time.

To parallelize a program, the computation to be executed must be partitioned into *tasks* that will be executed by instances. There should be at least as many tasks as there are instances. If not, one or more instances are idle and useless. If there are more tasks than instances, some instances execute more than one task.

The computation partitioning methods can be categorized in two classes: *static* and *dynamic*.

When statically partitioning a computation, the partition into tasks is known before starting the computation. When dynamically partitioning a computation, the partition process occurs at the program execution and can change according to run-time parameters.

The main quality of a good partitioning is *load balancing*. When the load is perfectly balanced, each computation resource (processors and communication channels) is used at maximum capacity. This does not mean that the resources are never idle, it means that when a resource is idle, it cannot do something that a busy resource is doing.

In some applications, load balancing can easily be achieved. However, in other applications, its achievement requires smart use of computation partitioning techniques. Using the right techniques needs careful thought and imagination, and is hard to achieved without proper knowledge of the application.

2.2 Inter-Instance Communication

In a parallel application, instances are collaborating to obtain the result of a computation. To collaborate, the instances must communicate. This communication can be achieved using different models. Two communication models are presented: the *shared memory model* and the *message passing model*. An *hybrid model* could be inferred from these two models, but it is not discussed.

2.2.1 Shared Memory Model

In the shared memory model, each instance can access a common repository of data. This repository is the shared memory. The operations that can be performed on the shared memory are LOAD and STORE, allowing respectively to consult and modify data.

The main advantage of the shared memory model is that the sharing of data structures between instances is easy. When data is placed in the shared memory, it can be consulted and modified by any instance. Developing a parallel application using this model is relatively easy but requires explicit use of synchronization primitives. The synchronization primitives usually provided for the development of shared memory programs are: mutexes, semaphores, barriers and condition variables ([HP96, Ste98, SGG02]).

Synchronization Primitives

Mutexes

Mutexes are used to enforce mutual exclusion. Only one instance can hold a mutex, allowing a single instance to execute critical code. This critical code is usually called a *critical section.* A "*lock*" on a mutex is a request to hold it, returning only when the mutex has been granted. An "*unlock*" on a mutex releases it, allowing another instance to "lock" it.

One of the uses of mutexes is to guarantee the atomicity of a sequence of otherwise non-atomic operations. When the atomicity of some particular sequences of operations is not guaranteed, the program behavior becomes unpredictable. An example of such a sequence is the addition of a constant to a variable where two operations, the read and the write to this variable should be atomic. If, when reading and updating the variable, a mutex was previously "locked", a single instance can modify the variable. This usage of the mutex removes the possibility of two instances updating the variable at the same time, eliminating the case where the program behavior is unpredictable.

Semaphores

Semaphores are a generalization of mutexes. Semaphores can be waited on ("*wait*"), released ("*signal*") and initialized. When a semaphore is initialized to N resources, N instances can hold the semaphore simultaneously. A mutex is equivalent to a semaphore initialized to one resource.

Barriers

Barriers are used to wait for instances at a particular moment in the program. When a barrier is "*reached*" by an instance, this instance is blocked. The barrier will be "*lowered*" when it is "reached" by a predefined number of instances. When the barrier is lowered, the blocked instances can resume their computation. *Global barriers* are barriers where the number of reaching instances needed before "lowering" the barrier is the total number of instances. Barriers, and especially global barriers, can be used to wait for initialization of data prior to a computation, or to wait that all instances are done before starting the next step of an iterative computation.

Condition Variables

Condition variables are used to avoid *active waiting.* An active wait occurs when an instance waits for an event using a loop. An active wait should be avoided because it needlessly consumes CPU cycles. To avoid losing these cycles, a "*wait*" on a condition variable blocks an instance. A "*signal*" wakes up a single instance currently waiting on the variable. A "*broadcast*" wakes up all instances currently waiting on the variable. Emphasis is put on "currently waiting" because a "broadcast" or a "signal" done prior to a "wait" does not have any effect. In this case, the waiting instance is blocked until the next "broadcast" or "signal".

Condition variables are often used in conjunction with mutexes. Instances usually "lock" a mutex prior to doing some operation (consume data). If the operations cannot be done because some other instances have not yet produced the data, the consuming instances would "unlock" the mutex and "wait" on a condition variable. However, if the data is produced between the "unlock" and the "wait" (the situation shown on the left side of Figure 2.1), the consuming instance is blocked because the "wait" has been done after the "signal".

To avoid this situation, some implementations of condition variables allow to atomically "unlock" a mutex at the same time a "wait" operation is performed. With this feature, the blocking can be avoided as shown on the right side of Figure 2.1.

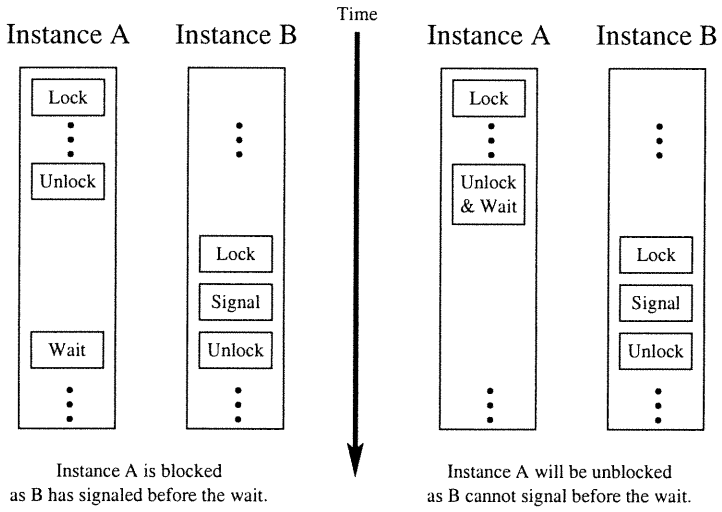

Figure 2.1: The Potential Blocking of Condition Variables.

Deadlocks

When using synchronization primitives, *deadlocks* can occur ([Sta97]). A deadlock is an endless wait that can be caused by a bad use of synchronization primitives. A deadlock could occur if a circular dependence exists when "locking" mutexes, if an instance never "reaches" a barrier, in the case of Figure 2.1 for condition variables, or in some other cases. Producing a deadlock is a situation that must be avoided at all costs. Programs without deadlock can be obtained by rigorous programming. An example of rigorous programming method to avoid deadlocks is, when needing to hold more than one mutex or semaphore, to "lock" or "wait" for them in the same order. With this method, no circular waiting can be generated ([Sta97]).

2.2.2 Message Passing Model

In the message passing model, the instances communicate by exchanging messages. There exists two communication primitives, RECEIVE and SEND, allowing to respectively get and emit messages.

In the pure message passing model, no shared memory is available. When two instances need to communicate, they must do it using messages. When a large data structure must be exchanged between instances, it must be converted to fit in one or more messages which are sent to their destinations.

The deadlock problem, presented in Section 2.2.1, can also occur when using the message passing model. If an instance A is blocked on a RECEIVE waiting data from instance B, but instance B is also blocked on a RECEIVED waiting data from instance A, a deadlock occurs.

The major drawback of the message passing model is its complexity of use due to the additional code needed to explicitly manage communication between instances ([HP96]). When using message passing, the resulting application source code is larger and harder to understand than when using shared memory. Message passing libraries, as MPI ([mpi]), have been proposed to ease the development of message passing applications. However, developing parallel applications using a message passing library is still usually harder than using shared memory.

Despite its drawbacks, message passing is used because it is the only communication model available on some parallel computer architectures.

2.3 Parallel Computer Architectures

A parallel computer architecture is composed of many processors executing instances in parallel. The two most popular architectures are *multiprocessor* and *network of workstations* (NOW). These two architectures are presented in this section.

2.3.1 Multiprocessor

In a multiprocessor, also known as a tightly coupled system, all the processors can access the same physical shared memory. To implement this type of architecture, the processors can share the same bus to the memory ([HP96]) as shown in Figure 2.2.

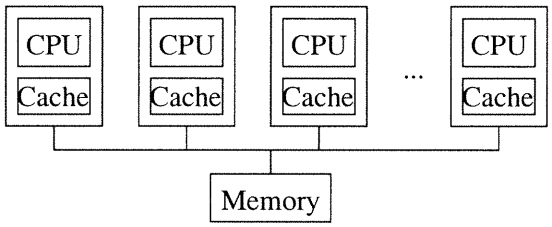

Figure 2.2: The Multiprocessor Architecture.

For performance reasons, each processor has its own local cache. This design improves the performance but also introduces the problem of *cache coherence*. When a processor modifies data residing in other caches, these caches must be updated or invalidated to keep a processor from reading old data. Since all the processors share the same bus, each cache can listen on the bus for STOREs from other processors and update themselves accordingly. This method is known as snoopy caching ([HP96]).

The natural inter-instance communication model for multiprocessors is the shared memory model as a physical shared memory is provided by the architecture. Message passing can also be used, but its use is less common because the shared memory model is usually easier to use. With the availability of a shared memory, developing a parallel application on a multiprocessor is relatively easy.

The drawback of the multiprocessor architecture is the limit on the number of processors. This limit is due to the interconnections between processors and memory that do not scale

well. A bus could be used for two or four processors, but, when using more processors, the bus contention becomes a bottleneck. This problem can be reduced by using more complex interconnection network, involving multiple buses to the memory. These interconnections are expensive, increasing the cost of multiprocessors composed of tenth of processors.

2.3.2 Network of Workstations

A network of workstations (NOW), also known as a loosely coupled system or a cluster, is composed of many computers, called *nodes*. The nodes are linked by a network that can be Ethernet ([Hal96]), Myrinet ([myr]), or any other. As the node and the network are usually made from off-the-shelf components, the cost of a NOW is low. Moreover, the number of nodes in a NOW scales well as networks can contain hundreds of nodes.

The nodes of a NOW do not share physical memory. Each node has its own private memory that can only be accessed by the local node as shown in Figure 2.3[1]

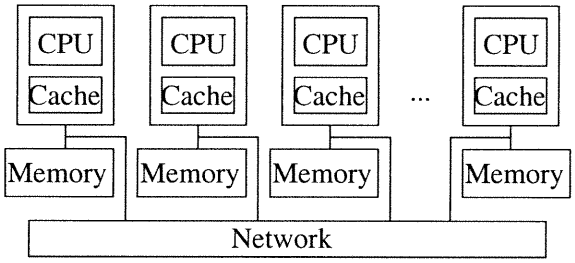

Figure 2.3: The Network of Workstations Architecture.

Without a shared memory, the only communication model available on NOWs is message passing. Since the use of this model is complex, the development of a parallel application on NOWs is also complex. Moreover, the node interconnections in NOWs are less efficient than in multiprocessors. This causes performance problems in applications where a lot of communication is required. However, as they are less expensive, NOWs are largely used in applications with low to moderate communication requirements.

[1] Some exceptions exist when the network interface provides functions to access remote memory, but these are not considered for the general case of NOWs.

2.4 Summary

This chapter was a brief introduction to parallel application development. Computation partitioning, the inter-instance communication models and the parallel computer architectures were introduced.

Partitioning the computation is essential to use parallelism. When partitioning properly the computation, load balancing is achieved, which is a desirable characteristic of a good computation partition.

Two communication models, shared memory and message passing, were presented. Shared memory is more attractive than message passing due to its relatively simple use.

Only multiprocessors provide shared memory. NOWs are cheaper and can incorporate more processing power. Because they often only have access to a NOW, programmers must reluctantly accept the use of message passing to develop parallel applications.

The next chapter is an introduction to distributed shared memory (DSM). A DSM is an emulation of shared memory on a computer architecture that does not provide physical shared memory. Its use eases the development of parallel applications on NOWs by simulating a shared memory.

Chapter 3

Distributed Shared Memory

Computationally intensive applications are often run on networks of workstations (NOWs) in the hope of getting a cost-effective solution. However, the effort required to develop and maintain such applications can be considerable as the message passing communication model, the only model available on NOWs, is complex to use (see Sections 2.2.2 and 2.3.2).

Distributed shared memory (DSM) has been proposed as a means to ease the development of parallel applications on NOWs. A DSM provides a virtual shared memory on top of the message passing model. With the shared memory provided by the DSM, a programmer can design an application to be run on a NOW using an inter-instance communication model very close to the shared memory model. This alternative to the message passing model is attractive due to its easier usage.

This chapter presents the field of DSMs. Because this research focuses on them, emphasis is put on software DSM (SDSM). The first section defines DSMs. This sections is followed by a section on memory consistency models. The implementation and utilization of SDSM systems are then presented. Other subjects in the SDSM field are also introduced. Finally, the niche of SDSM is discussed.

3.1 DSM Definition

DSMs were introduced in the mid 1980s. Since the beginning, the goal of DSMs has been to ease the development of parallel applications by providing a shared memory on NOWs ([CBZ91, FP89, Li88]). With a DSM, a programmer can design a parallel application on NOWs using an inter-instance communication model very close to the shared memory model.

As the nodes from a NOW do not share physical memory, DSMs must be implemented on top of the message passing model. As the shared memory does not physically exist and is implemented over message passing, it is sometimes called a *distributed virtual shared memory* (DVSM). The DSM does not eliminate the message passing model from NOWs, it provides a layer hiding the message passing model to programmers.

In DSMs, each instance of the computation has a local cache of the shared memory. Since two instances could cache the same data, the problem of cache coherence must be addressed (see Section 2.3.1). How the caches are kept coherent is defined by a memory consistency model. The general problem of enforcing cache coherence has been studied prior to the introduction of DSMs. The previously proposed solutions, such as snoopy caching (see Section 2.3.1), could be applied to DSM, but they do not result in an efficient implementation. These bad results are explained by the less efficient node interconnections in NOWs. Achieving efficient cache coherence is the main challenge in DSM implementations.

Providing classic memory consistency models does not always result in an efficient DSM. The reason for this poor performance is that classic memory models, like the sequential model, require lots of communication to achieve cache coherence. Since communication resources must not be wasted in NOWs, memory consistency models needing less communication, such as the release and lazy release model, has been introduced (see Section 3.2).

There exists three types of DSMs classified according to the way they are implemented: hardware, software and hybrid. A *hardware DSM* is totally implemented in hardware. A *software DSM*, also known as a SDSM, is totally implemented in software, using standard network hardware. A *hybrid DSM* is implemented using both custom hardware and software components.

Hardware DSMs are usually more efficient as they are implemented with dedicated optimized components. However, they are expensive DSMs due to the custom elements needed for their implementation.

SDSMs are less expensive, but also less efficient as they do not benefit from specific hardware optimizations. However, SDSMs are attractive due to their easy and quick implementation. SDSMs can include more complex algorithms, are easy to tune, enhance and customize, and are the only option on off-the-shelf systems ([Sco00]). For these reasons, SDSMs are also a good research platform because new algorithms can easily be tested and evaluated.

Hybrid DSMs take the best from both worlds: the optimization of hardware DSM for expensive operations and the flexible environment of SDSMs. Hybrid DSMs probably have the most promising future as they will probably provide the highest performance-price ratio.

Software DSM systems could be again classified in three classes according to "where" they are implemented. SDSM systems could be implemented at *kernel level*, *user level*, or user level with minimal kernel modifications.

A SDSM system implemented at kernel level is not limited by the fewer permission given to user processes. It has access to all system resources, such as advanced features of the network and of miscellaneous hardware. Moreover, a kernel implementation could benefit from direct accesses to network resources, reducing the overhead of using a multi-layers network. However, modifying an OS kernel is a complex task. Furthermore, the kernel modifications done to implement the SDSM system could be incompatible with the next version of the kernel, voiding much of the work done.

A user level implementation is the easiest way to implement a SDSM system as it is usually easy to implement a user library. Moreover, a user library is more likely to be compatible with future versions of the system, especially if its implementation follows standards such as the Single UNIX Specification ([sin]). However, it must rely on OS services and must use the OS API (application programming interface) to access these services. The use of this API might be expensive as it could need a context switch to kernel level.

If the cost of using the OS API forbids the efficient implementation of the SDSM, or if a desired function is not provided to user processes, some extension to the kernel could be made to optimize or provide the needed function. This approach is more likely to be compatible with future versions of the kernel. Even if a next version is not compatible with these modifications, porting them should be easier and faster than porting an entire SDSM system.

DSM is a new programming tool for the development of parallel application on NOWs. They

will not replace multiprocessors, nor message passing. DSMs keep some of the drawbacks of the NOW architecture such as slow communication between nodes. They also cannot reach the same level of performance as well tuned message-passing applications. However, they could perform well in some applications. The niche of SDSM systems is described in more details in Section 3.6.

3.2 Memory Consistency Models

When there exist many caches for the same memory, coherence among the caches must be enforced. How the caches are kept coherent is specified by a *memory consistency model.* As DSMs maintain a cache of the shared memory in each instance, the cache coherence problem cannot be avoided. Thus, memory consistency model is a fundamental concept in the DSM field.

There are many memory consistency models. Some models provide the programmer with simple but inefficient memory accesses. Others provide more efficient accesses at the cost of ease of use.

Memory consistency models can be classified according to the extent of restrictions specified on consistency ([HP96]). A model with lots of restrictions is a strict model. A model with few restrictions is a relaxed model. Usually, strict models are easy to use, but are also less efficient as they require lots of operations to achieve consistency. More relaxed models are harder to use as they give more burden on the programmer, but their correct use results in performance improvement.

Many memory consistency models have been defined, but only the one related to SDSMs are discussed. These models are the sequential, processor, release and lazy release models. However, before discussing them, notation that gives us a framework to understand and compare the different models must be introduced.

3.2.1 Notation

To better understand memory consistency models, the notation presented in [SD86] is used. This notation was first used in the context of multiprocessors to define memory consistency models.

The original version of the definitions refer to processors. To adapt them to NOWs, the references to a processor must be replaced by references to a node. The original definitions are presented in this document.

The [SD86] notation defines memory request initiation, issue, performance, performance with respect to a processor and global performance. The first three definitions follow:

Initiating a memory request: A memory access request is *initiated* when a processor has sent the request and the completion of the request is out of its control.

Issuing a memory request: An initiated request is *issued* when it has left the processor environment, including the CPU and local buffers, and is in transit in the memory system.

Performing a memory request: A STORE is considered *performed* at a point in time when a subsequently issued LOAD to the same address returns the value defined by this or a subsequent STORE. A LOAD is considered *performed* at a point in time when the issuing of a STORE to the same address cannot affect the value returned by the LOAD.

As explained in [SD86], these three definitions are relevant whether or not memory accesses are atomic. In this case, atomicity means that the value modified by a STORE operation becomes accessible at the same time for all processors. An example where memory accesses could be atomic is some particular implementations of multiprocessors.

When atomicity cannot be enforced, a memory request could be performed at a processor a but not yet at processor b. This situation is frequent in DSMs: the cache of instance a is often updated before the cache of instance b, introducing a small interval of time where a and b may not read the same value for a variable in the shared memory. This situation is usually caused by the impossibility to guarantee the reception of messages at different destinations at the same time in a NOW.

To also consider memory accesses that are not atomic, [SD86] introduces the performance with respect to a processor and the global performance. These two definitions follow:

Performing a memory request with respect to a processor: A STORE by processor i is considered *performed with respect to processor* j when a sub-

sequent LOAD to the same address by processor j returns the value defined by this or a subsequent STORE. A LOAD by processor i is considered *performed with respect to processor j* when a subsequent issued STORE to the same address by processor j cannot affect the value returned by the LOAD.

Performing globally a memory request: A STORE is *globally performed* when it is performed with respect to all processors. A LOAD is *globally performed* when it is performed with respect to all processors and when the STORE which is the source of the returned value has been globally performed.

Performed a memory request with respect to all processors is equivalent to the previous definition of performing a memory request. Thus, after a STORE is globally performed, which is equivalent to be performed with respect to all processors, no subsequent LOAD can return an older value from the one written by the STORE.

A subtle difference between a LOAD performed with respect to all processors and a globally performed LOAD must be pointed out. Once a LOAD is performed with respect to all processors, the value returned is locally fixed and cannot be altered independently of any action from any processor. Thus, performing a LOAD with respect to all processor is an operation that could be considered local to the current processor because it does not impose restriction to other processors.

When a LOAD is globally performed, the returned value is locally fixed and cannot be altered (performed with respect to all processors), and *any* other LOAD issued subsequently by *any* processor cannot return a value that is older than the value returned by the globally performed LOAD. Thus, performing a LOAD globally is stronger than performing a LOAD with respect to all processors because it is an operation that restricts the behavior of future LOADs on all processors.

3.2.2 Sequential Model

The *sequential memory consistency model* is a strict model introduced in [Lam79] in the context of multiprocessors:

"[A system is *sequentially consistent* if] the result of any execution is the same as

> if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear, in this sequence, in the order specified by its program."

The conditions needed to enforce sequential consistency, using the notation from Section 3.2.1, are ([GLL+90]):

1. before a LOAD is allowed to perform with respect to any other processor, all previous LOAD and STORE accesses must be globally performed;
2. before a STORE is allowed to perform with respect to any other processor, all previous LOAD and STORE accesses must be globally performed.

The sequential memory consistency model is a strict model as it specifies lots of restrictions on the order of LOAD and STORE. As presented in Section 3.3.3, the sequential model is also the less efficient model because of the restrictions that must be enforced in its implementation. However, with sequential consistency, a programmer knows exactly the behavior of his program. Moreover, no special annotation of the source code is required to enforce consistency, contrary to some relaxed models.

3.2.3 Processor Model

The *processor memory consistency model* is more relaxed than the sequential model. It was introduced in [Goo91] again in the context of multiprocessors:

> "[A system is *processor consistent*] if the result of any execution is the same as if the operations of each individual processor appear in the sequential order specified by its program."

The conditions needed to enforce processor consistency, using the notation from Section 3.2.1, are ([GLL+90]):

1. before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed with respect to all processors;

2. before a STORE is allowed to perform with respect to any other processor, all previous LOAD and STORE accesses must be performed with respect to all processors.

Comparing the conditions for processor consistency with the conditions for sequential consistency, the following differences can be noticed:

- Using processor consistency, a STORE access s preceding a LOAD access l does not need to be globally performed, nor performed with respect to any other processors before l can be performed with respect to any other processor;
- Using processor consistency, LOAD accesses do not need to be globally performed, but only performed with respect to all processors, before a subsequent access is allowed to perform with respect to any other processor.

These differences explain why the processor consistency model is more relaxed than the sequential consistency model. The implementation of the processor consistency model can be more efficient because of these fewer restrictions.

[Goo91] presents an example where the results using processor consistency differ from sequential consistency, but this program is not very useful. It is also pointed out in [Goo91] that some multiprocessors do not provide sequential consistency but processor consistency.

3.2.4 Release Model

To allow more pipelining and buffering of memory accesses, the *release memory consistency model* was introduced in [GLL+90]. If used properly, release consistency can obtain the same computation results as sequential consistency usually using less communication resources and usually causing less delay. However, the release model requires the programmer to use annotation to enforce consistency.

In the release model, accesses are categorized in a hierarchical way. The hierarchy of access types is given in Figure 3.1.

Competing accesses are defined as accesses to the same memory location (more detail about the location is given in the implementation section) that could execute simultaneously where at

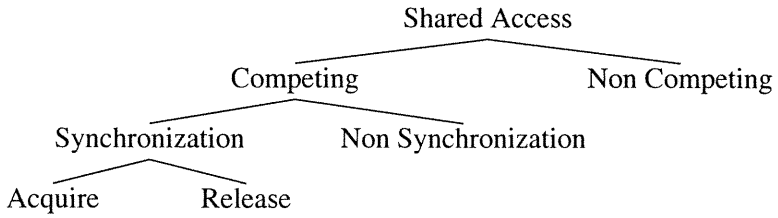

Figure 3.1: Shared Accesses in the Release Memory Consistency Model.

least one access is a STORE. Synchronization accesses are competing accesses that are used to order competing accesses (as mutexes). Acquire and release accesses are used to gain access to shared locations and grant this permission respectively ("lock" and "unlock" on mutexes).

Having defined memory access types, a way to specify them in the program must be supplied. For this purpose, a label for each access is defined. The labels corresponding to the access type are given in Figure 3.2. An access labeled $\mathcal{L}$ is also implicitly labeled with all the labels higher than $\mathcal{L}$ in the label hierarchy (a "sync" access is also labeled "special" and "shared").

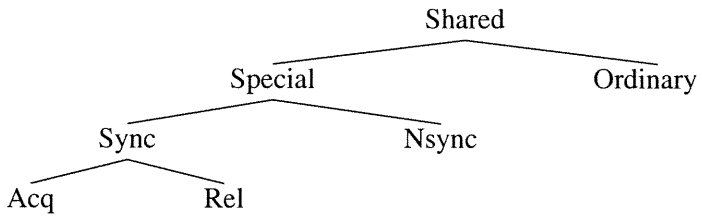

Figure 3.2: Labels for Shared Accesses in the Release Memory Consistency Model.

The formal definition of the release memory consistency model is based on the access and label notions. The conditions needed to enforce release consistency, using the notation from Section 3.2.1, are ([GLL+90]):

1. before an "ordinary" LOAD or STORE access is allowed to perform with respect to any other processor, all previous "acquire" accesses must be performed with respect to all processors;
2. before a "release" access is allowed to perform with respect to any other processor, all previous "ordinary" LOAD and STORE accesses must be performed with respect to all processors;
3. "special" accesses are processor consistent with respect to one another.

Release consistency allows all "ordinary" STORE accesses preceding a release to be pipelined or buffered until the moment the "release" performs with respect to another processor. At this moment, all the "ordinary" STOREs that have been initiated prior to the "release" must be performed with respect to all processors.

When "acquire" and "release" are considered as "lock" and "unlock" on a mutex, the model becomes very clear and simple: the "ordinary" STOREs done prior to the "unlocking" of a mutex must be performed with respect to all processors when a processor subsequently "locks" the mutex.

3.2.5 Lazy Release Model

The *lazy release memory consistency model* has been developed specifically for SDSMs. It was introduced in [KCZ92]. This model is derived from the eager release model[1]. It keeps the notions of access types and labels of the eager release model, but modifies the conditions to achieve consistency. These new conditions are:

1. before an "ordinary" LOAD or STORE access on processor i is allowed to perform with respect to processor j, all previous "acquire" accesses on processor i must be performed with respect to processor j;
2. before a "release" access r on processor i is allowed to perform with respect to processor j, all "ordinary" LOAD and STORE accesses on i prior to r must be performed with respect to processor j;
3. "sync" are sequentially consistent with respect to one another.

The improvement of the lazy release model over the eager release model is to allow a "release" access to be performed with respect to processor i without having to perform previous "ordinary" accesses globally, but only with respect to processor i. Propagating only the modifications to the next acquiring processor, not to all processors, reduces the use of network resources.

As for the eager release model, when "acquire" and "release" are considered as "lock" and "unlock" on a mutex, the model becomes very clear and simple: the "ordinary" STOREs done

[1] To simplify the discussion and clearly distinguish the two release models, the original release model is called the eager release model.

prior to the "unlocking" of a mutex must be performed with respect to the next "locking" processor when this processor "locks" the mutex.

Lazy release consistency is the model closest to message passing. Almost like in message passing, when using the lazy release model, the only processor receiving data is the one requiring it. However, implementing the lazy release model is much harder than any other model, as presented in Section 3.3.3.

3.3 SDSM System Implementation

In this section, an overview of the implementation of SDSM systems is given. The aspects covered are the detection of shared memory accesses, the different types of protocols for the implementation of synchronization primitives and shared memory, the implementation of the memory consistency models, and the implementation of the synchronization primitives.

From now on, a SDSM computation instance will be called a *worker*. To access cache coherence and synchronization services, the worker must have access to a *manager* that provides these services. This manager, which implements the SDSM, can be merged in the worker, or implemented as a distinct module. Each worker can invoke the manager using the SDSM API.

3.3.1 Detecting Shared Memory Accesses

Detecting accesses to the shared memory is an important part of a SDSM system as it is the foundation of the implementation of memory consistency models. When a shared memory access is performed, consistency operations must be executed to enforce the model which is applied to the accessed memory.

There exist two detection methods: *annotations* and the *virtual memory hardware* (VMH). Both methods have their advantages and drawbacks.

Annotations

When using annotations to detect shared memory accesses, code is added to the original program to inform the manager of shared memory accesses.

Annotation can provide the manager with precise information about the memory accesses performed and the memory ranges accessed. However, inserting annotations manually is a task that programmers usually dislike and that is prone to mistakes.

An existing compiler could be modified to ease the addition of annotations. With a modified compiler, the programmer only needs to declare that a variable is part of the shared memory, the compiler does the rest of the job.

Virtual Memory Hardware

The other way to detect shared memory accesses is by using the virtual memory (VM) hardware included in computers. This hardware allows the protection of the memory at the granularity of VM pages. When a forbidden access is performed, the OS is notified by a page-fault interruption. Upon receiving a page-fault, the OS can take any appropriate action, such as inhibiting the protection to allow the program to resume its execution. These page-faults can even be trapped outside the OS kernel in application level code (see Appendix A).

The VMH is an attractive access detection method as it does not need involvement from the programmer to annotate his program. However, the use of the VMH can be more costly than annotations because hardware page-faults need the interruption of the running program followed by a context switch to the operating system. This context switch results in the loss of CPU cycles, and even worse, a potential loss of cache state. Moreover, the VMH cannot manage access to a smaller granularity than VM pages, which is usually large (4 kilobytes on Intel x86), and does not provide information about the end of memory accesses.

3.3.2 Protocols

A protocol prescribes how a specific part of the SDSM is implemented. Such protocol can be applied to memory consistency models and to synchronization primitives.

In this section, the general notions needed to classify the implementation memory consistency models and synchronization primitives are described. These notions will be needed in Sections 3.3.3 and 3.3.4 regarding the implementation of memory consistency models and synchronization primitives.

Centralized vs Distributed

A choice to be made for the implementation of both synchronization primitives and memory models is one between a *centralized* or *distributed* (decentralized) protocol. In a centralized protocol, a single worker, known as the *home*, answers or redirects requests from all workers about a specific memory range or synchronization primitive. In a distributed protocol, the responsibility of answering requests is spread between all the workers and usually migrates with data movements.

The implementation of a centralized protocol is easy and seems efficient as a single request results in a quick response. However, a centralized protocol lacks scalability as the home could become a bottleneck if lots of requests are made simultaneously.

Distributed protocols are more scalable, but are also more complex to implement. Each worker usually has a hint to the *temporary home*, the worker currently managing a specific memory range or synchronization primitive. A request is usually done using this hint. The request is forwarded by intermediary workers up to the temporary home. As distributed protocol uses extensively request forwarding, they could suffer from higher delays for serving requests. Moreover, care must be taken to avoid cycles in the hints. As they are more scalable than centralized protocols, distributed protocols are usually a better choice than centralized protocols in the implementation of synchronization primitives or memory consistency models.

Write-Update vs Write-Invalidate

The *write-update* (WU) and *write-invalidate* (WI) protocols only apply to memory consistency models. In a write-update protocol, complete data about a consistency operation follows the message about this operation. In a write-invalidate protocol, the complete information about a consistency operation stays at the level of the worker that initiates the operation. Minimal informations (invalidations) are sent to other workers, resulting in potentially less network use. When a LOAD is performed on invalidated memory, the loading worker has to fetch a valid copy of the data from the other workers.

WU protocols are usually easier to implement than WI protocols as they do not require data to be maintained locally in a worker. Moreover, when using a WU protocol, LOADs are local operations because the local cache of a worker is always valid. However, as WU protocols use lot

of network resources to maintain all caches valid, consistency operations could be slower than WI protocols. Moreover, again because of extensive network usage, WU protocols could scale badly. To be worthy, the high cost of STOREs of a WU protocol must be distributed on lots of LOADs. This will be the case when the read-write ratio of an application is high (data is read more often than it is written).

In situations where the read-write ratio is low or when the newly written data is not read by many workers before being modified, a WI protocol is more efficient because less useless data is sent on the network resulting in faster completion of consistency operations. However, using a WI protocol, LOADs are potentially expensive and high latency operations: if the cache of a worker has been invalidated, it generates a network request to obtain a valid copy of the data. Moreover, as workers must answer future data request from other workers, complex data structure allowing to serve these requests must be locally maintained, increasing the complexity of implementation of WI protocols.

3.3.3 Implementing Memory Consistency Models

The efficient implementation of the memory consistency models is crucial to achieve good performance. In this section, the implementation of the sequential, eager release and lazy release memory consistency models are presented with the two protocols write-update and write-invalidate. The processor model is not presented since it is not widely used in SDSM systems.

Before beginning the discussion about the models, basic notions about memory consistency must be introduced: the granularity and the granule. The *granularity* is the coherence unit of the memory. When operations are performed on shared memory, they are performed on at least one *granule*. A small granularity, 512 bytes or less, should be used when working on small data structures. When using large data structures, a larger granularity should be used to reduce the overhead of granule management. When using the VMH to detect memory accesses, the smallest granularity that can be used is the size of a VM page. On Intel x86, as the page size is 4 kilobytes, the smallest granularity available is 4 kilobytes. This could be too large for some applications. As pointed out in Section 3.3.1, using the VMH to detect accesses always results in large granularities.

Sequential Model

The classic way to implement the sequential memory model is a multiple-readers single-writer method. This means that a single worker can write to a granule, introducing the problem of false sharing (discussed later in this section).

For each granule, there are two possible global states:

Single-writer: The granule is only valid in the cache of a single worker cache which is readable and writable. All the other caches are invalid.

Multiple-readers: The granule can be valid in many caches, but is read-only in all of them. Some workers can also have invalid cache.

This global state is managed using a distributed protocol. The only worker which is aware of this state is the owner. This owner is always the last worker that had a valid cache in the single-writer state. Each worker has a hint to the owner. All requests of a worker are sent using this hint, and the request are forwarded up to the owner.

In each worker cache, the granule can be in the local state valid or invalid. When the shared memory is created, the granule is in the global state single-writer, and in the local state valid in the owner and invalid in all other workers. Then, the owner, the global state and local state are managed according to the following rules:

- When, in the global state single-writer, a worker caching an invalid granule needs to write to the granule, the invalid granule is promoted to valid after being updated using the granule at the owner, the granule of the owner is demoted to invalid, the requesting worker becomes the owner, and the hint of the previous owner is updated to the new owner.
- When, in the global state single-writer, a worker caching an invalid granule needs to read to the granule, the invalid granule is promoted to valid after being updated using the granule at the owner, the granule of the owner stays valid but becomes read-only, and the global state becomes multiple-readers.
- When, in the global state multiple-readers, a worker caching an invalid granule needs to read to the granule, the invalid granule is promoted to valid after being updated using the granule at the owner (or any other valid cache).

- When, in the global state multiple-readers, a worker caching an invalid granule needs to write to the granule, the invalid granule is promoted to valid after being updated using the granule at the owner (or any other valid cache), all the other granules in the other workers are demoted to invalid, the requesting worker becomes the owner, and the hint of the previous owner is updated to the new owner.
- When, in the global state multiple-readers, a worker caching a valid read-only granule needs to write to the granule, all the other granules in the other workers are demoted to invalid, the requesting worker becomes the owner, and the hint of the previous owner is updated to the new owner.

The previous description of the algorithm implements a write-invalidate protocol. It is possible to modify the algorithm to implement a write-update protocol. To do so, when changing from the state single-writer to multiple-readers, all the invalid granules must be updated to become valid. Therefore, the WU version of this algorithm consumes lots of network resources, which explains why it is not implemented in many SDSM systems.

The implementation of sequential consistency potentially results in *false sharing*. False sharing occurs when different workers are forbidden to access different data residing on the same granule. As, in this implementation, a single worker can write to a granule, the other workers needing to read or write to the same granule cannot progress, resulting in poor performance. Moreover, when using a VMH approach to implement the model, two nodes writing to the same granule can exchange back and forth the write permission, causing lot of network requests that results in trashing.

To avoid this ping-pong effect, the SDSM system Mirage has introduced a minimum ownership time Δ ([FP89]). When getting read or write permission on a granule, this permission is kept at least Δ to enforce locality. This optimization reduces the trashing, but does not always eliminate it. Moreover, Δ must be correctly tuned to enforce locality without eliminating parallelism. As the optimal value of Δ could be application, processor-speed, network load, bandwidth and latency specific, tuning it optimally is difficult.

The algorithm previously presented has been designed to be implemented using the VMH. Indeed, in an annotated approach, the transition may not be taken at any time as it is impossible to change the permissions on memory ranges while access to it has been granted. As the programmer supposes that he has permissions until he notifies the system that he is done with

a memory access, dynamically removing read or write permissions should not be done. Thus, changing the global state of the temporary home can only be done at the end of an access.

To implement the annotated version of the sequential consistency, a pending access queue must be added to the algorithm to store accesses that cannot be granted immediately. This queue introduces the deadlock problem: if two workers a and b have granule x and y in read mode respectively and they request writing on y and x respectively, a deadlock occurs because the privileges cannot be dynamically removed nor granted. These problems, combined with the aversion of programming with annotations, could explain why the annotated implementation of sequential consistency has not been favored by implementers and has not been found in any studies SDSM systems.

Traditionally, the whole granule is sent when data update is needed. The resulting high network use explains why WU implementations of the sequential model have not been favored by implementers. Nevertheless, using twins and diffs methods to transfer updates as explained in the next section could reduce network use, diminishing the overhead of WU protocols. However, as the eager and lazy release models have mostly replaced the sequential model, research on the improvement of the sequential model is not extensive.

Eager Release Model

For the implementation of the release models (both eager and lazy), the *twins* and *diffs* notions are needed. A diff corresponds to modifications to the shared memory that is composed of a list of offset, length and data tuples. A twin is an image of the memory before a write operation. Twins are used to compute diffs.

To implement eager release consistency, the notion of release is needed. In the simplified version of the eager release model implemented in MUNIN ([CBZ91, Car94]), all the accesses are either ordinary or release. The ordinary accesses are performed with respect to all workers at the next release. This is a conservative implementation of the release model because the accesses are performed at the time of the release, and not at the moment the "release" is performed with respect to another worker.

The implementation of the eager release model follows this algorithm:

1. On the first write to a shared memory granule, a twin copy of the granule is taken, all the subsequent write proceeding normally;
2. On a "release", the actual copy of the memory is compared with the twins to compute the diffs;
3. The diffs (or corresponding invalidations) are sent to the other workers to update (or invalidate) their caches;
4. The "release" can only return when all the diffs (or invalidations) have been acknowledged by all workers.

This implementation of the eager release model allows concurrent writers to the same granule (multiple-writers), avoiding false sharing. If workers are writing to different sections of memory, the diffs will not overlap and the memory will be kept coherent as shown in Figure 3.3. If two workers are writing to the same data at the same time, the diffs could overlap and the result of the computation is undefined.

The overlapping of diffs depends of the diff unit. The size of a diff is always a multiple of the diff unit. The diff unit specifies what is the "same memory location for competing accesses" used in Section 3.2.4 . If the diff unit is small (1 or 2 bytes), concurrent writers can modify small data like characters of short integers. However, if the diff unit is large (4, 8 or more bytes), concurrent writers cannot modify consecutive characters because the diffs will overlap. The diff unit is often specified as 4 bytes in many SDSMs, but its exact value is specific to the SDSM system used.

To implement eager release consistency using a WU protocol, the diffs are sent to all workers at the "release". When implementing the eager release model using a WI protocol, only invalidations are sent, requiring the releasing worker to locally store the diffs that will be sent when workers will read the invalidated granules.

To remove the need to store diffs locally, a home implementation of the WI protocol could be used. This implementation is derived from the home implementation of the lazy release model ([ZIL96]). At the "release", invalidations are sent to all workers caching the memory, and diffs are only sent to the home. When a worker does a LOAD on an invalidated granule, it requests the granule data from the home. The use of this home implementation of the WI eager release model eliminates the need for storing diffs locally and reduces the memory consumption of the implementation. Moreover, as the home always has a valid copy, the LOADs done by

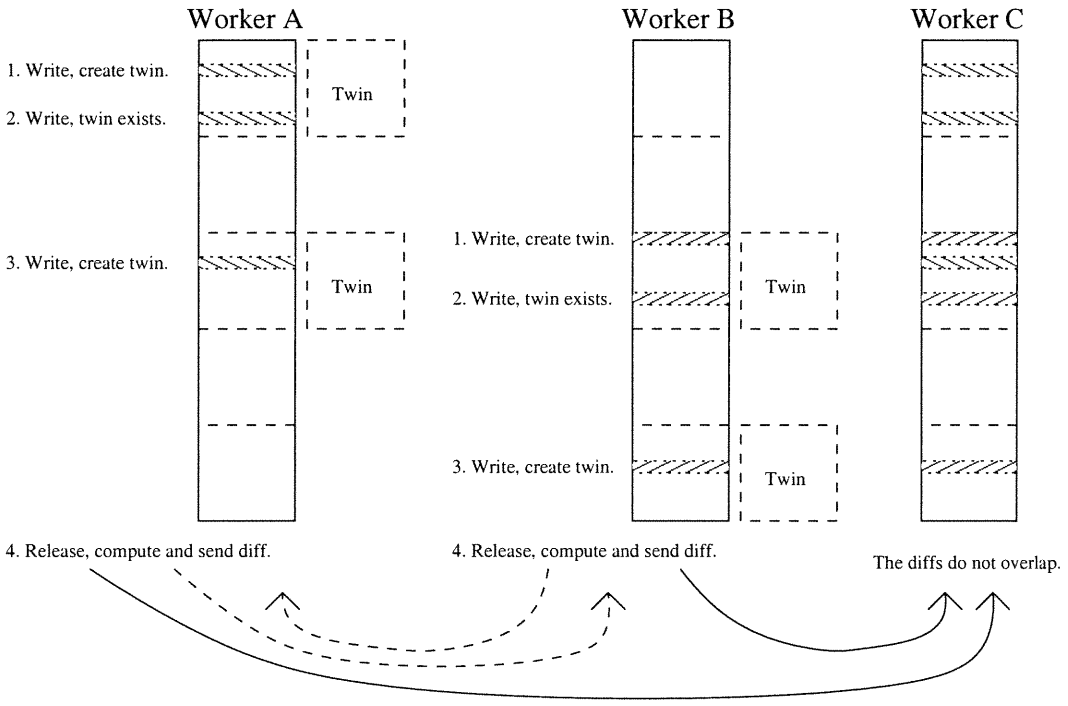

Figure 3.3: The Consistency Operations of the Eager Release Model.

the home are always local operations, which could be an interesting feature to take advantage from. However, as diffs must be sent to the home, more network resources are consumed and consistency operations take more time to complete than when storing diffs locally.

Using the home implementation, the fetching of the whole granule on an invalid access could be unnecessary as only a small part of the granule could have been modified. To reduce this overhead, the write vector technique has been proposed ([Hu99]). Using write vectors, granules are partitioned into blocks, the home maintaining a write vector about each granule cached by each worker. On reception of diffs, the home updates the write vectors according to the diffs received to mark sections of granules dirty. Beside sending the whole granule, the home only sends the dirty blocks to the requesting worker, reducing the amount of data sent and reducing network congestion. When the home sends data to a worker, it updates its write vector to mark the block valid for this worker. More than reducing the size of data sent to worker, a write vector implementation could be used to reduce the need for sending invalidations. The write vectors from the home could be used to avoid sending invalidations to workers where the granules have already been invalidated.

Allowing the buffering of the writes until the "release" and allowing concurrent writers to the same granule are the improvements of the eager release models over the sequential model. However, the potential transmission of data to workers not needing it, even using WI protocol, is an issue that could be addressed to increase the performance. The lazy release model solves most of this problem at the cost of needing more complex data structures for its implementation.

Lazy Release Model

The classic implementation of lazy release consistency from the TreadMarks ([KCZ92]) SDSM system uses the notion of twins and diffs presented in the previous section, and also uses *timestamp vector* ([Mat88]). A timestamp vector is an integer array V of size k, where k is the maximum number of workers. Each worker w keeps a vector of the events that have been performed locally: $V_w[w]$ corresponds to the current event on this worker, and $V_w[i]$ (where $i \neq w$) is the last event of worker i that has been performed locally.

In the case of lazy release consistency, events correspond to "acquire" and "release". On each "acquire" and "release", worker w increments $V_w[w]$. On an "acquire", w sends its V_w to the last releasing worker r. On reception of V_w, r compares V_w with the timestamp vector that was stored at the last release (or with its current V to avoid storing the vector on release). When $V_w[i]$ is less than $V_r[i]$, events initiated by worker i have been performed at r and not at w. These events must be performed at w prior to the "acquire" completion. After the comparison, r sends w an updated version of V_w with the corresponding invalidations or data updates (in respectively WI and WU protocols). After these operations, all the events that were performed at r prior to the "release" are performed at w.

An example of the timestamp vector applied to the lazy release model using three workers and four "acquires" is given in Figure 3.4. In this figure, one could notice that, even if worker c did not send message to worker a, worker c is aware of events that have occurred at a.

To send the corresponding invalidations or data updates, the worker r must either cache all the modifications or contact other workers caching them. Usually, r caches all the invalidations, but not the corresponding diffs. These diffs are stored locally in each worker that has computed them. This avoids local memory consumption for r, but results in a bad implementation of a WU protocol because many workers have to be contacted on an "acquire". This is why a WI protocol is almost always used with this algorithm. Moreover, using a WI protocol, the

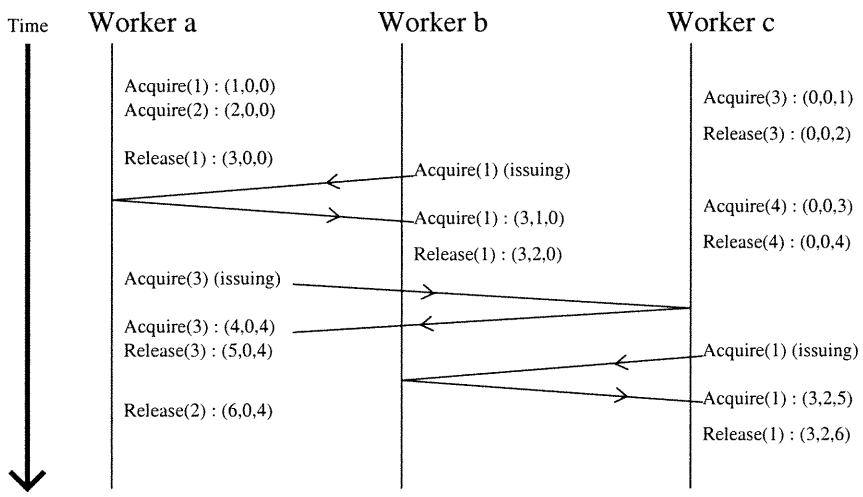

Figure 3.4: The Timestamp Vector Applied to the Lazy Release Model.

acquiring worker only requests data if memory is read, which is not necessarily the case in all applications, reducing even more network use.

The TreadMarks algorithm for the implementation of lazy release consistency allows concurrent writers on a granule because the modifications are propagated with diffs (see Figure 3.3). However, the quantity of memory needed to locally store the diffs is enormous. Garbage collection must be done regularly to avoid running out of memory. A solution to this problem, using diffs compression, has been proposed in [RDF+00], but is not described in detail here.

To reduce this memory consumption, a single-writer implementation of lazy release consistency has been proposed in [Kel96b]. According to benchmark results presented in that paper, the multiple-writers implementation only performs an average of 9% better than the single-writer implementation. The implementation of this single-writer lazy release consistency almost corresponds to the implementation of sequential consistency. A single worker, the owner, can have a writable copy of a granule. At an "acquire", a worker receives invalidations about granules. On a LOAD to an invalid granule, a worker fetches a copy of the granule cached at the owner. Prior to a STORE, a worker which is not the owner fetches the granule cached at the owner and becomes the owner. This algorithm allows LOADs in non-owner worker to perform concurrently with STOREs at the owner, and eliminates the diffs computation and storing overhead at the cost of transmitting more data on the network. This implementation is suited when false sharing on a granule is caused by a single writer and many readers since it does not prevent this

situation[2].

When false sharing is caused by many writers to the same granule, the single-writer implementation of lazy release consistency is not suitable. A home implementation of lazy release consistency benefitting of multiple-writers and low memory consumption is presented in [ZIL96]. As for the eager release model home implementation, diffs are sent to the home. On an "acquire", a worker still receives invalidations, but, on a LOAD to invalidated memory, it requests the granule to the home. As in eager release consistency, fetching of the whole granules could be unnecessary because only small parts of the granule could have been modified. The write vector technique, explained in the eager release consistency implementation, could be used to reduce unnecessary network traffic.

In both home and TreadMarks implementation, lazy diff creation could be implemented ([Kel95]). When using lazy diff creation, the diffs are not computed at the "release", but as late as possible. In the TreadMarks implementation, the diffs are only required when the updates are first requested by a worker, or when an invalidation is received on a granule where pending modifications exist. In the home implementation, the diffs are only required when the first worker fetches an invalidated granule at the home, or when an invalidation is received on a granule where pending modifications exist. To avoid a diff request from the home to the worker, diffs could be computed at the next "acquire" from a worker.

A performance comparison of the TreadMarks and home implementation of lazy release consistency was presented in [Cd+99]. The results are that four out of seven benchmarks perform within a 4% variance for both implementations, two benchmarks are faster with the TreadMarks implementation and one is faster with the home implementation. However, these results were published prior to the optimization of [Hu99] and other optimizations of the home-based implementation of lazy release consistency that are not presented in details here ([HST99, YLLM01]). A new comparison of the TreadMarks, the home and the single-writer implementation algorithms would be useful to actualize these results.

[2]The eager release memory consistency model could also be implemented using a similar algorithm to avoid the diff computation.

3.3.4 Implementing Synchronization Primitives

All the synchronization primitives can trivially be implemented using centralized protocols. For barriers, a "reach" sends a notification to the home, the home informing all workers when the barrier can be "lowered". For mutexes, an "unlock" sends a notification that the mutex is available to the home, a "lock" sends a request for the mutex to the home, this request being answered when the mutex becomes available. For condition variables, a "wait" causes a worker registration at the home in the waiting queue, a "signal" notifies the home that one waiting worker can be woken up, and a "broadcast" notifies the home to wake up all waiting workers.

The centralized implementations of the synchronization primitives are simple, but lack adaptability and scalability. For global barriers, this is not a major issue because it is always an expensive operation. In the case of mutexes and condition variables, if the home does not use the primitive, the implementation could be enhanced by migrating the responsibility of primitive management to workers that are using the primitive.

Distributed Mutex

Distributed mutexes can be implemented using the migration algorithm presented in [NTA96]. Each time a mutex is "locked", the locking worker becomes the owner of the mutex. This migration avoids network messages when a worker "locks" a mutex twice in a short time. To implement this migration, each worker only has a hint to which worker is the owner, this link being updated from worker i to worker j when a worker forwards a lock request from worker j to worker i. In addition to the ownership hint, each worker has a link to the next worker to give the mutex to when the mutex is to be "unlocked". The number of messages needed to "lock" a mutex using this algorithm is in $O(\log(n))$ for the average case (n being the number of workers).

Distributed Condition Variables

For distributed implementations of condition variables, [Mue00] presents an algorithm partially based on the mutex algorithm of [NTA96]. This algorithm is complex, thus not presented in this document. The reader is referred to this article for more information about distributed implementation of condition variables.

Distributed Global Barriers

Even if a centralized implementation of global barriers is acceptable, the home must send many messages on a "lower" and receives many messages of barrier "reach" when lots of workers participate in the computation. To reduce the number of messages sent by the home, a *tree barrier* ([Mue00]) could be used. In a tree barrier, the home sends lowering message to workers that, on reception, forward the message to other workers. This distributed barrier implementation allows the home to send less messages and reduces the total time spent by the home for the barrier management. Moreover, a tree-barrier can keep the home from receiving the "reach" messages from all workers. Once the first tree barrier has been done, a worker only sends "reach" message to its parent. When a worker has received "reach" messages from all its children and has itself reached the barrier, it then sends a "reach" message to its parent for all its children and itself, reducing the number of messages received by the home. The main advantage of the tree barrier algorithm is to reduce the number of messages sent and received by the home.

3.4 SDSM System Utilization

How to use a SDSM is system-specific. The exact information about the utilization of a SDSM system can be found in its documentation. However, some general concepts about program startup, task partitioning, shared memory and synchronization primitives are similar in many systems and are presented. These concepts, taken from the documentation of the SDSM systems Unify, CRL, CVM, Quarks and TreadMarks ([GYF95, JKW95, Kel96a, Kha95, TMK96] respectively), are presented in this section.

3.4.1 Program Startup

The usual way to start a SDSM program is by calling a special startup function. The arguments given to this function, usually taken from the program arguments, specify what work must be done. The meaning of the arguments is system-specific.

To execute a SDSM program, a list of nodes is usually given as argument to the program. The startup function will receive this list and remotely start a worker on each node. How this is exactly done is system-specific, but often use the `rsh` UNIX program.

3.4.2 Task Partitioning

The SDSM systems usually support a static task partitioning where the task assigned to each worker is based on the number of workers and the index of each worker. At run-time, each worker performs its task according to these two parameters.

Prior to the execution of the program, the number of workers is not known (except if the user always runs his program using the same number of workers). In the general case, the program must be written using worker indexes and the number of workers to partition the computation. In this case, if the user wants better performances, he can run his program using more workers on more nodes.

Using this partitioning method, the index assigned to each worker is not known prior to the program execution (except if the SDSM system assigns worker index according to the order of the nodes specified as arguments). This restriction means that a programmer cannot develop an application assuming that a specific node will perform a specific task. As the workers must be considered equivalent, the programmer must not assume the availability of specific resources only accessible on some nodes.

3.4.3 Shared Memory Utilization

The use of the shared memory in each SDSM system is different. However, the notion of region, not always named like this, can be found in almost all SDSM systems. The region is the memory allocation unit of a SDSM. When shared memory is allocated, a region is created. After creating a region, the workers can map the region in their local memory. Once mapped, the region becomes the local cache of the shared memory. Programmers can also unmap and destroy regions.

When executing the mapping, the region can be loaded at the same memory address as the other workers, or at a different address. When regions are not mapped at the same address, standard programming language pointers cannot be stored in shared memory because the mapping location is potentially different on each workers (a pointer is valid in a worker cache, but not on other caches). In this case, to address shared memory, the programmer must use custom pointers composed of a region id and an offset in that region.

When regions are mapped at the same address, pointers can be used in shared memory. However, memory must be reserved prior to program execution to allow the mapping of regions, limiting the amount of shared memory to the quantity of memory reserved at program startup. Moreover, using this algorithm, the amount of shared memory is limited by the node having the less memory. Even if trying to avoid this problem by a kernel implementation, shared memory is still limited to the size of the address space (4 gigabytes on 32-bit architectures like the Intel x86). When regions are not mapped at the same addresses, an application could theorically use more shared memory than the address space limit if the workers do not locally map more than this limit.

Some SDSM systems, such as CRL, CVM and Quarks ([JKW95, Kel96a, Kha95]), explicitly allow to create, map, unmap and destroy regions. Other SDSM systems, such as Unify and TreadMarks ([GYF95, TMK96]), allocate and map the regions at startup, providing a memory allocator that manages memory in these regions. These last two SDSM systems map their regions at the same address in memory, the others do not necessarily do so.

Support Several Memory Consistency Models

The support for many memory consistency models has been proposed to increase performance of SDSM applications ([Car94]). The advantage of supporting many consistency models is the flexibility provided to the programmers of choosing which model to apply to different sections of shared memory. In an application, different sections of memory are accessed using different access patterns. With many memory models at his disposition, a programmer is able to tune his application by using the appropriate model according to the access pattern. One of the most interesting advantage of providing multiple memory consistency models is the ability to choose between a write-update and a write-invalidate protocol.

3.4.4 Synchronization Primitives

The utilization of synchronization primitives follows the normal use of barriers, mutexes and condition variables as introduced in Section 2.2.1. Barriers allow waiting for workers at a point in the program. Mutexes allow enforcing mutual exclusion by allowing only one worker to be in a critical section. Condition variables allow one or many workers to wait for a condition to be

set by another worker, avoiding active waiting.

The behavior of synchronization primitives are SDSM system specific. To obtain more details about their use, such as the reentrancy behavior of mutexes, the documentation of the SDSM system used should be consulted.

When using the eager and lazy release models, the "acquire" and "release" accesses can be merged with synchronization primitives to simplify the programming model. A "reach" and "lower" on a barrier are considered as "release" and "acquire" respectively. A "lock" and "unlock" on a mutex are considered as "acquire" and "release" respectively. Condition variables are not merged with consistency operations because they are used in conjunction with mutexes.

3.5 Other SDSM Topics

This section is devoted to interesting topics in the SDSM field that are not directly addressed by this document but worth mentioning. These topics are not deeply covered, but some bibliographic references are given for the interested readers. The four topics addressed are SDSMs on multiprocessors, fault-tolerant SDSMs, the Linda model and SDSMs son heterogeneous NOWs.

3.5.1 SDSMs on Multiprocessors

The basic target architecture of SDSMs is NOWs. However, as multiprocessors with two processors are often less expensive than two uni-processor workstations due to shared components, a NOW could be composed of multiprocessor nodes. Spreading the cost of a high performance network and other expensive hardware on dual-processors multiprocessors could be a cost effective solution to increase the computing power of a NOW.

To take advantage of this special type of NOW, as many workers as the number of processors could be launched on each node. However, as these workers have been initially designed to run on distinct nodes, they communicate using message passing. They do not take advantage of the physical shared memory available on local nodes. Moreover, on a NOW of dual-processors multiprocessors, the amount of communication is doubled compared to uni-processor because both workers on the same node must be notified of consistency operations. Furthermore, workers on the same node could cache the same shared memory, increasing memory consumption.

To provide a better way to use multiprocessor NOWs, a SDSM system could spawn locally many workers for a single managing instance. This would reduce the communication needs between the workers as only one manager per node needs to be notified of consistency operations. Moreover, with a single manager for many workers, only one cache of the shared memory would be needed, reducing the memory consumption.

The SDSM system DSM-Thread provides the programmers with these facilities. More information on multi-threaded SDSMs can be found in [Mue97] and [Mue00].

3.5.2 Fault-Tolerant SDSM Systems

In classic SDSM systems, when a node fails, the complete application crashes because data needed by the application could be lost due to the failure. As the probability of the failure of a node is low, this behavior is acceptable in the majority of applications.

However, since the probability of a single node failure grows with the number of node, the probability of failure can become unacceptable on a large NOW. Moreover, in applications taking a long time to execute, a crash is very annoying near the end of the computation. Furthermore, in some critical applications, such failures are unacceptable. In these cases, fault tolerance strategies must be implemented to avoid an application crash in case of a node failure.

Some fault-tolerant SDSM systems have been developed to avoid the crash of an application after a node failure. A strategy used to provide fault tolerance is *backward error recovery* (BER) also known as *checkpointing*. In checkpointing SDSM systems, regular backups of the state of the computation are performed ([KCG+95, MKB97]). In the event of a node failure, the computation is restarted from the last backup.

The checkpointing strategy can be suitable for scientific applications where the lost of a small amount of computation can be afforded. However, in applications needing more real-time responsiveness, this strategy cannot be used. The boundary-restricted protocol ([FMST00]) addresses this problem. This protocol is implemented in the Oasis+ SDSM system ([WLF01]).

Fault-tolerant SDSM systems are still an open field of research. More results are needed to achieve acceptable SDSM fault tolerance at relatively low cost in a wide range of applications.

3.5.3 The Linda Model

Other shared memory models exist. Among them, the *Linda* model ([ACG86, CG90]) provides the programmer with a tuple space where tuples can be inserted and removed according to field matching. Once added to the tuple space by a computing instance, a tuple can be removed by any instances of the computation. Once removed, a tuple is only accessible by the instance which has removed it from the tuple space.

The task partitioning model of a tuple space is the bag of tasks. Tasks are inserted and removed from the tuple space to represent the computation. The bag of tasks partitioning discussed later in this document has been elaborated from the Linda model.

Contrary to other SDSMs, Linda does not provide linear shared memory abstraction. The only shared memory available is the tuple space. All shared data must be converted to tuples before being used. A problem with this approach is the implementation of constant data. To allow other workers to consult the data, a worker must always insert back the tuple after having removed it and read its content.

A commercial implementation of Linda is available. More details are available on the web site of the company Scientific Computing Associates Inc. ([lin]).

3.5.4 SDSMs for Heterogeneous NOWs

The focus of this work is on homogeneous NOWs built from nodes of the same computer architecture (but not necessarily of the same speed). Some SDSM systems, such as InterWeave ([PCD+00]), target heterogeneous NOWs built using different computer architectures. Having a distributed shared memory on heterogeneous NOWs allows to maintain a distributed shared state on applications running on heterogeneous NOWS, such as the Internet. One of the problem that must be addressed by such systems is the different representation of data on different architectures.

3.6 SDSM in Context

SDSM systems are a relatively new tool for the development of parallel applications on NOWs. Their niche is between multiprocessing and pure message passing. SDSM systems will not replace multiprocessors because the performance of SDSM applications cannot match their performance in applications needing lots of communication. Moreover, some applications, traditionally developed using message passing, do not perform well on SDSM systems because of the overhead introduced by the management of shared memory.

The greatest advantage of SDSMs is the reduction of code needed to develop a parallel application, thus producing a more elegant solution than message passing. This clear superiority of SDSM, used adequately on problems needing little communication or where the speed is not the prime objective, is the niche of SDSM systems.

When the cost of development and maintenance of a message passing application will be greater than the performance degradation engendered by the use of a SDSM system, programmers could switch from the message passing model to SDSM systems. However, this assumes the availability of SDSM systems and the awareness, by the parallel programming community, of their advantages. This still needs to be done by presenting results about the success of SDSM systems solving known, impressive and relevant problems. The SDSM killer application has not been found yet.

3.7 Summary

This chapter was an introduction to the field of DSMs and more particularly SDSMs. The definition of a DSM, the memory consistency models, the implementation of SDSM systems, their use, miscellaneous SDSM topics, and the niche of SDSMs have been discussed.

We have seen that a DSM is an emulation of shared memory on NOWs. This shared memory does not physically exist but is simulated using a cache on each node. Because many caches are maintained, coherence among these caches must be enforced. How the caches are kept coherent is defined by a memory consistency model.

Four memory consistency models have been presented: the sequential, processor, eager re-

lease and lazy release models. These models define different restrictions on the ordering of memory accesses. Some models are more relaxed than others, enforcing less restrictions on memory access ordering. Strict models are usually easier to use, but result in inefficient implementations due to their restrictions on access ordering.

In the section about the implementation of SDSM systems, the centralized and distributed protocols, and the write-update and write-invalidate memory implementations have been presented with their advantages and drawbacks. Algorithms about the implementation of the sequential, eager release and lazy release memory models were also presented. We have seen that implementing a relaxed model such as the lazy release is more difficult than a strict model like the sequential or eager release models, but the benefits obtained from these implementations are worth their complexity. Centralized and distributed algorithm for the implementation of synchronization primitives have also been mentioned.

In the section about the utilization of SDSM systems, the startup of a SDSM program (by giving a list of nodes as the program arguments) has been presented along with the task partitioning method based on the number of workers and the index of each workers. The shared memory region concept and its related operations (creation, mapping, unmapping and destruction) have also been presented along with the utilization of synchronization primitives. Moreover, the reader was introduced to SDSM systems supporting many memory consistency models, allowing the programmer to choose the appropriate model according to the sharing pattern of his application.

The miscellaneous SDSM topics covered included SDSMs on multiprocessors and fault-tolerant SDSM systems. We have seen that some optimizations could be done in SDSM systems to increase their performance on multiprocessor nodes and that fault tolerance is an issue that SDSM systems will have to address before being suitable for long running and critical applications. The Linda model and SDSM systems on heterogeneous NOWs were also mentioned.

Finally, we have explained why SDSM systems will neither replace multiprocessors nor message passing. Their advantage over message passing is ease of use, and, over multiprocessors, low cost. However, as they introduce overhead, they cannot beat a well tuned message passing application, and, as they are limited by the communication interface of NOWs, they cannot reach the high performance of multiprocessors. However, they can achieve relatively good performances in some applications considering the easy and quick development of SDSM applications.

Chapter 4

YADL Description

YADL is designed to address most needs of SDSM programmers, usability being the first one. It features many key concepts of SDSMs including support for multiple memory consistency models. Moreover, it provides the programmer with the ability to choose between two task partitioning methods.

Section 4.1 describes the specifications of the system. The implementation details are given in Section 4.2, followed, in Section 4.3, by the utilization of the system.

4.1 Specifications

This section describes the specifications of YADL. These specifications result from the choices that were made prior to the implementation of the system. As the reader will notice, this section does not present implementation details. Those details are presented in Section 4.2.

This section begins by elaborating the requirements and design choices. The API of the system, designed according to the requirements, is then presented and explained.

4.1.1 Requirements and Design Choices

Prior to the development of YADL, the following requirements were identified:

1. Accommodate novice and expert programmers;
2. Support several memory consistency models;
3. Allow easy extension;
4. Supply efficient computation partitioning methods;
5. Allow the use of as much shared memory as possible.

Each of these requirements is presented in this section. The design choices motivated by these requirements are presented along them.

Accommodate Novice and Expert Programmers

Some programmers are novices and others are experts. The novices want most of the work to be done by the SDSM system. YADL should provide them with an interface hiding most of the SDSM actions, such as detecting memory accesses with the VMH and combining memory consistency operations with synchronization primitives (see Sections 3.2.4, 3.2.5 and 3.3.1). However, relying only on a high-level interface forbids some optimizations that could be done by experts. Thus, a low-level interface should also be provided.

To accommodate novice and expert programmers, YADL must provide memory models using annotations and using the VMH. Moreover, consistency operations must be available directly or via synchronization primitives. To fulfill these requirements, an interface to the shared memory must allow the annotation of beginning and end of memory accesses. Moreover, multiple region and synchronization primitive types must be available and must be able to be used simultaneously in a program. This will allow a programmer to use both annotated and VMH shared memory in his program and will allow him to both directly and indirectly control the consistency operations performed on shared memory.

Support Several Memory Consistency Models

Supporting several memory consistency models has been proposed to allow tuning an application according to the data sharing pattern (see Section 3.4.3). Providing such a feature would allow YADL to be more flexible and would increase its efficiency.

To support several memory consistency models and to allow easy extension, the region cre-

ation and shared memory access annotation processes must be as generic as possible to allow the implementation of any memory model. The region creation process must allow supplying a memory type to specify which memory consistency model should be applied to the region. Moreover, the region creation process must also allow the programmer to supply type-specific data. This data could be tuning parameters, such as Δ for sequential model or the diff unit of the release models (see Section 3.3.3). Finally, the granularity of the region must also be specified at the creation process to allow the programmer to choose the granularity according to the characteristics of his application (see the introduction of Section 3.3.3).

Allow Easy Extension

Equally, if not more important, is the ability to easily add new consistency models and new implementations of existing models to the system. This would serve research needs as it would allow the comparison of different models and implementations. Moreover, this would be a great benefit to programmers as, when in need for a new or custom memory consistency model, they could add it to the system. This would allow YADL to be used in many situations where the development of a communication protocol is normally needed.

To allow easy extensions, the implementation of the system must be as modular as possible. The addition of a memory consistency model or synchronization primitive should only be a matter of adding a new module to the system.

Supply Efficient Partitioning Methods

Partitioning the computation is essential to the parallelization of an application (see Section 2.1). YADL should provide the programmers with the ability to easily and efficiently partition their computation.

As exposed in Section 3.4.2, SDSM systems usually support a static computation partitioning method based on the number of workers and the index of each worker. This simple and intuitive partitioning method is only suited for problems where equal size task partitioning can be achieved. If this cannot be done, load balancing cannot be achieved and some workers finish their tasks sooner and stay idle. Moreover, for the same reason, this partitioning method is also not suited when different processor speeds are used. Furthermore, it does not allow new workers

to join the computation because no task can be dynamically given to them.

Because of its simplicity and popularity, the *static partitioning* (this is how will be called in the rest of this document the partitioning method based on the number of workers and the indexes of workers) should be supported by YADL. However, a dynamic partitioning approach should also be provided for the cases where the static partitioning delivers bad performance. The *bag of tasks* (BOT) approach ([CG90]) is suitable for this purpose.

When using the BOT partitioning, a program is partitioned in tasks that are stored in a management service. Each worker requests and executes tasks. An extended task management service could support task dependencies where a task β dependent on task α will not be assigned to a worker before the completion of α. Moreover, an extended BOT service could also support task substitution. Substituting a task τ consists in replacing it by sub-tasks which must all be completed before the assignation of a task depending on τ. This substitution allows divide-and-conquer algorithms to be implemented easily.

The BOT partitioning allows new workers to join the computation because tasks can be dynamically assigned to them. It also allows a worker to leave the computation when it is done with its current task. The BOT partitioning also reduces the problems associated with different processor speeds and unequal task partitioning: the faster workers simply execute more tasks.

Allow the Use of As Much Shared Memory As Possible

Allowing the use of as much shared memory as possible is useful to avoid being limited by a node with less physical memory. It is also useful to avoid the 4 gigabyte limitation of the 32-bit architectures (see Section 3.4.3).

To allow the use of as much shared memory as possible, explicit shared memory mapping has been chosen. As explained in Section 3.4.3, one of the operations on shared memory is the mapping. The mapping can be done implicitly or explicitly. Explicit memory mapping was chosen to provide the programmer with the ability to unmap unused memory. This does not forbid the mapping of regions at the same address on different workers as this feature can be implemented by a specific region type.

4.1.2 Application Programming Interface

This section presents the application programming interface (API) of YADL in pseudo-C language. This API has been designed to fulfill the requirements without regards to implementation concern. The proposed flexible API is the stronger aspect of YADL as it allows to address most current and future requirements.

All the functions of the API return an integer value. This value is zero on success (the constant `DSM_OK`), or a non-zero error code on failure.

System Management

The system management functions are listed in Figure 4.1.

```
int DSM_Start(int argc, char *argv[]);
int DSM_Stop();

int DSM_GetWorkerIndex();
int DSM_GetNumberOfWorkers();

int  DSM_GetTask(DSMTask_t *pTask);
int  DSM_CommitTask(DSMTask_t *pTask);
int  DSM_ReplaceTask(DSMTask_t *pTask, DSMAddTask_t *pAddTasks, size_t NbTasks);
void DSM_InitAddTask(DSMAddTask_t *pTask, int nType, char *TaskData, size_t nTaskData, int nDep);
```

Figure 4.1: The System Management API.

`DSM_Start` and `DSM_Stop` are used to initialize and halt the SDSM system. The parameters given to `DSM_Start` specify initialization informations.

`DSM_GetWorkerIndex` and `DSM_GetNumberOfWorkers` return, when using the static partitioning, the index of the current worker and the number of workers respectively. When using the BOT partitioning, they return a unique identifier for the current worker and the last known number of workers. It is important to notice that the unique identifier returned by `DSM_GetWorkerIndex` could be greater than the last known number of workers returned by `DSM_GetNumberOfWorkers` if workers leave the computation.

`DSM_GetTask`, `DSM_CommitTask` and `DSM_ReplaceTask` are used to request a task, signal the completion of a task and substitute a task respectively. `DSM_InitAddTask` is supplied for ease of programming: it initializes the task `pTask` with the arguments given in parameters. These functions should only be called when using the BOT partitioning.

Operations Commons to All Resources

The functions common to all resources are listed in Figure 4.2. The YADL resources are the regions, the barriers, the mutexes, the condition variables and the semaphores.

```
int DSM_Create<Resource>(DSM<Resource>Data_t *pData);
int DSM_Info<Resource>(int n, DSM<Resource>Data_t *pData);
int DSM_Destroy<Resource>(int n);
```

Figure 4.2: The Operations Common to All Resources.

The function `DSM_Create*` is used to create a resource. The parameter `pData` includes the resource number, type, and type-specific data. The resource number is used to globally identify the resource, the type corresponds to a resource type, and the type-specific data is an union incorporating all possible informations that could be required at the resource creation. The function `DSM_Info*` returns informations about resource number `n` in the parameter `pData`. The function `DSM_Destroy*` destroys the resource `n`.

In the special case of the regions, `pData` also includes fields about the size and granularity of the region. The size is the requested shared memory size and the granularity is the unit at which the region is managed.

Operations on Regions

The region management functions are listed in Figure 4.3.

```
int DSM_CreateRegion(DSMRegionData_t *pRegionData);
int DSM_InfoRegion(int n, DSMRegionData_t *pRegionData);
int DSM_DestroyRegion(int n);

int DSM_MapRegion(int nRegion, void **pp);
int DSM_UnMapRegion(int nRegion);

int DSM_BeginAccess(int nRegion, size_t Off, size_t Len, DSMAccess_t Access, void **pp);
int DSM_EndAccess(int nRegion, size_t Off, size_t Len, DSMAccess_t Access);
int DSM_PrefetchAccess(int nRegion, size_t Off, size_t Len, DSMAccess_t Access);
```

Figure 4.3: The Region Management API.

The function `DSM_MapRegion` allocates local memory for region `nRegion`, and modifies the parameter `pp` to indicate its mapping site. When a region is not mapped, no memory is locally reserved. This allows the total amount of memory used to exceed the address space of the processor (if the amount of memory mapped at any time by any worker fits in the address

space). The function DSM_UnMapRegion frees the memory allocated for region nRegion.

The functions DSM_BeginAccess, DSM_EndAccess and DSM_PrefetchAccess are used to annotate memory accesses. They are provided for the implementation of the low-level interface discussed in Section 4.1.1. The parameter nRegion is the region number to which the access is performed, the parameter Off is the offset where the access is performed, the parameter Len is the length of the memory range where the access is performed, the parameter Access is the access type that is performed (DSM_REGION_ACCESS_READ, DSM_REGION_ACCESS_WRITE or DSM_REGION_ACCESS_READ_WRITE), and the parameter pp is modified to point to the memory that is accessed. The use of these functions is specific to the region type.

Operations on Barriers

The barrier management functions are listed in Figure 4.4.

```
int DSM_CreateBarrier(DSMBarrierData_t *pBarrierData);
int DSM_InfoBarrier(int n, DSMBarrierData_t *pBarrierData);
int DSM_DestroyBarrier(int n);

int DSM_WaitInit(int n);
int DSM_ReachBarrier(int nBarrier);
```

Figure 4.4: The Barrier Management API.

The function DSM_WaitInit implements a global barrier (see Section 2.2.1). This barrier must be provided to allow the synchronization of workers at initialization when using the static partitioning. If this barrier not was provided, no synchronization would be possible at program startup because the synchronization primitives are not yet created. The parameter n, a number identifying the barrier, is only used for clarity of code and debugging purposes. This function should only be called when using the static partitioning.

The function DSM_ReachBarrier performs a "reach" operation on the barrier nBarrier. This function returns only when the correct number of workers have reached the barrier.

Operations on Mutexes

The mutex management functions are listed in Figure 4.5.

The function DSM_LockMutex performs a "lock" on the mutex nMutex, and the function

```
int DSM_CreateMutex(DSMMutexData_t *pMutexData);
int DSM_InfoMutex(int n, DSMMutexData_t *pMutexData);
int DSM_DestroyMutex(int n);

int DSM_LockMutex(int nMutex, int nFlags);
int DSM_UnlockMutex(int nMutex);
```

Figure 4.5: The Mutex Management API.

`DSM_UnlockMutex`, an "unlock". The parameter `nFlags` given to `DSM_LockMutex` is specific to the mutex type and could specify a non-blocking mutex. The reentrancy behavior of mutexes is mutex type-specific.

Operations on Condition Variables

The condition variable management functions are listed in Figure 4.6.

```
int DSM_CreateCondVar(DSMCondVarData_t *pCondVarData);
int DSM_InfoCondVar(int n, DSMCondVarData_t *pCondVarData);
int DSM_DestroyCondVar(int n);

int DSM_WaitCondVar(int nCondVar, int nMutex);
int DSM_BroadcastCondVar(int nCondVar);
int DSM_SignalCondVar(int nCondVar);
```

Figure 4.6: The Condition Variable Management API.

The function `DSM_WaitCondVar` performs a "wait" on the condition variable `nCondVar` unlocking the mutex `nMutex` at the same time of the "wait", the function `DSM_BroadCastCondVar`, a "broadcast", and the function `DSM_SignalCondVar`, a "signal".

Operations on Semaphores

The functions related to semaphore management are listed in Figure 4.7.

```
int DSM_CreateSemaphore(DSMSemaphoreData_t *pSemaphoreData);
int DSM_InfoSemaphore(int n, DSMSemaphoreData_t *pSemaphoreData);
int DSM_DestroySemaphore(int n);

int DSM_WaitSemaphore(int nSemaphore, int nFlags);
int DSM_SignalSemaphore(int nSemaphore);
```

Figure 4.7: The Semaphore Management API.

The function `DSM_WaitSemaphore` performs a "wait" on the semaphore `nSemaphore`, and the function `DSM_SignalSemaphore`, a "signal". The parameter `nFlags` given to `DSM_WaitSemaphore`

is semaphore type-specific and could specify a non-blocking semaphore.

Consistency Operations

The consistency management function is listed in Figure 4.8.

```
int DSM_Release(void);
```

Figure 4.8: The Consistency Management API.

The functions `DSM_Release` perform a "release". It is used for the implementation of the eager release consistency model (see the implementation of eager release consistency in Section 3.3.3).

4.2 Implementation

In Section 4.1.1, the specifications have been presented without regard to the implementation details. The current section explains how YADL is implemented. As the system is only partially implemented, the first section introduces an overview of the implementation describing what has been implemented. Following the overview, the architecture of the system is presented, explaining the general flow of data and messages. The different memory consistency models currently available are then presented, and their implementation algorithm are also explained. Following the implementation of memory consistency models, the implementation of synchronization primitives is presented. Finally, the implementation of the bag of tasks is explained.

4.2.1 Overview of the Implementation

YADL is functional, but some features are not implemented yet. Partially or completely implemented features are the static and the BOT partitioning, the region and the mutex management functions, the "init" global barrier and the function `DSM_Release`. The features that are not implemented are the general barriers, the semaphores and the condition variables.

The static partitioning has been completely implemented. The functions `DSM_GetWorkerIndex` and `DSM_GetNumberOfWorkers` have also been implemented according to the static partitioning specifications. Moreover, the function `DSM_WaitInt` provides a functioning global barrier.

As barriers are not implemented, the "init" is actually the only way to perform a global barrier in the system.

The BOT partitioning is partially implemented. The functions `DSM_GetTask`, `DSM_CommitTask` and `DSM_ReplaceTask` are implemented. `DSM_GetWorkerIndex` and `DSM_GetNumberOfWorkers` are also implemented according to the BOT partitioning specifications. Very little work would be needed to implement the addition of workers in the middle of the computation. However, worker departure from the computation is not implemented, and lot of work would be needed to implement it. The complexity of worker removal implementation is due to the modification of the memory consistency algorithm required to support this feature.

The region management interface is almost completed. The functions `DSM_CreateRegion`, `DSM_MapRegion`, `DSM_UnMapRegion`, `DSM_BeginAccess`, `DSM_EndAccess` and `DSM_PrefetchAccess` are implemented. Three region types are implemented, their implementations are described in Section 4.2.3. As two of these region types implements eager release consistency, the function `DSM_Release` is provided to perform a "release". However, the function `DSM_InfoRegion` and `DSM_DestroyRegion` are not implemented, but, as they are not essential to the development of parallel applications, this is not a limitation.

The mutex management interface is almost completed. `DSM_CreateMutex`, `DSM_LockMutex` and `DSM_UnlockMutex` are implemented. A single mutex type has been implemented, its implementation is described in Section 4.2.4. However, the functions `DSM_InfoMutex` and `DSM_DestroyMutex` are not implemented, but again, as they are not essential to the development of parallel applications, this is not a limitation.

Even if only a partial implementation has been done, complete parallel programs can be developed with YADL using both the static and the BOT partitioning. The benchmarks presented in Chapter 5 prove this.

4.2.2 Architecture

The architecture of YADL is composed of three different entities: the server, the clients and the network. In the course of execution of a YADL program, one server and many clients communicate via the network as shown in Figure 4.9. The next sections present each of these components.

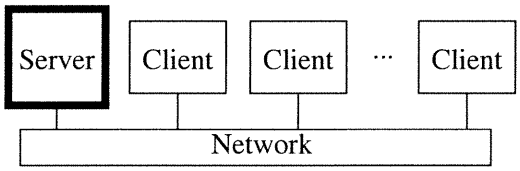

Figure 4.9: The Architecture of YADL.

The Network Layer

The network layer, called UDN (Universal Datagram Network), provides reliable packet-oriented communication. The packet-oriented nature of the network layer implies that the unit of communication is a message including a size and the corresponding data. The reliability characteristic of the packet transmission guarantees that, once emitted, the packet will be received only once at its destination and that the packet payload will not be corrupted. However, the order of packet reception is not defined: two packets sent to the same destination by the same sender are not necessarily received in the same order as they were sent.

The network layer has been designed to be easily implemented over UDP/IP ([Ste94]). When implementing UDN over UDP, only reliability needs to be added to the protocol. This could be achieved by packet numbering, retransmission and acknowledgment as for HDLC ([Hal96]). Moreover, to avoid the overhead of the IP protocol, UDN could be implemented directly over Ethernet or any other low overhead link layer protocol.

UDN has not been implemented over UDP yet. The current implementation of UDN is over TCP/IP. This implementation is straightforward as TCP already provides reliable delivery services. The packet transmission over TCP is implemented by inserting, in the TCP stream, the size of the packet followed by the payload. When TCP data is read, the size is read first to know how much data follows. After reading the data, the next read on the TCP stream is aligned to a packet size. This alignment is important to keep the integrity of the packet communication. To reduce the number of system calls, when reading packet data, enough buffer space is allocated to also read the next packet size. This reduces the number of system calls, when another packet is available in the TCP stream, by reading both the payload and the next packet size using a single system call.

UDN provides blocking and non-blocking send and receive functions. It also provides send and receive UNIX "select" wrappers to avoid active waiting for sending and receiving packets.

Its implementation should be thread-safe in the future, but is not in the current implementation. When many threads are sending packets through UDN, mutual exclusion must be enforced to avoid more than one thread to be in the UDN layer at the same time. The current implementation is not multi-threaded, but future releases of UDN over UDP and Ethernet will probably be multi-threaded.

The Server

The server purpose is to be the first point of contact for workers when they are started and when they need information about regions or synchronization primitives. When a worker is created, it sends a *join request* to the server. This request is normally answered by a *join notification*, informing the worker that it has been placed in the worker list. When using static partitioning, once all the workers have joined the computation, a *start computation notification* is sent by the server to all workers. When using the BOT partitioning, the server sends a *start computation notification* immediately after the *join notification*. This allows, when using the BOT partitioning, workers to start their computation as soon as they join the computation. The *start computation notification* includes the index of the worker and the number of workers participating in the computation. These values are accessible using the functions `DSM_GetWorkerIndex` and `DSM_GetNumberOfWorkers` respectively.

While performing their computation, the workers need to create and access shared memory and synchronization primitives. When creating these resources, a worker registers itself at the server as the creator of the resource. When a worker needs to access an unknown resource, it sends a *data request* to the server. The server forwards the request to the creator of the resource that answers it.

The two other responsibilities of the server in the current implementation are to manage the "init" barrier and to manage the BOT service. These responsibilities could be implemented using distributed algorithms. However, the centralized implementation in the server has been chosen for ease of implementation.

The tasks to be carried by the server, apart from the BOT management, are tasks that occur infrequently: adding or removing workers is occasionally done, resource data request forwarding occurs only the first time a worker needs informations about a resource, and the "init" barrier should only be used at the beginning of a static partitioning computation. Except for the BOT,

the centralized server implementation should not reduce the performance of YADL nor cause a bottleneck. Only the BOT could cause problems which could be solved in the future by implementing a distributed BOT management.

The server is implemented as a simple message loop. The server waits in a loop for requests. When a request is received, it is treated according to its type. These requests could be a resource creation notification, a resource data request, a task request, a task commit, a task replacement, an "init" reach, or a worker join request. As these requests are infrequent, the server can be run on the same node as one of the client to avoid dedicating a node of the NOW to the server execution.

The Client

The architecture of the client is more complicated than the architecture of the server because the client is an entity composed of two threads: the manager and the worker. The worker thread is running the SDSM application, and the manager thread, the SDSM system itself. Both threads communicate with the server and other clients, but only the manager receives and answers requests from other clients, as shown in Figure 4.10. As the current implementation of UDN is not thread-safe, a mutex must be locked before the manager or the worker send data via UDN to guarantee the integrity of the UDN layer.

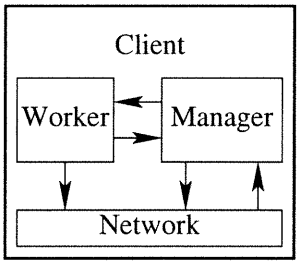

Figure 4.10: The Architecture of YADL's Client.

This architecture avoids merging the manager and the worker in the same thread. Multi-threaded clients have been avoided in the past, probably due to the overhead brought by multi-threading. In single-threaded SDSM systems, the `SIGIO` signal is used to interrupt the SDSM application to allow answering requests ([KCDZ94, Kha]). The multi-threaded approach was chosen for its simpler implementation and to eventually support multiprocessor NOWs by having

more than one worker with a single manager (see Section 3.5.1). Moreover, by not using the `SIGIO` signal to implement the system, a programmer can use this signal in his application to manage asynchronous messages, which provides him with more freedom.

The Manager

The manager is mainly responsible for answering requests from other clients and to receive the answers from requests issued by itself or by the worker thread. The manager is in a "select" loop, waiting for data the be available, or waiting for the availability of network buffers.

Little data is sent directly when handling a request because the manager must not block on a send, resulting in the impossibility of answering other requests and perhaps resulting in a network deadlock. To allow to serve these requests later, the manager has a pending send queue of the requests that have not been completely answered yet. When network buffers are available, the pending queue is partially emptied, fulfilling pending requests.

When a message is received, it is dispatched to the correct module or directly handled according to the message type. There exist modules for the region, barrier, mutex, condition variable, semaphore and tasks management. Each of these modules can be composed of sub-modules for different region types, barrier types, and so on.

By virtue of its modular design, the manager can be extended to support new memory consistency models and synchronization primitives. Implementing a new model or primitive consists only in adding a new module and modifying the message redirection interface to redirect messages associated with the new model or primitive to the right module.

The Worker

The SDSM application is running in the worker thread. Most of the time, this thread is executing application code. When a SDSM API call is performed, a request is usually sent to another client via UDN. If an answer is required before returning from the API call, the worker thread waits on a condition variable for the manager thread to notify that the request has been answered. If no answer is needed or if the answer will only be needed later (as for prefetching), the function returns and the manager thread handles the answer when it is received.

Another way for the worker to invoke indirectly the SDSM API is to perform an invalid

access to shared memory trapped by the VMH. In this case, the page-fault handling function, registered at initialization, sends a request to other clients if required. Again, it is the manager that notifies the completion of the request by doing a "signal" on a condition variable.

If lot of data must be sent by a worker, the worker delegates the delivery of this data to the manager and waits on a condition variable for the end of the process. The manager is more suited than the worker to send lot of data because it manages the pending send queue.

4.2.3 Memory Consistency Models

This section describes the implementation the three memory consistency models currently available in YADL. These models are write-once, eager release write-update and eager release write-invalidate. Other models, such as the sequential, other versions of the eager release, and the lazy release models, could be implemented in the future.

Write-Once Model

The write-once model is intended to be used for read-only shared memory. The "write-once" name was chosen instead of "read-only" because the name "read-only" does not indicate that this memory can only be written once, then only read.

In the implementation of the write-once model, the worker creating the region is the only worker that can initialize it. Once initialized, all the workers can read the memory.

Annotated and VMH versions of this model are provided. Both implementations use a centralized protocol where the home is the creator of the region. The general algorithm for both implementations is:

- Once the region is created, the home must initialize it;
- When a worker performs a LOAD to uncached data, the data is requested from the home;
- When a worker performs a LOAD to cached data, no consistency operations are required because the memory cannot be modified (the local cache is always valid).

When using the annotated version of write-once model, of which the type in the system is `DSM_REGION_WRITE_ONCE_ANNOTATED_HOME` or equivalently `DSM_REGION_WOAH`, the creator of the

region must use the functions DSM_BeginAccess and DSM_EndAccess to indicate the system of the beginning and end of the initialization of the memory respectively. Because all the memory must be initialized (no partial initialization being allowed), the offset (the Off parameter) and length (the Len parameter) must be 0 and the size of the region respectively. Moreover, when initializing the region, the home must use DSM_REGION_ACCESS_READ_WRITE as the access type (the Access parameter). Once the region has been initialized, any worker can use the functions DSM_BeginAccess and DSM_EndAccess to read data from the region. When reading write-once memory, a worker must use the access type DSM_REGION_ACCESS_READ. Prefetching can also be done to attempt reducing the idle time waiting for data. To prefetch data, the function DSM_PrefetchAccess is used with access type DSM_REGION_ACCESS_READ. Contrary to the function DSM_BeginAccess, the function DSM_PrefetchAccess does not block waiting for data from the home. The prefetching of data is performed while the worker executes the application code. When a prefetch has been performed, a subsequent call to DSM_BeginAccess will hopefully not block.

When using the VMH version of write-once model, of which the type in the system is DSM_REGION_WRITE_ONCE_VMH_HOME or equivalently DSM_REGION_WOVH, the annotation functions do not need to be used (and must not be used). The creator (the home) is still required to initialize the region. The end of the initialization process occurs when the first worker requests data from the region. If a request occurs before the initialization of memory, the system behavior is unpredictable.

For both implementations of write-once model (annotated and VMH), request forwarding has been implemented to optimize data request answering. Without this optimization, the home is responsible to answer all the data requests from every worker. As request answering is limited by the available network bandwidth, the home could need to queue many pending requests, resulting in delay for request answering. The request forwarding optimization consists in the delegation of request answering to other workers already caching the data. This optimization leads to great improvements in the answering process and to greater scalability. Figure 4.11 shows that without the optimization, the delays required to send 128 megabytes of data is $O(n)$ (where n is the number of nodes). The complexity becomes $O(log(n))$ when the optimization is used (the experience methodology is described in Section 5.2).

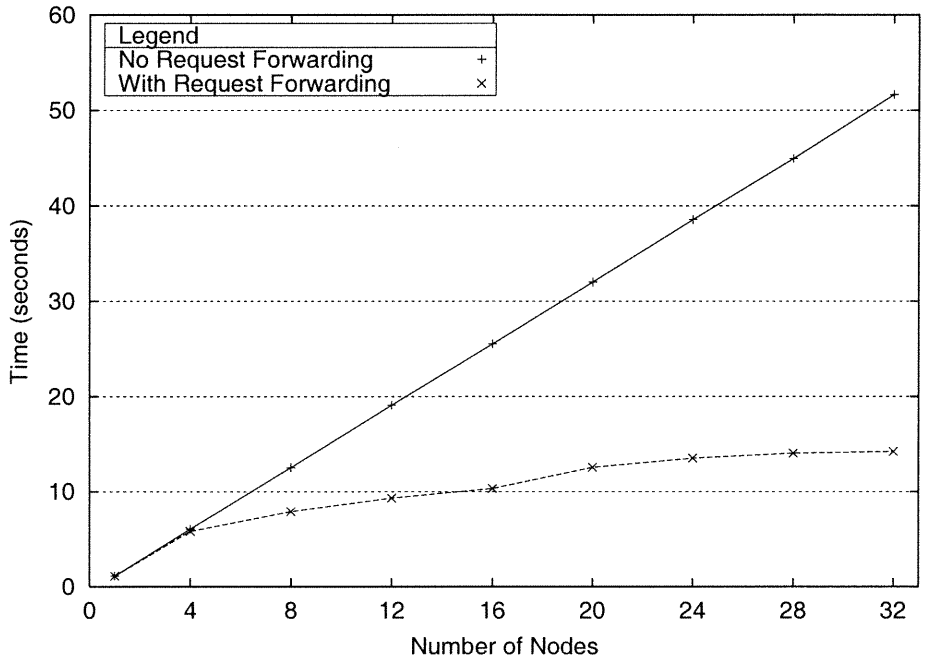

Figure 4.11: 128 Megabytes Sent With and Without Request Forwarding.

Eager Release Write-Update Model

The eager release write-update model is intended to be used for shared memory that is more often read than modified, or that is read by most of the workers before being modified. To implement this model, a centralized protocol where the creator of the region is the home is used. The implementation is based on the conservative algorithm for eager release consistency presented in Section 3.3.3. The algorithm is:

- When a worker other than the home maps the region, a copy of the entire region is sent to the mapping worker;
- When the first STORE on a granule following a "release" is performed, a twin of the granule modified is created and all subsequent STOREs perform normally;
- When a "release" is performed (by calling the function `DSM_Release`), the diffs are computed and sent to all workers caching the region.

Note: a LOAD is always a local operation because the write-update protocol is used.

To know which workers are caching the region, a releasing worker w sends a release initiation request to the home. On reception of such request, the home sends w the list of the n workers currently caching the region and updates its local data structures to record that w is doing a "release" and that it has been indicated that n workers cache the region (the purpose of this information will be explained shortly). The release initiation request must be sent on each "release" because the list of workers caching the region can change.

On reception of the caching workers list, w sends each worker the diffs. Then, w waits for the diffs reception acknowledgments from all the workers in the list. When all the acknowledgments have been received, w sends a release completion request to the home including the number of caching workers it is aware of. Finally, w waits for a release completion acknowledgment from the home.

The release completion request and its acknowledgment are required to guarantee that all the caching workers have received the diffs. As workers can map the region at any time, the number of caching workers can change while a "release" is in progress. When a new worker maps the region, the home notifies all the releasing workers to send diffs to this new caching worker, and updates its local data structures accordingly. The home only acknowledges the release completion request when the number of caching workers transmitted by the releasing worker corresponds to the number stored in the home local data structures.

It would be possible to avoid the release completion request and acknowledgment by sending the diffs to the home first, and, when its acknowledgment is received, sending the diffs to the caching workers identified in the diffs acknowledgment from the home. This avoids the release completion acknowledgment, but prevents sending diffs to other workers at the same time they are sent to the home. This algorithm has not been implemented, but could be to compare the performance of the two methods.

A client can receive diffs in three different situations: while the worker is mapping a region, while the region is locally mapped, and when the region has been unmapped. When receiving diffs about an unmapped region, the diffs are simply discarded. When receiving diffs while in the process of mapping a region, the diffs are stored to be applied when the region is completely mapped (when all the data is received from the home). When receiving diffs while the region is locally mapped, the diffs are applied to the shared memory. The application of diffs is dependent on the access detection method and is discussed shortly.

When creating a region implementing the eager release WU model, a type-specific parameter, the diff unit, must be supplied. This parameter specifies the granularity of diff management. Theorically, this parameter can be any power of two, but the currently accepted values are 1, 2, 4 and 8. If a small diff unit is specified, small parts of the region can by concurrently modified by different workers without the diffs overlapping (see the implementation of the eager release model in Section 3.3.3). However, a small diff unit can result in more network transmission overhead because the system could consider two consequent modifications on large data as distinct modifications. In this case, the diffs could not be merged, resulting in the transmission of two diffs instead of one. Since each diff must be transmitted with its header (6 bytes in the current implementation), transmitting useless diffs must be avoided especially if the diff data is small. For example, if two workers are each incrementing one half of an array of $2N$ integers (with the size of an integer being 4 bytes), and the diff unit is specified as 1, each incrementation will be considered as a different modification, resulting in the transmission of $2N$ diffs of 1 byte as shown in Figure 4.12. This transfers at least $14N$ bytes (the header of 6 bytes and the data of 1 byte). Instead, if using a diff unit of 4, only two large diffs would be needed after diff merging as shown in Figure 4.12, reducing the need for network transfer to $2 \times (6 + 4N)$ bytes.

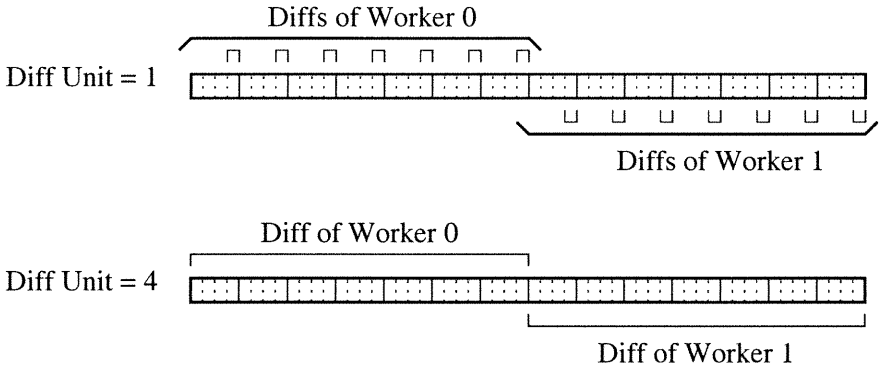

Figure 4.12: Diff Units of 1 and 4 for the Integer Array Incrementation.

Using a diff unit of 8 or more would result in more network use because the diff unit would be larger than the diff header. However, this could still be useful, not to reduce the network congestion, but to reduce the diff application overhead by needing to apply less diffs.

When using the annotated version of eager release WU model of which the type in the system is `DSM_REGION_EAGER_RELEASE_ANNOTATED_HOME_WRITE_UPDATE` or equivalently `DSM_REGION_ER-`

AHWU, the functions DSM_BeginAccess and DSM_EndAccess must be used to indicate the SDSM of the beginning and the end of write accesses in order to create the twins. When indicating write accesses, the access type must be DSM_REGION_ACCESS_WRITE. Read accesses do not need to be annotated as they can always be performed on WU memory. To apply diffs, the data from the diffs is directly copied from the network message to the memory. The diff data must also be copied to the existing twins (if applicable) to avoid detecting false write to the memory.

When using the VMH version of eager release WU model, of which the type in the system is DSM_REGION_EAGER_RELEASE_VMH_HOME_WRITE_UPDATE or equivalently DSM_REGION_ERVHWU, the twin creation is done by the VMH. The memory permission is initially set to read-only, the twins being created on page-faults. When a twin is created, the memory permissions are set to read-write, allowing both read and write accesses to perform normally. Applying diffs in the VMH version is more complicated than in the annotated version. The diff data cannot simply be copied to the memory and to the twins because this would require removing the write protection from the memory. In doing so, modifications could be lost because a write could be performed by the worker thread while the diffs are applied by the manager thread, missing a twin creation. To solve this problem, the diffs are stored locally and the permissions to the memory are removed. When a page-fault occurs, or when the region is unmapped, or when a "release" is done, these diffs are applied by the worker and the permissions are granted back to the memory. With this diff application algorithm, the worker thread applies itself the diffs, avoiding the problem of lost modifications.

Eager Release Write-Invalidate Model

The eager release write-invalidate model is intended to be used for shared memory that is more often modified than read. To implement this model, a centralized protocol where the creator of the region is the home is again used. The implementation is based on the conservative algorithm for eager release consistency presented in Section 3.3.3. The algorithm is:

- When a worker other than the home maps the region, all the memory is allocated without any permission (invalid) and the mapping worker registers itself at the home;
- When a LOAD access to an invalid granule is performed, a copy of this granule is fetched from the home;
- When the first STORE on a granule following a "release" is performed, a twin of the

modified granule is created and all subsequent STOREs perform normally;

- When the function `DSM_Release` is called, the diffs are computed and sent to the home;
- When all the diffs have been sent to the home, the corresponding invalidations are sent to all workers caching the region.

For the clients to know which workers are caching the region, the home includes a list of the workers caching the region with the diffs reception acknowledgment.

On reception of the caching workers list, the releasing worker sends each worker the invalidation. Then, the releasing worker waits for the invalidation acknowledgments from all the workers in the list. When all the acknowledgments have been received, the function `DSM_Release` is allowed to return.

As the home receives diffs before the releasing worker sends the invalidations, a worker fetching an invalid granule will always fetch an up-to-date version of that granule. Moreover, as the home maintains the valid copy of the memory, it always has a valid copy of the memory. Thus, a read by the home cannot cause a page-fault and is always a local operation.

It is possible to have write permission on a granule without having a valid read copy of the memory. As the diffs are computed from the twins, even if the twins contain invalid data, the algorithm still works (when all the diff unit is modified). However, if a read fault occurs on a granule with write permission, the diffs must be computed and stored locally before fetching the data from the home to avoid losing the current modifications. When the data has been fetched from the home, the diffs are applied back to the data to reflect the previous modifications.

As for the WU version of the eager release model, the diff unit must be specified at region creation. The reader is referred to the previous section for a complete discussion about this parameter.

When using the VMH version of the eager release WI model, of which the type in the system is `DSM_REGION_EAGER_RELEASE_VMH_HOME_WRITE_INVALIDATE` or equivalently `DSM_REGION_ERVHWI`, the twin creation and the detection of invalid accesses are done by the VMH. As for the implementation of the eager release WU model, when applying the diffs at the home, care must be taken to avoid losing STOREs to the memory (see the previous section for a more detailed description and the solution to this problem).

The annotated version has not been implemented yet. Its implementation introduces more problems than the implementation of the VMH version because, when read permissions are granted on a granule, these permissions cannot be dynamically removed on reception of an invalidation as the programmer supposes he will have a valid copy of the granule until his next annotation. To implement an annotated version of the eager release WI model, on reception of an invalidation on a granule where read accesses have already been granted, the invalidation acknowledgment needs to be replaced by a diff request to update the memory. The "release" can only be completed when all invalidation acknowledgments have been received and all diffs requests from caching workers have been answered.

4.2.4 Synchronization Primitives

This section presents the implementation of the three synchronization primitives currently available in the system. These primitives are the "init", a standard mutex and the function `DSM_Release`. Other primitives, such as condition variables, semaphores, tree barriers, special mutexes and mutexes and barriers including "acquire" and "release" could be implemented in the future.

Init

The "init" is provided to allow the synchronization of workers at startup when using the static partitioning. It is basically a barrier that is "lowered" when all the workers have "reached" it. In the current implementation, it is the only way to perform a global barrier as no barrier types have been implemented.

As it does not need to be particularly efficient due to its seldom use, the "init" implementation uses a centralized protocol where the home is the server. When calling the function `DSM_WaitInit`, a worker sends a message to the server indicating that it has "reached" the "init". The server notifies all workers to "lower" the "init" when all the "reach" messages have been received from all the workers.

This simple implementation is satisfactory as, when the real barriers will be implemented, the "init" will only be used at system initialization.

Standard Homeless Mutex

The standard homeless mutex, of which the type in the system is `DSM_MUTEX_STD_HOMELESS` or equivalently `DSM_MUTEX_SHL`, implements a standard mutex with a FIFO policy for serving lock requests. The implementation of this mutex type follows the algorithm of [NTA96] mentioned in Section 3.3.4.

When a mutex of this type is created (using the function `DSM_CreateMutex`), only the mutex type (`DSM_MUTEX_SHL`) needs to be specified. When the mutex is created, it is ready to be "locked" by a worker. The mutex must always be "unlocked" by the same worker that "locked" it.

Implementing this algorithm provides the programmer with efficient FIFO mutexes where the number of messages needed to "lock" a mutex is $O(log(n))$ for the average case ([NTA96]). This algorithm is based on a mutex owner which is the last worker that has "locked" the mutex, and on a distributed next queue indicating to which worker give the mutex when "unlocking" it. The implementation follows these simple rules:

- Each worker stores a hint to the owner of the mutex and a pointer to the next worker to give the mutex to (*next*);
- When a worker initializes a mutex, the owner is set to the first owner of the mutex;
- When a worker "locks" a mutex, a lock request is sent to the owner using the owner hint, the owner hint is set to self, and *next* is also set to self;
- When a worker which owner hint is not set to self receives a lock request, it forwards the request to the owner using its owner hint, and update its owner hint to the requesting worker;
- When a worker which owner hint is set to self receives a lock request for a mutex that is not locked, the mutex is granted to the requesting worker, and its owner hint is set to the requesting worker;
- When a worker which owner hint is set to self receives a lock request for a mutex that is locked, the owner hint is set to the requesting worker, and *next* is set to the requesting worker;
- When a worker "unlocks" a mutex, if *next* is not set to self, the mutex is granted to the worker indicated by *next*.

A more formal and unambiguous description of this algorithm is presented using pseudo-C language in Figure 4.13.

```
typedef struct {
  int      bLocked;
  Worker_t Hint;
  Worker_t GiveTo;
} MutexSHLData_t;

InitMutexSHL(MutexSHLData_t *pData, Worker_t FirstOwner) {
  pData->Hint    = FirstOwner;
  pData->bLocked = 0;
}

LockMutexSHL(MutexSHLData_t *pData) {
  pData->bLocked = 1;
  pData->GiveTo  = "current worker";
  if (pData->Hint != "current worker") {
    "send a lock request to pData->Hint";
    pData->Hint ="current worker";
    "wait for the mutex to be granted";
  }
}

ReceiveLockRequest(MutexSHLData_t *pData, Worker_t Requester) {
  if (pData->Hint != "current worker") "forward lock request to pData->Hint";
  else if (pData->bLocked) pData->GiveTo = Requester;
  else "granted mutex to Requester";
  pData->Hint = Requester; /* <-- Here is what makes the algorithm O(log(n)). */
}

UnlockMutexSHL() {
  pData->bLocked = 0;
  if (pData->GiveTo != "current worker") "grant mutex to pData->GiveTo";
}
```

Figure 4.13: Standard Homeless Mutex Implementation.

A potential optimization to the algorithm of the standard homeless mutex is the enforcement of locality on mutex ownership. As the mutex ownership must travel on the network, two workers repeatedly "locking" and "unlocking" the same mutex in a short interval cause excessive network traffic. Moreover, as the mutex is sent back and forth between these two workers, a lots of time is spent in the network waiting for the mutex. Enforcing locality in this case would introduce a short wait (like the Δ of sequential consistency presented in Section 3.3.3) before giving the mutex to the next worker in the pending queue. With this optimization, if a worker "locks" the mutex twice in a short amount of time, the network transaction is avoided and this could result in a more efficient implementation in some cases. However, the FIFO behavior of the mutex is lost. Moreover, care must be taken to avoid always granting the mutex to the same worker, causing starvation. To avoid this starvation, a maximum number of times a mutex can be "locked" enforcing locality can be introduced.

DSM_Release

The function `DSM_Release` calls a "release" operation on all the regions implementing release consistency models. When called, this function blocks until the completion of all "releases". The actions performed by a "release" operation is region type-specific. The reader is referred to the eager release consistency implementation presented in Section 4.2.3 for more information.

4.2.5 Bag of Tasks

The implementation of the bag of tasks uses a centralized protocol where the home is the server. The functions `DSM_GetTask`, `DSM_CommitTask` and `DSM_ReplaceTask` send one or more messages to the server, and the server immediately answers these request.

The centralized implementation of the BOT is satisfactory only if the tasks take enough time to execute to avoid overloading the server with requests. If too many small tasks are created, the overhead of requesting and committing tasks becomes noticeable and the performance of the application deteriorates.

To reduce the task request and commit overhead, task prefetching can be used. While executing a task, the manager can prefetch a task to avoid the idle time caused by the task request. This optimization is not currently implemented in the system.

4.3 Utilization

This section describes the utilization of YADL. The differences between the static and the BOT partitioning are explained in Section 4.3.1. The startup of a SDSM program is then discussed with a description of the server and client parameters in Section 4.3.2. Typical static and BOT partitioning programs are presented and explained in Section 4.3.3. Finally, the utilization of the shared memory and the synchronization primitives are explained in Sections 4.3.4 and 4.3.5.

4.3.1 Static and Bag of Tasks Partitioning

When using the static partitioning, a programmer must rely on worker indexes and the number of workers to assign tasks to workers. These parameters can be obtained by calling the functions `DSM_GetWorkerIndex` and `DSM_GetNumberOfWorkers` respectively. The worker indexes are arbitrarily assigned to each worker at program startup. Using this partitioning method, a programmer can easily develop a SDSM parallel program. However, when running his application, the user cannot dynamically add or remove workers to the computation.

When using the bag of tasks partitioning, the user can theorically add or remove a worker to the computation. The worker addition is almost implemented, but the worker removal is not currently implemented. When a computation is started using the BOT partitioning, a special initial task is placed in the BOT by the server. When a worker gets this task, it usually substitutes it by many tasks that correspond to the computation to be executed. In addition to the operations get and substitute, a worker can commit a task to indicate its completion. Committing a task allows the BOT management to assign tasks that depend on the completion of the committed task (see Section 4.1.1 for more information about task dependencies). The task addition, where a task is inserted in the BOT without prior dependencies or without substituting an existing task, is not currently implemented. It is not currently needed because substituting an initial task can express most of the computation. It could be incorporated in the system in the future to allow the implementation of transactional systems.

The attributes of a task are its type and its data. The task data is a buffer of size `DSM_TASK_DATA_SIZE` that can be used to store any information relevant to the task execution. The current value of this constant is 512 bytes, but can be increased by changing the code of YADL and recompiling the library. The task type is an integer used to indicate which kind of work must be done. The task types less than 100 are reserved to special use by the system, and the task type 100 corresponds to the special initial task placed in the BOT. The information residing in the task data buffer is usually parsed according to the type of a task. The data associated with the task type 100 is a string specified at program startup.

4.3.2 System Startup

To run a parallel program using YADL, a server and many clients must be started. These two operations have been included in the function `DSM_Start`.

As seen in Section 4.1.2, the two parameters given to `DSM_Start` have the same specifications as the one received from the function `main`. These two parameters should be exactly the one from the function `main`. When a "-s <server>" is specified as the program arguments (that will be forwarded to `DSM_Start`), it indicates that the program is a client that must connect to the server <server>. In this case, the function `DSM_Start` returns to allow the worker to perform its computation. If the "-s" is not present, `DSM_Start` starts a server and never returns.

More precisely, the exact way `DSM_Start` behaves when no "-s" parameter is present is by transforming the current executing program to a server using the `exec` UNIX system call. To find the server program, the environment variable `DSM_SERVER_PATH` must be set to the directory where the executable `DSMServer` can be found. Prior to executing the server program, `DSM_Start` parsed all its parameters to give them correctly to the server.

The parameters that can be specified to the server, and that are forwarded from `DSM_Start` when "-s" is not present, are the following:

- `"-d <data>"` specifies to start the server in BOT partitioning mode (the static partitioning is the default when `"-d"` is not specified) and initializes the initial task with `<data>`;
- `"-h <node>"` specifies to start a client on node `<node>`;
- `"-n <integer>"` specifies the number of clients to wait for before beginning the computation;
- `"-e <program>"` specifies the program to remotely execute as clients with its full path;
- `"-p <parameters>"` specifies the parameters to be given to the remotely executed program;
- `"-x"` specifies to start the clients in "xterm" windows using the `DISPLAY` environment variable as the X server address where the "xterm" is displayed.

To be able to add clients to the system, the server prints its address. This address is used to manually start clients using the "-s" parameter.

Receiving the parameters from the function `main` causes a problem: the separation of the parameters of the program and those of the SDSM. This problem is solved by putting "`--`" between the program parameters and the SDSM parameters, the program parameters being placed first. The UNIX function `getopt` is used by YADL to parse its arguments and this function stops when meeting "`--`". If `getopt` is also used by the program to parse its arguments, the `getopt` of YADL will resume the parsing where the application stopped. A common error that introduces bugs is to forget to put "`--`" in the program parameters or to forget to call `getopt` before `DSM_Start`.

A list of examples explaining how to use the server and client parameters follows:

- `"DSMServer -e /home/user/prg -p "-n 1024" -h node0 -h node1 -x"` starts a server in static partitioning mode that remotely spawns "xterm" on "node0" and "node1". These "xterm" execute this program: `"/home/user/prg -n 1024 -- -s <server address>"`;
- `"DSMServer -e /home/user/prg -d 123:456:789 -h node0 -h node1"` starts a server in BOT partitioning mode with the initial task having "123:456:789" as data. The server also remotely spawns the following program on "node0" and "node1": `"/home/user/prg -- -s <server address>"`;
- `"DSMServer -n 4"` starts a server in static partitioning mode waiting for four workers;
- `"DSMServer -d 123:456:789"` starts a server in BOT partitioning mode. As soon as a worker joins the system, it starts its execution;
- `"DSMServer -d 123:456:789 -n 4"` starts a server in BOT partitioning mode waiting for four workers before allowing to start their executions;
- `"/home/user/prg -- -h node0 -h node1"` is equivalent to `"DSMServer -e /home/user/prg -h node0 -h node1"`;
- `"/home/user/prg -n 1024 -- -h node0 -h node1 -x"` is equivalent to `"DSMServer -e /home/user/prg -p "-n 1024" -h node0 -h node1 -x"`;
- `"/home/user/prg -- -d 123:456:789 -h node0 -h node1"` is equivalent to `"DSMServer -e /home/user/prg -d 123:456:789 -h node0 -h node1`.
- `"/home/user/prg -n 1024 -- -s 48568:132.204.25.170"` executes the SDSM program "prg" giving 1024 as the "-n" parameter to the program (not the SDSM). This program

is run as a client that must connect to server "48568:132.204.25.170"[1].

4.3.3 Typical Programs

A Typical Static Partitioning Program

A typical SDSM program using the static partitioning method is presented in Figure 4.14 (complete examples are given in Appendix B). The first operation to be done in a static partitioning program is the parsing of the command line arguments up to the "--" using the UNIX function `getopt`. Then, `DSM_Start` can be called with the parameters received from the function `main`.

After having called the startup function, the initialization of the computation must be performed. This initialization includes the creation and initialization of shared memory regions and synchronization primitives. The initialization is done by a single worker, usually the worker whose index is zero (by convention, worker zero acts as the main program). All the workers must wait for the end of the initialization using a barrier. As the barriers used in the computation are created at initialization, the workers must use the barrier "init", provided exactly for this purpose (see Section 4.1.2).

After having completed the initialization, each worker performs its task according to its index and the number of workers. Once their tasks completed, the workers wait at a barrier to allow all workers to finish their computations. When this barrier has been lowered, one worker, usually worker zero, gathers the result of the computation and either saves it to disk or displays it. Finally, the workers wait at a barrier for the end of the saving process, call `DSM_Stop` to clean the environment, and terminate the program.

A Typical BOT Program

A typical SDSM program using the BOT partitioning method is presented in Figure 4.15 (a complete example is given in Appendix B). The first operation to be done in a BOT partitioning program, as in a static partitioning program, is the parsing of the command line arguments up to the "--" using the UNIX function `getopt`. Then, `DSM_Start` can be called with the parameters

[1]The network addresses in UDN follow the format <port number>:<IP address> instead of the standard notation where the IP address precedes the port number. This eases the parsing of network addresses.

```
/* Program-specific include file. */
...

#include "DSM.h"

int main(int argc, char *argv[]) {
  /* Variable declarations. */
  int c, id0, id1, nPerW, nLeft, ns, ne, NbTasks;
  ...

  /* Manage the program parameters. */
  while ((c = getopt(argc, argv, /* The parameters. */)) != -1)
    switch(c) {
      ...
    }

  DSM_Start(argc, argv)

  if (DSM_GetWorkerIndex() == 0) { /* Initialization, done by worker zero. */
    ...
  }

  DSM_WaitInit(1); /* Wait for the end of initialization. */

  /* Compute the number of tasks (application specific). */
  NbTask = ...;

  /* Perform tasks according to worker index and the number of workers. */
  id0   = DSM_GetWorkerIndex();
  id1   = id0 + 1;
  nPerW = NbTasks / DSM_GetNumberOfWorkers();
  nLeft = NbTasks % DSM_GetNumberOfWorkers();
  ns    = id0 * nPerW + id0 * nLeft / DSM_GetNumberOfWorkers();
  ne    = id1 * nPerW + id1 * nLeft / DSM_GetNumberOfWorkers();

  ...

  DSM_WaitInit(98); /* Wait for the completion of all tasks. */

  if (DSM_GetWorkerIndex() == 0) { /* Save the result. */
    ...
  }

  DSM_WaitInit(99); /* Wait for the completion of the saving. */

  DSM_Stop();

  return 0;
}
```

Figure 4.14: A Typical Static Partitioning YADL Program.

received from the `main`.

After having called the startup function, the workers enter a task loop. As long as all the tasks have not been completed, the workers repeatedly get, execute, and either commit or replace tasks.

As seen in Section 4.3.1, the special task numbered 100 is the initial task placed in the BOT. The data associated with this task is specified at program startup. When a worker executes this task, it usually initializes the computation, creates and initializes shared memory regions and synchronization primitives. After initialization, the initial task is usually substituted by two tasks: the computation to be done and the saving process, the saving process depending on the completion of the computation task. The task of the computation is also eventually replaced by many tasks corresponding to the computation partitioned in many parallel tasks.

When a task is available, `DSM_GetTask` returns the constant `DSM_OK`. When the BOT is empty, but there still exists tasks to be committed and potentially be replaced, `DSM_GetTask` returns the constant `DSM_NO_TASK_AVAILABLE`. When the BOT is completely empty, all tasks having been committed, `DSM_GetTask` returns the constant `DSM_NO_MORE_TASK`. In this last case, the task loop is exited and `DSM_Stop` is called.

Returning the constant `DSM_NO_TASK_AVAILABLE` when no tasks are available forces the worker to actively wait in a loop for a task to be available or for the BOT to become empty. As active waiting should usually be avoided, this implementations seems a bad choice. However, as workers should be allowed to quit the computation at any time, active waiting must be used to allow a user to remove a worker at any time, especially when no tasks are available. If the worker was blocked waiting for a task, removing it from the computation would be more complicated because it would require the interruption of the `DSM_GetTask` function.

When `DSM_GetTask` returns `DSM_OK`, a task has been assigned to the worker. The worker must perform this task according to its type and data. When the task has been completed, it must be committed or replaced. Forgetting to commit or replace a task leads to a "task leak" in the BOT, resulting in the non-termination of the program.

When using the BOT partitioning, the "init" barrier must not be used as the number of workers in the computation can dynamically change. Global barriers are represented as task dependencies and substitutions. Eventually, when barrier types will be implemented, programmers

```
/* Program specific include file. */
...

#include "DSM.h"

int main(int argc, char *argv[]) {
  /* Variable declarations. */
  int bLoop = 1;
  DSMAddTask_t *pAddTasks;
  DSMTask_t Task;
  ...

  while (getopt(argc, argv, "") != -1) ; /* Read up to the --. */

  DSM_Start(argc, argv);

  while (bLoop) { /* The task loop. */
    switch(DSM_GetTask(&Task)) { /* Try to get a task. */
    case DSM_OK: /* A task is available and will be executed. */
      switch(Task.nTaskType) { /* Perform the task. */
      case 100: /* Initial task. */
        /* Parse BOT Data. */
        ...

        ...

        DSM_ReplaceTask(&Task, pAddTasks, nNbTasks);
        free(pAddTasks);
        break;
      case ...: /* A particular task type. */
        ...
        DSM_CommitTask(&Task); /* Commit this task. */
        break;
      }
      break;
    case DSM_NO_TASK_AVAILABLE: /* No task currently available. */
      usleep(1000); break;
    case DSM_NO_MORE_TASK: /* All the tasks have been completed. */
      bLoop = 0; break;
    default:
      printf("Error.\n"); bLoop = 0; break;
    }
  }

  DSM_Stop();

  return 0;
}
```

Figure 4.15: A Typical Bag of Tasks YADL Program.

will be able to use barriers in BOT programs.

4.3.4 Shared Memory

To use the shared memory, a region must first be created by calling the function `DSM_CreateRegion`. When creating a region, the region number, type, size, granularity and type-specific parameters are provided using the region data parameter (`pRegionData`).

The five region types available are:

DSM_REGION_WOAH: the write-once annotated home region type implementing the write-once algorithm presented in Section 4.2.3;

DSM_REGION_WOVH: the write-once VMH home region type also implementing the write-once algorithm presented in Section 4.2.3

DSM_REGION_ERAHWU: the eager release annotated home WU region type implementing the eager release WU algorithm presented in Section 4.2.3;

DSM_REGION_ERVHWU: the eager release VMH home WU region type implementing the eager release WU algorithm presented in Section 4.2.3;

DSM_REGION_ERVHWI: the eager release VMH home WI region type implementing the eager release WI algorithm presented in Section 4.2.3.

Once a region which accesses are detected by the VMH is mapped using the function `DSM_MapRegion`, it can be used like normal memory. If the accesses are detected using annotations, the programmer must use the functions `DSM_BeginAccess` and `DSM_EndAccess` to notify the system to perform the required consistency operations. The function `DSM_PrefetchAccess` can be used to perform consistency operations prior to the access of shared memory to attempt avoiding blocking when calling `DSM_BeginAccess`.

The use of the three functions `DSM_BeginAccess`, `DSM_EndAccess` and `DSM_PrefetchAccess` are region type-specific. Some region types may not require the programmer to indicate the read accesses, but only the write accesses. A list of access notifications required for the two annotated region types currently provided by YADL follows:

DSM_REGION_WOAH: the first write access to write-once annotated home region must be

notified (the initialization) using the access type `DSM_ACCESS_READ_WRITE`. All the subsequent read accesses must be notified using the access type `DSM_ACCESS_READ`. Read prefetching can be used to reduce the time the system is blocked waiting for data to be imported to the local worker.

DSM_REGION_ERAHWU: the write accesses to eager release annotated home WU region must be notified using the access type `DSM_ACCESS_WRITE`. The read accesses do not need to be notified as the cache is always up to date (write-update protocol). Prefetching cannot be done.

When using eager release region types, the diff unit parameter must be specified (see Section 4.2.3). The accepted values for the diff unit is 1, 2, 4, and 8. This parameter is specified using the `SpecificData` field of the `pRegionData` parameter. This field is a union having a type-specific structure for each region type. The diff unit field are:

- `pRegionData->SpecificData.ERAHWU.nDiffUnit` for the eager release annotated home write-update region;
- `pRegionData->SpecificData.ERVHWI.nDiffUnit` for the eager release VMH home write-invalidate region;
- `pRegionData->SpecificData.ERVHWU.nDiffUnit` for the eager release VMH home write-update region.

When creating regions, the granularity of the region must be specified. The valid granularities are any power of two beginning from 2^5 to 2^{30}: `DSM_REGION_GRAN_32`, `DSM_REGION_GRAN_64`, `DSM_REGION_GRAN_128`, and so on. If the programmer does not want to use these constants, he can use the exponents corresponding to the granularity: 5 for 32, 6 for 64, 7 for 128, and so on. Of course, a granularity larger than or equal to `DSM_REGION_GRAN_4096` (2^{12}) must be used when using a VMH region because the VM page size is 4096 bytes (4 kilobytes).

If eager release memory is used, the release primitive must be used to propagate modifications. To do a "release", the function `DSM_Release` must be called.

When unmapping a region, the memory allocated for the shared memory is not freed immediately when the function `DSM_UnMapRegion` is called. The region becomes in the cached state where it is still considered as mapped. In the cached state, a region still receives data

regarding consistency operations. Using a cached state for region allows the mapping following an unmapping to be a very fast operation. The garbage collecting of the cached region is done when no more memory is available when mapping regions. However, this garbage collection is not implemented yet.

4.3.5 Synchronization Primitives

The two synchronization primitives currently provided by YADL are the "init" and the standard mutex. More primitives, including tree barriers, will be implemented in the future.

To use the "init", the programmer only has to call the function `DSM_WaitInit` with an integer as argument. This integer's purpose is only for clarity of code and debugging: when the "init" is performed, the integers are compared and must be identical.

To use the standard mutex, of which the type in the system is `DSM_MUTEX_SHL`, the mutex must be created using the function `DSM_CreateMutex`. No mutex type-specific data must be given when creating this mutex. When a critical section must be entered, the mutex is locked using the function `DSM_LockMutex` with the number of the mutex and the flag `DSM_FLAGS_NONE`. This function will only return when the mutex is granted. When exiting the critical section, the function `DSM_UnlockMutex` must be called to "unlock" the mutex and to allow other workers to enter the critical section.

Chapter 5

YADL Evaluation

The last chapter has presented the specification, implementation and the utilization of YADL. After knowing how is built and implemented YADL, a question stays unanswered: is YADL efficient? This chapter answers this question by presenting an evaluation of YADL.

The description of the benchmark programs used for the evaluation is given in Section 5.1. The results of the evaluation are presented and discussed in Section 5.2. Finally, a general discussion is presented in Section 5.3, including the future development that can be done on YADL.

5.1 Benchmark Programs

The benchmark programs used to evaluate the performance of YADL are: matrix multiplication, ray tracing, Mandelbrot fractal, fast matrix exponentiation, n-queens and tridiag. Their description follows.

5.1.1 Matrix Multiplication

The matrix multiplication program uses the traditional $O(n^3)$ algorithm (not the block version). The static and the BOT partitioning are evaluated using square matrices of size 4096x4096 ini-

tialized with random numbers. When using the static partitioning, each worker has to compute $4096/NbWorkers$ consecutive lines of the resulting matrix. In the BOT partitioning version, a task corresponds to the computation of four consecutive lines of the resulting matrix for a total of 1024 tasks. The two matrices to be multiplied are stored in two write-once annotated home regions, and the resulting matrix, in an eager release VMH home write-invalidate region. Because the matrices to be multiplied are never modified and the resulting matrix is only read by one worker, these choices of region types are unambiguous.

The timer measuring the execution time is started after the initialization of the matrices. The data distribution of the matrices (the time needed to send these matrices over the network to all workers), is included in the execution time. The timer is stopped after one worker has read the whole resulting matrix, but without including the time needed to save it to disk[1].

5.1.2 Ray Tracing and Mandelbrot Fractal

The ray tracing ([FvDFH96]) and Mandelbrot fractal ([man]) benchmarks are computationally intensive applications not requiring a lot of collaboration between the workers. Using the same partitioning strategy as the matrix multiplication does not lead to good speedups because these two applications are irregular. Because some lines of the resulting image require more time to be computed than others and because these lines are usually consecutive, some workers have much larger tasks than others, resulting in idle processors and poor load balancing.

Two solutions are proposed, one using the static partitioning and the other using the BOT partitioning. In the solution using the static partitioning, each worker has $1/NbWorkers$ of the lines of the resulting image to compute, but these lines are not assigned consecutively, they are assigned round-robin to workers. With this strategy, the consecutive lines are assigned to different workers, achieving better load balancing. In the solution using the BOT partitioning, a task corresponds to the computation of n lines of the resulting image (in these benchmarks, $n = 1$). By putting these tasks in the BOT, good load balancing is achieved because the workers executing smaller tasks will do more tasks than the workers executing larger tasks.

The ray tracing application produces an image of size 2048x1024 pixels from a scene of 114 primitives and four lights, throwing four rays per pixel for antialiasing. The ray tracing

[1] Reading the matrix is essential to generate the page-faults that import the data in the local cache of the saving worker.

implementation intersects a ray with all the primitives, no bounding box being used. An overview of the resulting image is given in Figure 5.1. The timer measuring the execution time is started after the initialization of the scene. The data distribution of the scene to all workers is included in the execution time. The timer is stopped when one worker has saved the resulting image to disk. For the same reasons as for the matrix multiplication, the scene is stored in a write-once annotated home region and the resulting image, in an eager release VMH home write-invalidate region.

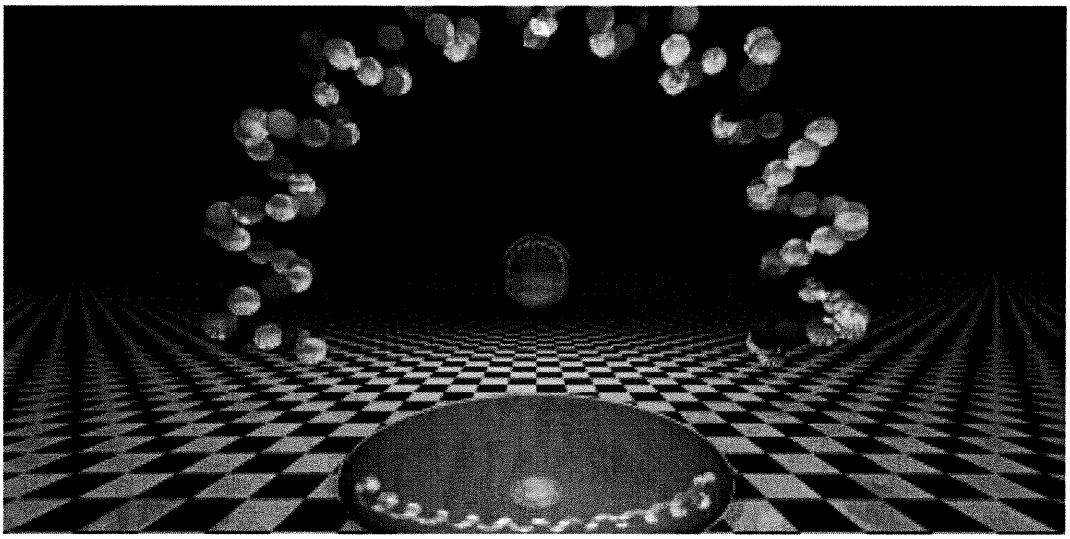

Figure 5.1: The Resulting Image of the Ray Tracing Benchmark.

The Mandelbrot fractal application produces an image of size 2048x1024 pixels where the lower left corner of the image is placed at $(0.2787636 - 0.009297555i)$. The image width is 2.5×10^{-7}, using 60,000 iterations and no antialiasing. An overview of the resulting image is given in Figure 5.2. The timer measuring the execution time is started after the creation of the shared memory to store the image. The timer is stopped when one worker has saved the resulting image to disk. Again, for the same reason as for the matrix multiplication, the resulting image is stored in an eager release VMH home write-invalidate region.

5.1.3 Fast Matrix Exponentiation

The fast matrix exponentiation benchmark measures the performance of an iterative process collaborating between iterations, but with computationally intensive iterations. This benchmark

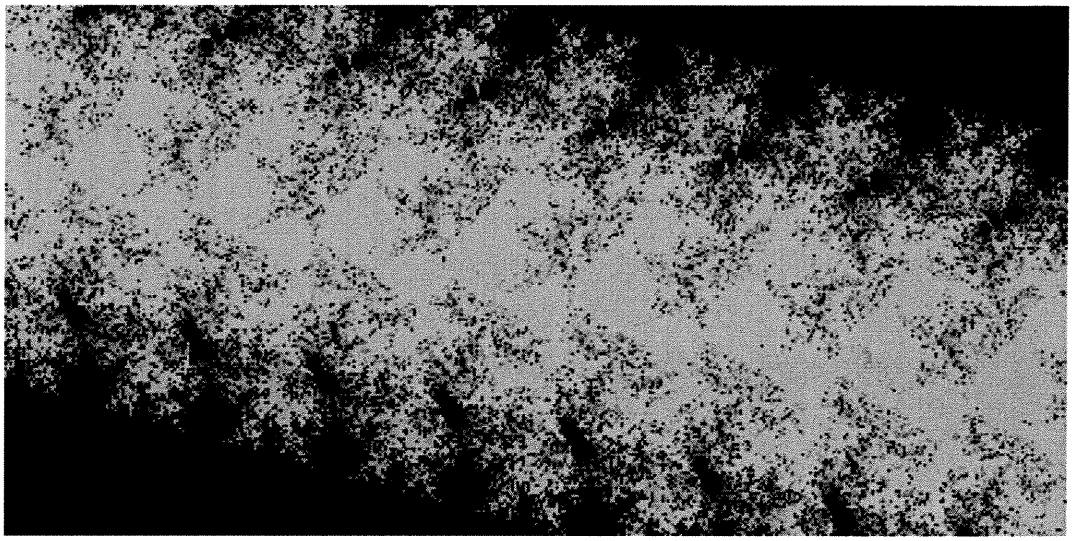

Figure 5.2: The Resulting Image of the Mandelbrot Fractal Benchmark.

stresses YADL more than the matrix multiplication as it uses the result of the previous iteration to execute the next iteration, thus requiring more consistency operations.

Each iteration is a matrix multiplication $O(size^3)$. The total computation complexity is $O(size^3 \times \log(exp))$ as the fast matrix exponentiation algorithm is used (see Figure 5.3 for the pseudo-code of the algorithm). After each iteration, the workers have to exchange the newly computed matrix to be able to perform the next iteration. The matrix to be exponentiated is stored in a write-once annotated home region because it is never modified, and the resulting matrix, in an eager release annotated home write-update region. The choice of the WU protocol is motivated by the need, by each worker, to access the whole matrix between iterations.

```
void CopyMatrix(Matrix_t *pFrom, Matrix_t *pTo);
void MatMult(Matrix_t *pA, Matrix_t *pB, Matrix_t *pResult);

void FastMatExp(Matrix_t *pToExp, int Exp, Matrix_t *pResult) {
  if (Exp == 1) CopyMatrix(pToExp, pResult);
  else if (Exp % 2 == 1) {
    FastMatExp(pToExp, Exp-1, pResult);
    MatMult(pResult, pToExp, pResult);
  }
  else {
    FastMatExp(pToExp, Exp/2, pResult);
    MatMult(pResult, pResult, pResult);
  }
}
```

Figure 5.3: The Fast Matrix Exponentiation Algorithm.

Again, the static and the BOT partitioning are compared. The problem size is a square matrix of size 2048x2048 initialized with random numbers and the exponent is 31. This exponent was chosen to stress equally the odd and even parts of the fast matrix exponentiation algorithm. The task partitioning is the same as for the matrix multiplication. For the static partitioning version, the task of each worker is the computation of $2048/NbWorkers$ consecutive lines. For the BOT partitioning version, a task corresponds to the computation of two consecutive lines of the resulting matrix for a total of 1024 tasks.

The timer measuring the execution time is started after the initialization of the matrix. The data distribution of the matrix to all workers is included in the execution time. The timer is stopped after one worker has read the whole resulting matrix, but without including the time needed to save it to disk.

5.1.4 N-Queens

The n-queens benchmark measures the performance of a mutex based application. It computes the number of solutions for placing n queens on a n by n chessboard without any queen threatening another queen. To parallelize the application, queen positions for the first l lines of the chessboard are precomputed, each of these positions corresponding to a task.

To manage these tasks, we can use the BOT or the static partitioning. When using the BOT partitioning, each precomputed position is a task inserted in the BOT. When using the static partitioning, each worker does $1/NbWorkers$ of the tasks. As for the ray tracing and the Mandelbrot fractal, assigning tasks consecutively for the static partitioning version results in poor load balancing. To solve this problem, the same solution as for the previous benchmarks is used: tasks are assigned round-robin to workers.

The number of solutions is the only data put in shared memory. The access to this variable is protected by a mutex to guarantee its consistency. The choice between an eager release annotated home write-update and an eager release VMH home write-invalidate region is not obvious. As the read-write ratio of the application is low (this ratio is one, the data being only read once before being modified), WI region type seems a good choice. However, as a read using this protocol suffers from high latency, each incrementation of the number of solutions introduces a delay. Because the WI implementation anyway sends invalidation to all workers,

and because the diffs sent using the WU implementation would be, for this application, only a little larger than the invalidations and would fit in a single packet, the WU region type could also be a good choice. Finally, as the read of the WI memory fetches uselessly 4 kilobytes of data when only 8 bytes are needed (the size of the `uint64_t` used to store the number of solutions), the choice of WU seams much better than WI.

Preliminary tests have been done using both WU and WI region types. Both lead to similar results. The results presented in the next section are the one with the WU region.

For this benchmark, the size of the chessboard is 17 and $l = 3$, resulting in 2786 tasks. The timer measuring the execution time is started after the initialization of the shared memory and of the mutex. It is stopped when all the tasks have been executed.

5.1.5 Tridiag

The tridiag benchmark solves a tridiagonal system of equations using the cyclic reduction algorithm ([tri]). This algorithm performs $log(n)$ iterations over three arrays of data, modifying all the array elements in each iteration (n is the number of variables). The complexity of the algorithm is $O(n \times log(n))$. Thus, tridiag is an iterative process whose iterations are not computationally intensive.

For this benchmark, only static partitioning is evaluated. Each worker computes $1/NbWorkers$ consecutive elements of the arrays in each iteration. The arrays of data resides in an eager release VMH home write-invalidate region. As a worker does not need all the elements of the array to compute the next values, but only a consecutive subsection of the array, this choice of region type avoids uselessly sending the update to all workers.

For this benchmark, the number of variables is 4 million (4,194,304). The value of the variables are randomly chosen at initialization, a system of equations being computed using only one as the coefficients. The timer measuring the execution time is started after the computation of the problem. The timer is stopped after one worker has verified the solution by comparing the computed solution with the expected result chosen at initialization.

5.2 Benchmark Results

This section presents the results of the execution of the seven benchmark programs presented in the previous section. Unless specified, the granularity of the shared memory region used while executing the benchmarks is one VM page (4 kilobytes).

The platform used to evaluate YADL is a NOW with the following characteristics:

- The NOW is composed of 32 dual-processor nodes;
- The processors are AMD Athlon MP clocked at 1.2 GHz;
- The motherboards are Tyan Thunder K7 model number S2462NG;
- Each node has 4 DIMMs of 256 megabytes of DDR memory for a total of 1 gigabyte of memory per node;
- Each node has a 80 gigabytes Seagate Barracuda ATA hard drive (7,200 RPM);
- The nodes are linked by a switched gigabit Ethernet network:
  - Each node has an Ethernet PCI Intel PRO/1000 TX Server Network Adapter (shared by both processors);
  - The switch is a Cisco Catalyst 6000, using a WS-C6509 9 slot Catalyst 6500 series chassis with three WS-X6316-GE-TX 16-port gigabit Ethernet modules;
  - The links between the nodes and the switch are RJ-45.
- Each node runs the Linux OS with the kernel version 2.4.9.

The results presented are the execution times in seconds and the corresponding speedups of the following versions of the benchmarks:

- Sequential version;
- SDSM version, using one worker per node, with 1, 2, 4, 8, 12, 16, 20, 24 and 28 nodes[2];
- SDSM version, using two workers per node to take advantage of the dual-processors, with 1, 2, 4, 8, 12, 16, 20, 24 and 28 nodes.

The presented execution times are the average of six executions of which the best and worst

[2]Because some nodes were down when the benchmarks were executed, 32 nodes could not be evaluated.

results were removed. The speedups are computed using the ratio of the sequential and the parallel execution times.

Speedup graphs are presented using logarithmic scale with the two perfect speedup functions for one and two workers being drawn with a continuous and dotted line respectively ($y = x$ and $y = 2x$). The speedups for the same benchmark are presented on the same graph allowing an easy comparison of different versions of the benchmark (usually the static and the BOT partitioning versions).

When using one worker per node, the execution times should be a bit shorter than they would be with a NOW built with uni-processors because the multi-threaded implementation of YADL takes partially advantage of the presence of two processors. However, these results are quite representative because the manager thread of YADL is not a computationally intensive thread. Furthermore, when using two workers per node on N nodes, the results should be longer than they would be with a NOW of $2N$ uni-processors because the two workers on the same node compete for network resource and physical memory bandwidth.

The benchmarks and the SDSM library have been compiled using the gcc compiler version 2.96 20000731 (Red Hat Linux 7.1 2.96-85) without using optimizations to forbid the compiler from removing memory accesses that would result in less page-faults. When executing the benchmarks, the YADL server is run on one of the nodes used for the computation. The server was modified to allow the workers, when using the BOT partitioning, to begin their computation only when all the workers have joined the computation, not when the first worker joins the computation. This avoids the case of some workers joining the computation a little late, causing bad speedups.

In addition to the execution times, the following information was collected for each worker while running the benchmarks. This information is not reported in this document, but they are used to analyse the results of the benchmarks.

- The number of UDN packet sent and received;
- The amount of network data sent and received;
- The number of "release" performed, the amount of data sent for performing all the "releases" and the time spent in the "releases";
- The number of times a mutex was "locked", the time spent waiting for the mutex to be

granted, and the time the mutex was held;

- The number of page-faults generated, the amount of data fetched in page-faults, and the time spent in the page-fault handler;
- The number of task gets, and the time spent waiting for tasks;
- The number of data requests that were handled, the amount of data sent for handling these requests, and the time spent while data requests were pending.

5.2.1 Matrix Multiplication

Good speedups are expected for this benchmark because matrix multiplication is a computationally intensive application not requiring a lot of collaboration between workers. The execution times and speedups for this benchmark are presented in Table 5.1, and the graph of the speedups is presented in Figure 5.4.

Sequential	1177.72							
	Static				BOT			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1115.7	1.06	567.9	2.07	1303.7	0.90	737.5	1.60
2 nodes	568.4	2.07	290.4	4.06	698.4	1.69	376.9	3.12
4 nodes	288.6	4.08	151.7	7.76	373.4	3.15	200.9	5.86
8 nodes	148.6	7.93	81.5	14.44	210.0	5.61	117.6	10.02
12 nodes	102.4	11.51	58.5	20.12	146.9	8.02	87.5	13.45
16 nodes	79.2	14.87	47.8	24.65	118.3	9.96	74.3	15.86
20 nodes	65.3	18.03	41.3	28.49	103.5	11.38	66.9	17.61
24 nodes	56.3	20.90	36.7	32.10	98.5	11.96	60.9	19.34
28 nodes	50.1	23.53	34.0	34.64	87.7	13.42	57.9	20.35

Table 5.1: Matrix Multiplication Execution Times and Speedups.

Surprisingly, the execution time of the sequential version is longer than for the static partitioning version using a single worker on one node. No precise reasons have been found to explain this 5.5% difference in the execution time. Great care was taken to reproduce as much SDSM operations in the sequential version to replicate this behavior, but without success. Possible hypothesis to explain this different are cache effects and scheduling affinity.

The static partitioning with one worker per node gives the expected good speedups. When

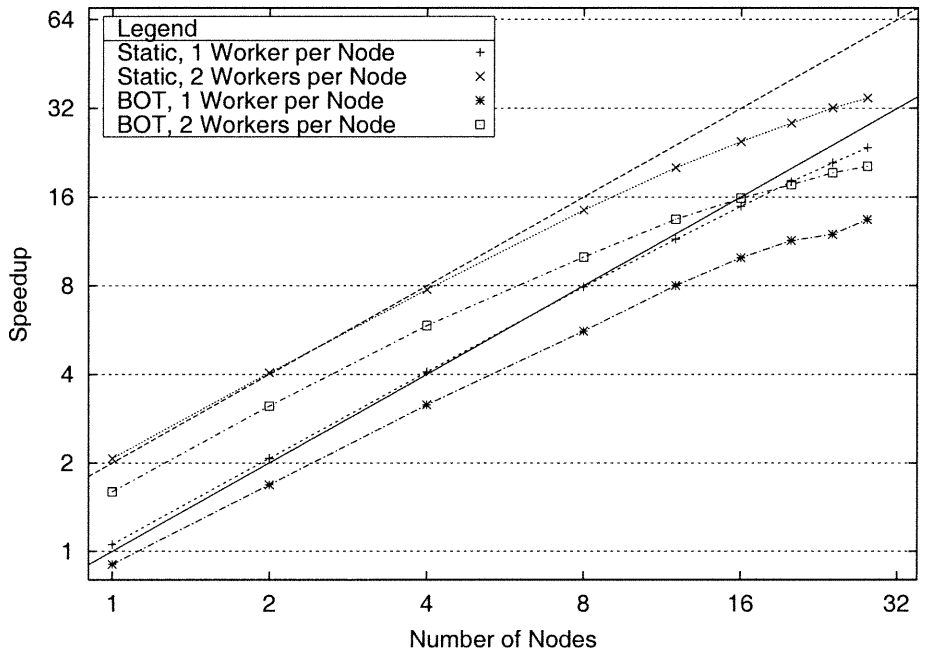

Figure 5.4: Matrix Multiplication Speedups.

using two workers per node, the speedups are a little worse than twice the speedups obtained with one worker per node. This can easily be explained by the delay required to distribute the matrices via the network. Sending 64 megabytes of data (the size of one matrix) using a write-once annotated home region to 16 nodes using one and two workers per node takes about 5 and 6.5 seconds respectively (it should take a bit more time for 28 nodes, but the exact delays have not been evaluated). When using only one worker per node, the 5 second distribution delay for 16 nodes is hardly noticeable compared to the computation time of 1/16 of the result: the expected computation time is about 74 seconds, the delay accounting for 6.8% of the execution time. However, when using two workers per node, the 6.5 second delay begins to be noticeable compared to the computation time of 1/32 of the result: the expected computation time is about 37 seconds, the delay accounting for 17.6% of the execution time. It will be even more noticeable when using two workers per nodes and 28 nodes: the expected computation time is 21.4 seconds, a 6.5 seconds delay accounting for 30.4% of the execution time (for 28 nodes, the distribution delay should even be larger).

When the data distribution delay is not considered in the execution time of the static version

of the benchmarks (by starting the timer measuring the execution time after the distribution[3]), the speedups are much better, almost perfect. This confirms the hypothesis that the acceptable, but not ideal, speedups are caused by the distribution delay. The execution times and speedups when the data distribution time is not considered are presented in Table 5.2, and the graph of the speedups is presented and compared with the previous results in Figure 5.5.

	Static Without Distribution				Static With Distribution			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1115.7	1.06	567.9	2.07	1163.4	1.01	578.7	2.04
2 nodes	568.4	2.07	290.4	4.06	583.9	2.02	290.3	4.06
4 nodes	288.6	4.08	151.7	7.76	292.6	4.02	145.8	8.08
8 nodes	148.6	7.93	81.5	14.44	146.9	8.02	73.6	16.00
12 nodes	102.4	11.51	58.5	20.12	98.2	12.00	49.4	23.86
16 nodes	79.2	14.87	47.8	24.65	74.1	15.90	37.6	31.35
20 nodes	65.3	18.03	41.3	28.49	59.4	19.82	30.2	38.94
24 nodes	56.3	20.90	36.7	32.10	49.8	23.66	25.5	46.25
28 nodes	50.1	23.53	34.0	34.64	43.1	27.35	22.1	53.28

Table 5.2: Matrix Mult. Execution Times and Speedups Without the Distribution.

The BOT partitioning gives unsatisfactory results. These could be caused by too many tasks in the BOT, overloading the BOT management service. It could also be caused by doing too many "releases", a "release" being done at the completion of each task. When doing too many "releases", a lot of time is spent doing consistency operations, explaining the flattening. Moreover, due to the implementation of the eager release VMH home write-invalidate region type, the BOT partitioning implementation does lot of page-faults avoided in the static partitioning version. In the static partitioning version, the home of the region collects the result of the computation, thus no invalid accesses are performed because the home always has a valid cache of the data. When using the BOT partitioning, the final result saving task is probably not assigned to the home, incurring lots of invalid accesses done to collect the matrix, resulting in worse execution times.

To reduce the impact of the result saving not being done by the home, the granularity of the shared memory can be increased. This causes less page-faults to occur, incurring less data

[3]The data distribution delay could be avoided by locally loading the matrices from disk and by only sharing the resulting matrix. However, this means to write these matrices on the local disk of each node, which would probably require more time than with the data distribution optimization of write-once regions.

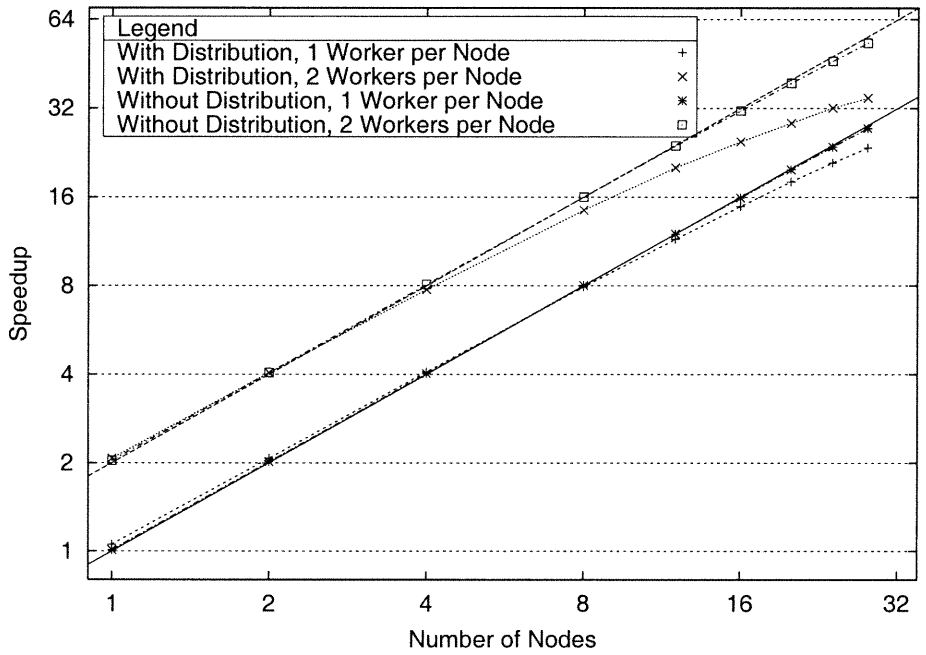

Figure 5.5: Matrix Mult. Speedups With and Without the Distribution.

requests to the home. By reducing the number of data requests, the worker saving the result suffers from less request latency. The granularity of the shared memory has been increased from 4 kilobytes to 64 kilobytes to observe the impact of less requests on the performance of the BOT partitioning version of the benchmark. The execution times and speedups with the increased granularity are presented in Table 5.3, and the graph of the speedups is presented and compared with the previous results in Figure 5.6.

As shown in Figure 5.6, increasing the granularity fills a large part of the gap between the static and BOT partitioning version of the benchmark[4]. This optimization gives good improvements for the BOT partitioning version, but the speedups are still smaller than the static partitioning version. To see the effect of having less tasks in the BOT, the BOT version has been run with 256 tasks instead of 1024. The execution times and speedups with less tasks are presented in Table 5.4, and the graph of the speedups is presented and compared with the previous results in Figure 5.7.

[4]Increasing the granularity for the static version of the benchmark was also done, but did not result in improvement of the execution time. These results are presented in Appendix D

	BOT Without Increased Granularity				BOT With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1303.7	0.90	737.5	1.60	1271.3	0.93	661.9	1.78
2 nodes	698.4	1.69	376.9	3.12	652.8	1.80	339.3	3.47
4 nodes	373.4	3.15	200.9	5.86	332.8	3.54	174.1	6.77
8 nodes	210.0	5.61	117.6	10.02	172.6	6.82	92.6	12.72
12 nodes	146.9	8.02	87.5	13.45	119.0	9.90	66.8	17.62
16 nodes	118.3	9.96	74.3	15.86	91.7	12.84	53.8	21.88
20 nodes	103.5	11.38	66.9	17.61	75.9	15.52	45.2	26.04
24 nodes	98.5	11.96	60.9	19.34	65.9	17.87	40.4	29.19
28 nodes	87.7	13.42	57.9	20.35	58.5	20.12	37.0	31.86

Table 5.3: Matrix Mult. Execution Times and Speedups With Increased Granularity.

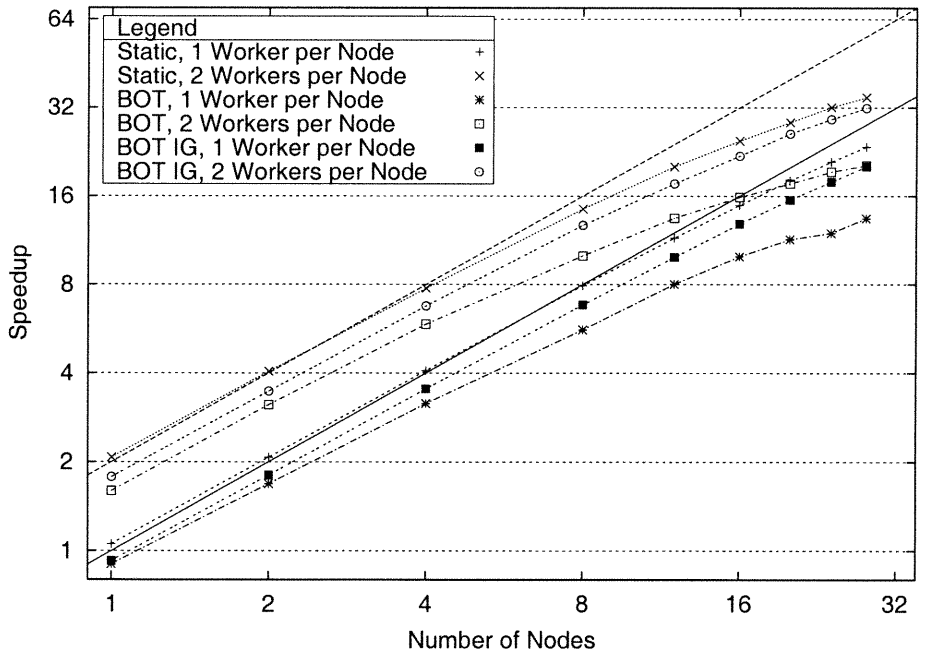

Figure 5.6: Matrix Mult. Speedups With and Without Increased Granularity (IG).

	BOT With 1024 Tasks				BOT With 256 Tasks			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1303.7	0.90	737.5	1.60	1287.5	0.91	666.8	1.77
2 nodes	698.4	1.69	376.9	3.12	663.5	1.78	355.7	3.31
4 nodes	373.4	3.15	200.9	5.86	349.3	3.37	180.5	6.53
8 nodes	210.0	5.61	117.6	10.02	185.6	6.35	105.0	11.21
12 nodes	146.9	8.02	87.5	13.45	129.8	9.07	78.2	15.06
16 nodes	118.3	9.96	74.3	15.86	106.0	11.11	60.1	19.60
20 nodes	103.5	11.38	66.9	17.61	94.5	12.47	58.5	20.14
24 nodes	98.5	11.96	60.9	19.34	94.9	12.41	53.3	22.10
28 nodes	87.7	13.42	57.9	20.35	89.0	13.23	47.8	24.66

Table 5.4: Matrix Mult. Execution Times and Speedups With Less Tasks.

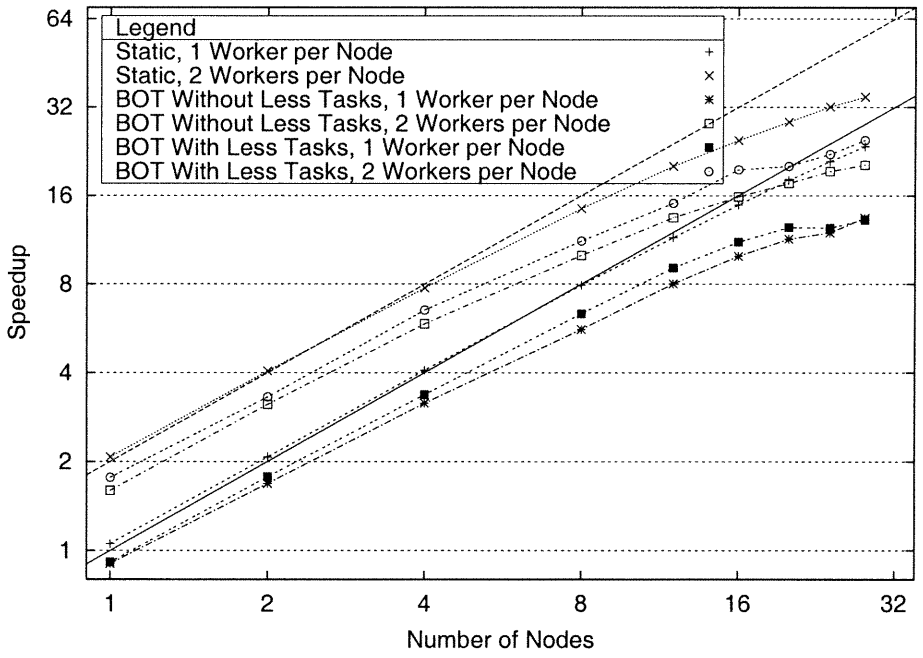

Figure 5.7: Matrix Multiplication Speedups With and Without Less Tasks.

As shown in Figure 5.7, using less tasks increases the speedups a little. Combining both optimizations, increasing the granularity and using less tasks, should give even better speedups for the BOT partitioning version of the benchmark, but has not been done. However, the number of tasks should not be reduced too much to keep the load balancing property of the BOT.

5.2.2 Ray Tracing and Mandelbrot Fractal

Good speedups are expected for these benchmarks because they are computationally intensive applications not requiring a lot of collaboration between the workers. Moreover, they are less memory intensive than the matrix multiplication, so the data distribution delay and the page-fault latency should not be a major problem. The execution times and speedups for the ray tracing and Mandelbrot fractal benchmarks are presented in Tables 5.5 and 5.6, and the graphs of the speedups are presented in Figures 5.8 and 5.9.

Sequential	1598.64							
	Static				BOT			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1621.2	0.99	812.3	1.97	1598.6	1.00	812.3	1.97
2 nodes	811.9	1.97	407.3	3.93	806.2	1.98	408.9	3.91
4 nodes	406.9	3.93	204.5	7.82	406.3	3.93	206.6	7.74
8 nodes	204.3	7.82	103.3	15.48	205.8	7.77	105.6	15.14
12 nodes	137.6	11.62	70.8	22.57	138.7	11.52	72.2	22.14
16 nodes	103.2	15.49	52.6	30.38	105.5	15.15	55.1	28.99
20 nodes	83.9	19.06	43.1	37.12	85.2	18.76	44.4	36.04
24 nodes	70.7	22.61	36.8	43.47	72.2	22.14	38.1	41.95
28 nodes	61.5	25.97	32.7	48.94	63.5	25.16	34.2	46.72

Table 5.5: Ray Tracing Execution Times and Speedups.

As expected, the speedups are quite good. There is practically no difference between the results of the static and BOT partitioning versions, which is surprising. This can be explained by the difficulty of seeing the BOT partitioning overhead because this overhead is very small in these two applications.

Increasing the granularity and using less tasks was also done. However, because the resulting image is relatively small compared with the resulting matrix of the matrix multiplication (eight

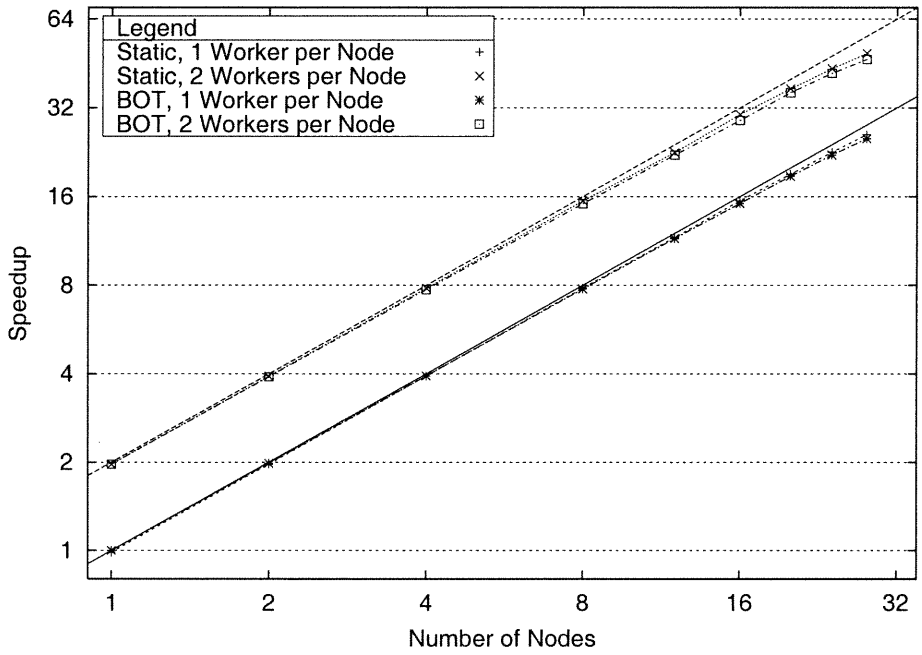

Figure 5.8: Ray Tracing Speedups.

Sequential	1603.68							
	Static				BOT			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1722.4	0.93	856.9	1.87	1622.8	0.99	827.7	1.94
2 nodes	862.7	1.86	431.4	3.72	817.6	1.96	414.8	3.87
4 nodes	432.4	3.71	217.1	7.39	412.9	3.88	209.4	7.66
8 nodes	217.0	7.39	109.0	14.71	208.9	7.68	106.4	15.07
12 nodes	147.3	10.89	74.3	21.58	141.2	11.36	72.9	22.00
16 nodes	109.6	14.63	55.6	28.84	107.7	14.89	55.9	28.67
20 nodes	90.2	17.78	45.9	34.92	87.3	18.37	45.7	35.07
24 nodes	74.6	21.51	39.6	40.51	73.7	21.76	39.0	41.08
28 nodes	64.6	24.81	34.6	46.32	64.7	24.80	34.2	46.85

Table 5.6: Mandelbrot Fractal Execution Times and Speedups.

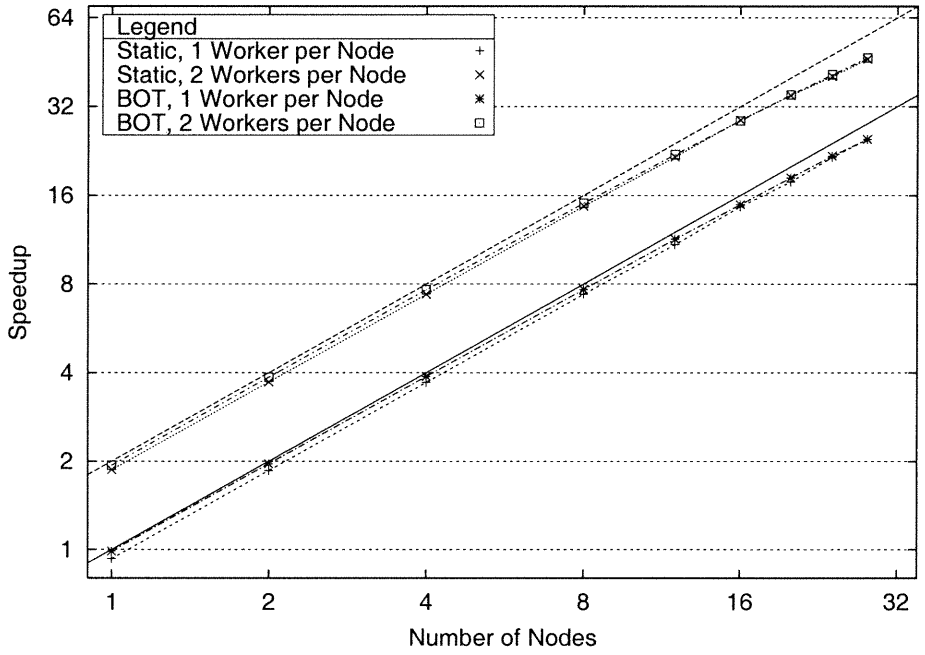

Figure 5.9: Mandelbrot Fractal Speedups.

times smaller), increasing the granularity does not result in great improvements because there are eight times less page-faults. Moreover, using fewer tasks (256 tasks instead of 1024), only improves the execution time by less than 5%. The tables of the execution times for these minimally optimized versions of the benchmarks are given in Appendix D.

5.2.3 Fast Matrix Exponentiation

Moderate speedups are expected for this benchmark as the communication required between each iteration reduces the efficiency of the parallel programs. Even if this is a computationally intensive benchmark, the required communication between each iteration will probably cause performance degradation. The execution times and speedups for this benchmark are presented in Table 5.7, and the graph of the speedups is presented in Figure 5.10.

As predicted, the speedups are moderate. A flattening can be noticed in the static version using two workers per node and in the BOT partitioning version when using one worker per node. Moreover, a performance degradation occurs in the BOT partitioning version when using two workers per node.

Sequential	1142.59							
	Static				BOT			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1123.9	1.02	574.3	1.99	1157.4	0.99	583.9	1.96
2 nodes	581.8	1.96	295.0	3.87	587.2	1.95	301.9	3.78
4 nodes	296.0	3.86	154.6	7.39	300.6	3.80	162.6	7.03
8 nodes	154.1	7.41	85.9	13.30	160.2	7.13	99.2	11.52
12 nodes	107.5	10.63	64.8	17.63	116.7	9.79	85.8	13.32
16 nodes	84.5	13.51	55.3	20.67	95.8	11.93	83.4	13.71
20 nodes	71.2	16.05	50.2	22.78	85.6	13.35	86.3	13.25
24 nodes	62.6	18.26	47.2	24.20	80.8	14.14	94.0	12.15
28 nodes	56.8	20.11	46.0	24.82	78.3	14.58	100.8	11.34

Table 5.7: Fast Matrix Exponentiation Execution Times and Speedups.

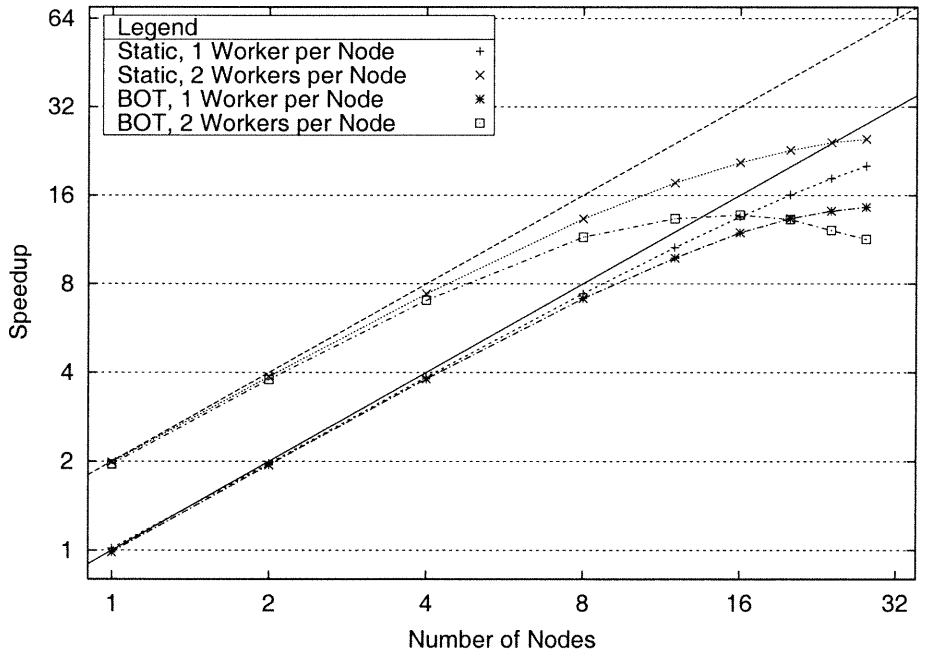

Figure 5.10: Fast Matrix Exponentiation Speedups.

An interesting characteristic of this benchmark is that its implementation needs to send more data at "release" when increasing the number of worker. When doing an iteration of the matrix exponentiation, each worker must communicate its part of the matrix to the other workers. Thus, for N workers, each worker must send $(N-1)/N$ data. For few workers (2 or 4), little data must be sent (1/2 or 3/4 of the total matrix respectively). When using more workers, the quantity of data sent soon becomes very close to the matrix itself.

As the quantity of data sent at each "release" grows asymptotically to the size of the matrix when increasing the number of workers, the amount of data sent when using more than eight workers increases only a little. This means that when increasing the number of workers, the overhead of the release should not increase linearly. This is exactly what is observed when looking at the time required to execute the "releases". For the static versions and one worker per node, this time is about 1.6 second for 2 nodes and about 3.4 seconds for 28 nodes. For two workers per nodes, it is about 2.5 seconds for 2 nodes and about 5.3 seconds for 28 nodes. The overhead is multiply by less than three when multiplying the number of nodes by 14. Therefore, overhead of the "release" cannot explain the flattening and the degradation of the speedups.

For the two static versions, the most important overhead is the data distribution at the beginning of the computation. The data to be distributed is the matrix to be exponentiated and the initial data in the result matrix. As these two matrices' size are 2048x2048 and the size of the items are 4 bytes (floats), the total amount of data to be distributed is 32 megabytes. As the matrix to be exponentiated is in a write-once region, its distribution takes advantage of the request forwarding optimization. However, because request forwarding has not been implemented for the eager release WU region, the home of the region must answer all the mapping requests. This leads to a long data distribution delay. Table 5.8 gives an overview of this delay for the static versions of the benchmarks, and the graph of the speedups without the distribution is presented and compared with the previous results in Figure 5.11.

As shown in Figure 5.11, the data distribution is the main cause of the flattening of the two workers per node static version of the benchmark. Without this data distribution, the results are much better, especially when using two workers per nodes.

To explain the flattening of the BOT version, the difference in the implementation of the two versions must be discussed. In the static version, when computing $A \times A$ (the else part of the algorithm of Figure 5.3), a copy of the matrix is locally taken before reaching a barrier and then

	Static	
	1 Worker	2 Workers
1 node	0	0.44
2 nodes	0.55	1.32
4 nodes	1.17	2.45
8 nodes	2.33	4.55
12 nodes	3.40	6.47
16 nodes	4.32	8.25
20 nodes	5.26	10.10
24 nodes	6.17	11.84
28 nodes	7.11	13.76

Table 5.8: The Data Distribution Delay in the Static Fast Matrix Exponentiation.

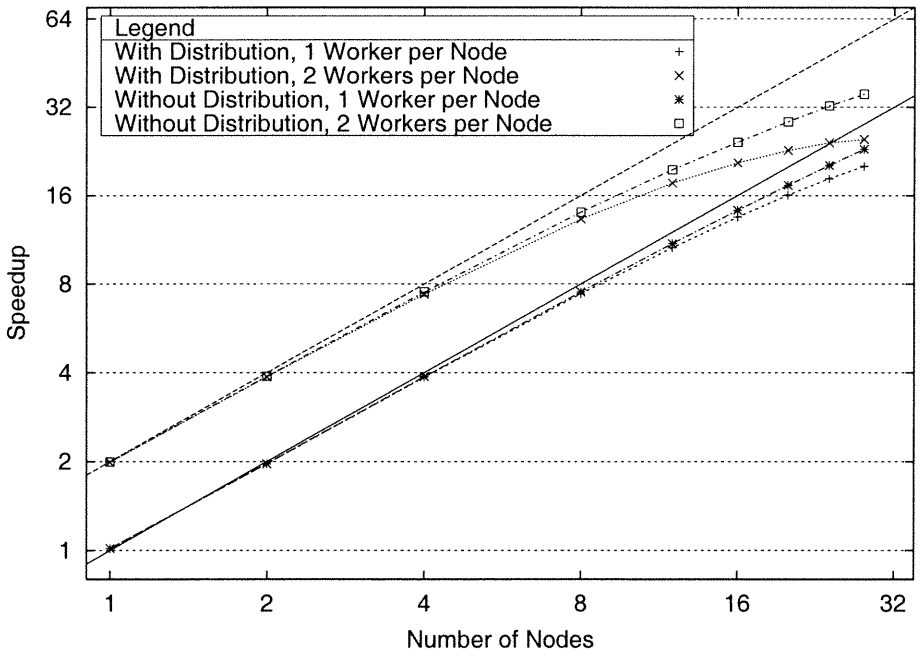

Figure 5.11: Static Fast Matrix Exp. Speedups With and Without the Distribution.

computing $A \times A$. This copy of the matrix is needed in order to remember the content of the matrix before modifying it. In the BOT version, as the workers cannot synchronize themselves using a barrier, a single worker copies the content of the matrix in another shared memory region. This copy introduces network overhead as new data must be sent to all workers. In the case of the exponent 31, this copy occurs three times (more precisely four times, but the first copy does not send data to all workers because the region is not mapped in workers yet). This copy overhead is $3 \times 16 \times (N - 1)$ megabytes where N is the number of workers. In the case of 16 nodes and 2 workers per nodes (a total of 32 workers), this overhead corresponds to 1.488 gigabytes of data, which is huge. As this overhead grows linearly with the number of workers, it eventually overcomes the parallelism gain. No precise measurements were done to evaluate this delay, but it is obviously non negligible. According to some experiments, sending one gigabyte of data via the current version of UDN takes about 15.32 seconds. With this information, an estimation of the copy overhead can be made. Table 5.9 gives these estimations, and the graph of the speedups without the overhead is presented and compared with the previous results in Figure 5.12.

	BOT	
	1 Worker	2 Workers
1 node	0.00	0.72
2 nodes	0.72	2.15
4 nodes	2.15	5.03
8 nodes	5.03	10.77
12 nodes	7.90	16.52
16 nodes	10.77	22.26
20 nodes	13.64	28.01
24 nodes	16.52	33.75
28 nodes	19.39	39.50

Table 5.9: The Estimation of the Copy Overhead in the BOT Fast Matrix Exp..

Moreover, as the initial content of the region used to save a copy of the matrix must be distributed to all workers and that request forwarding is not implemented, the distribution delay is also augmented. This delay is multiply by about two compared with the delay for the static version. Table 5.10 gives an overview of the data distribution delay for the BOT versions of the benchmarks, and the graph of the speedups without the delay is presented and compared with the previous results in Figure 5.13.

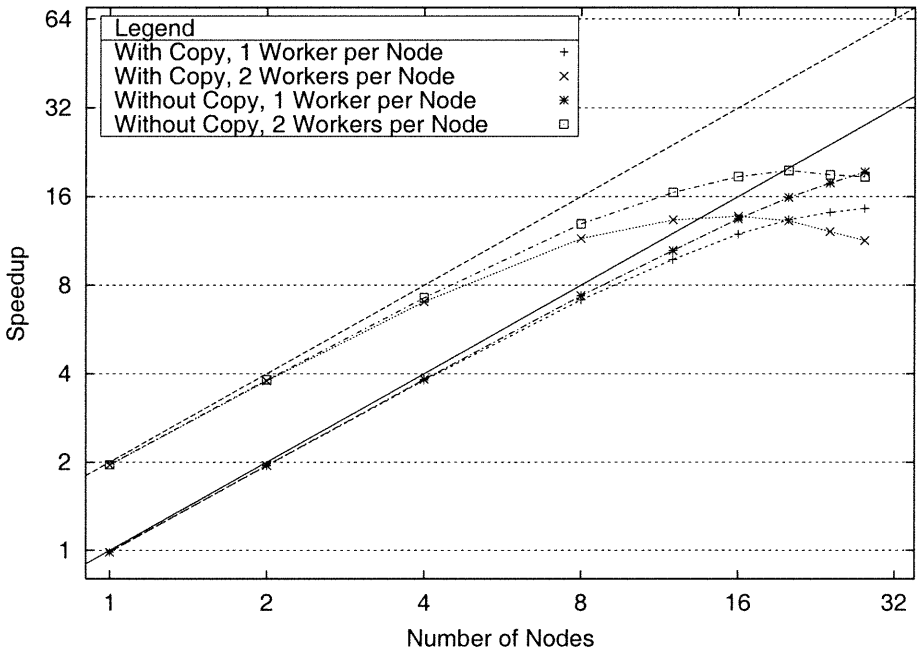

Figure 5.12: BOT Fast Matrix Exp. Speedups With and Without the Copy Overhead.

	BOT	
	1 Worker	2 Workers
1 node	0.00	0.60
2 nodes	0.82	2.07
4 nodes	1.74	3.90
8 nodes	3.53	7.77
12 nodes	5.45	11.44
16 nodes	7.24	15.44
20 nodes	8.98	19.07
24 nodes	10.61	23.68
28 nodes	12.58	27.04

Table 5.10: The Data Distribution Delay in the BOT Fast Matrix Exponentiation.

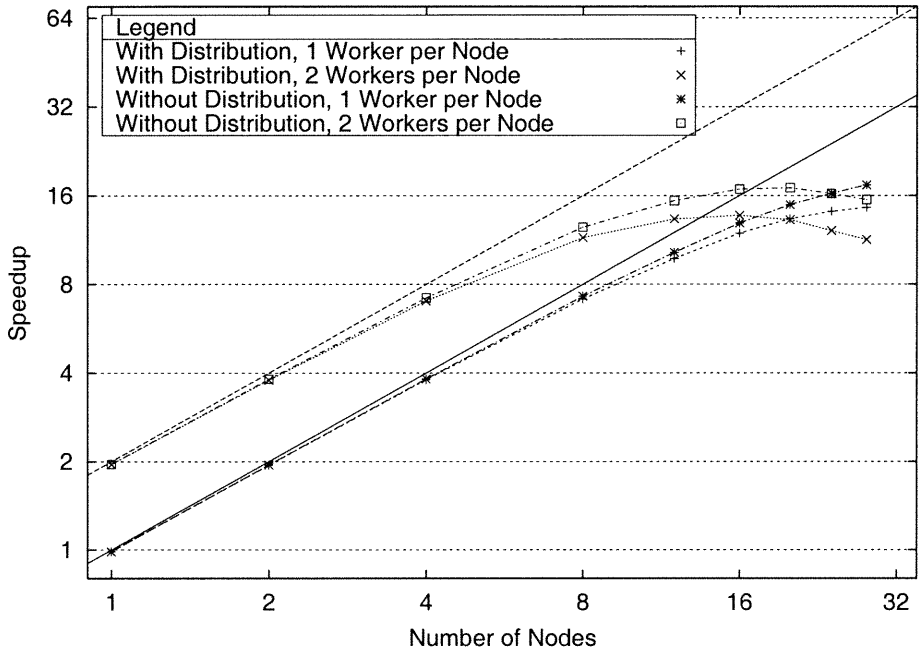

Figure 5.13: BOT Fast Matrix Exp. Speedups With and Without the Distribution.

When removing both overheads, the speedups become acceptable as shown in Figure 5.14. However, even if it is possible to remove the distribution overhead by implementing request forwarding, the copy overhead is hardly removable due to a limitation of the BOT partitioning model. A limitation of the BOT partitioning method has probably been found here.

5.2.4 N-Queens

For the static partitioning version, the expected speedups should be good as the number of solutions is only modified once per worker at the end of the execution of all tasks. For the BOT partitioning version, as the number of solutions must be modified at the completion of each task, the expected speedup should be a little worse than for the static partitioning version because of mutex contention. However, we still expect good speedups for the BOT partitioning version. The execution times and speedups for this benchmark are presented in Table 5.11, and the graph of the speedups is presented in Figure 5.15.

Almost perfect speedups are obtained for both static partitioning versions and for the one worker per node BOT partitioning version. For the BOT partitioning version using two workers

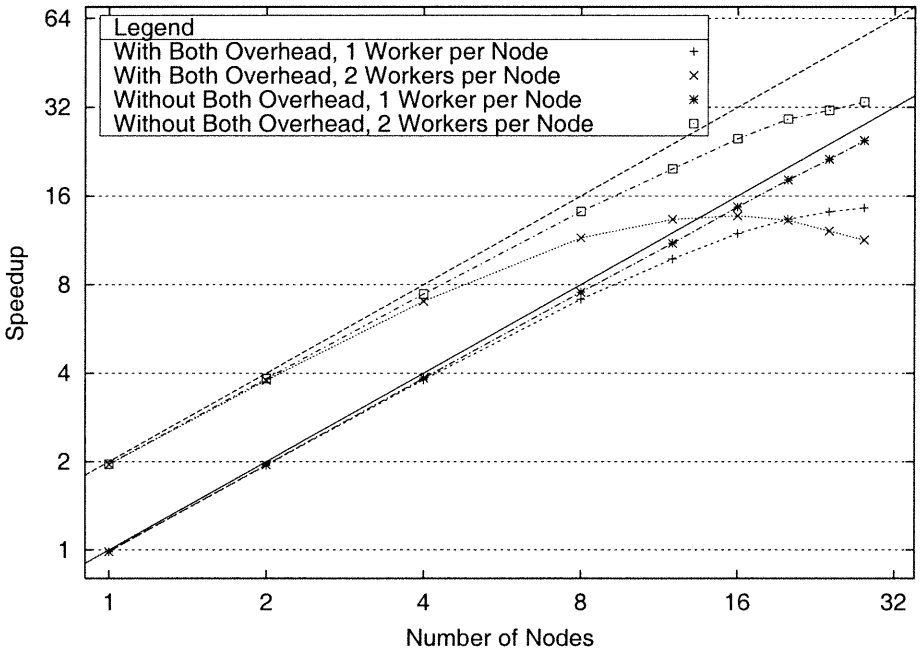

Figure 5.14: Fast Matrix Exp. Speedups With and Without Both Overheads.

Sequential	1280.61							
	Static				BOT			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1280.8	1.00	641.7	2.00	1281.7	1.00	643.0	1.99
2 nodes	640.9	2.00	322.8	3.97	642.4	1.99	326.7	3.92
4 nodes	321.9	3.98	161.4	7.93	321.6	3.98	162.5	7.88
8 nodes	161.1	7.95	81.7	15.67	161.4	7.93	81.6	15.70
12 nodes	107.7	11.89	54.0	23.70	108.0	11.86	55.2	23.20
16 nodes	81.6	15.70	41.2	31.05	80.8	15.85	42.5	30.10
20 nodes	64.9	19.75	33.5	38.21	64.7	19.78	37.8	33.91
24 nodes	53.9	23.74	27.4	46.69	53.9	23.75	34.1	37.60
28 nodes	46.4	27.60	23.5	54.48	46.4	27.60	34.4	37.21

Table 5.11: 17-Queens Execution Times and Speedups.

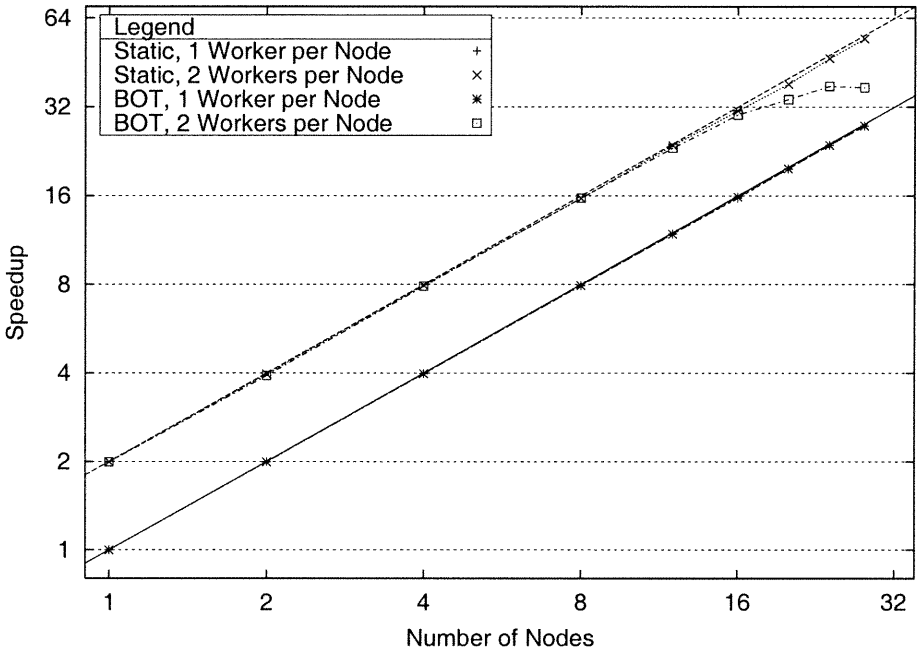

Figure 5.15: 17-Queens Speedups.

per node, the speedups flatten drastically when using more than 16 nodes. This is caused by the mutex contention. As the number of solutions is updated after the completion of each task, reaching this limit could have been expected. This limit is probably not only caused by the mutex implementation. It is probably cause by the "release" being done in the critical section while the mutex is "locked". Because consistency operations are done while holding the mutex, all the diff acknowledgments from all workers must be received before "unlocking" the mutex. To verify these assumptions, two additional versions of the benchmark were derived from the static partitioning version: a version only locking the mutex at the completion of each tasks, and a version where the number of solutions is updated at the completion of each tasks while holding the mutex (as in the BOT partitioning version). The execution times and speedups of these two modified versions are presented in Table 5.12, and the graph of the speedups is presented and compared with the previous results in Figure 5.16.

As shown in Figure 5.16, the flattening does not come from the mutex, but from the consistency operations. To avoid this flattening, the amount of consistency operations should be reduced. To do so, the lazy release model should be used. This model avoids sending diffs to all workers by only sending them to the next locking worker. However, as the lazy release model is

	Static With Lock				Static With Update			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1281.0	1.00	641.5	2.00	1280.9	1.00	642.5	1.99
2 nodes	640.9	2.00	322.8	3.97	641.9	2.00	331.0	3.87
4 nodes	322.7	3.97	161.5	7.93	323.6	3.96	164.7	7.78
8 nodes	161.3	7.94	81.8	15.66	162.2	7.89	82.6	15.51
12 nodes	107.7	11.89	54.1	23.69	108.5	11.81	55.3	23.15
16 nodes	81.7	15.67	41.4	30.96	82.1	15.60	43.1	29.70
20 nodes	64.9	19.74	33.7	38.05	65.3	19.62	39.6	32.33
24 nodes	54.1	23.69	27.5	46.50	54.2	23.61	36.8	34.76
28 nodes	46.7	27.45	23.6	54.20	46.7	27.43	37.1	34.52

Table 5.12: 17-Queens Execution Times and Speedups With Locking and Updating.

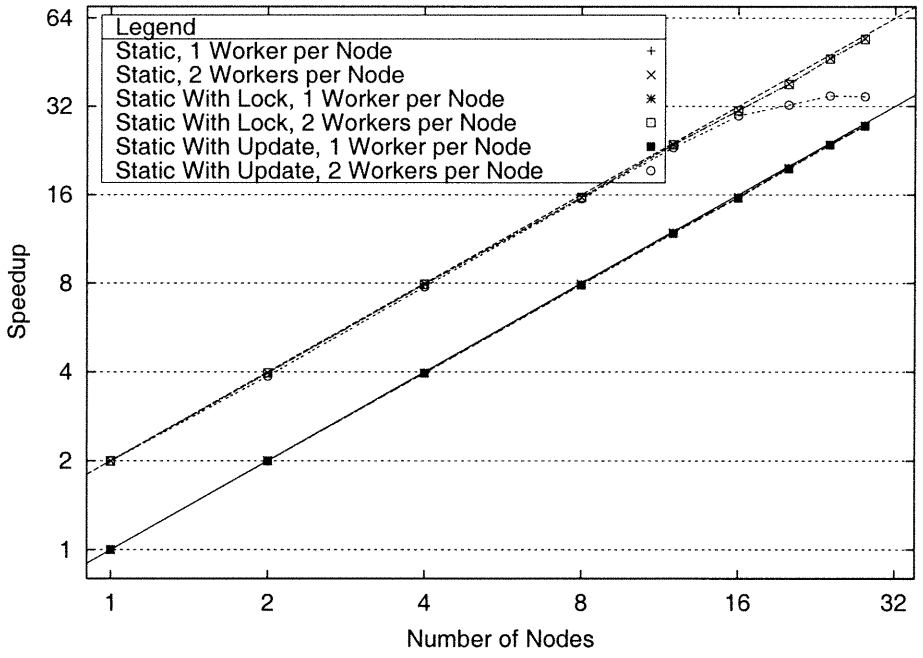

Figure 5.16: 17-Queens Speedups With and Without Locking and Updating.

not yet implemented in YADL, this optimization was not tested.

5.2.5 Tridiag

Poor speedups are expected for this benchmark because it is an application requiring a lot of communication without having computationally intensive iterations. The execution times and speedups for this benchmark are presented in Table 5.13, and the graph of the speedups is presented in Figure 5.17.

Sequential	41.89			
	1 Worker		2 Workers	
	Time	Speedup	Time	Speedup
1 node	50.3	0.83	79.7	0.53
2 nodes	59.1	0.71	47.7	0.88
4 nodes	45.8	0.91	42.7	0.98
8 nodes	38.1	1.10	39.7	1.06
12 nodes	38.2	1.10	41.2	1.02
16 nodes	38.7	1.08	43.3	0.97
20 nodes	39.9	1.05	46.1	0.91
24 nodes	41.2	1.02	48.5	0.86
28 nodes	42.5	0.99	51.1	0.82

Table 5.13: Tridiag Execution Times and Speedups.

As expected, the speedups are very bad. There is practically no speedups. This is caused by a computation requiring little CPU power and too costly consistency operations. Increasing the granularity of the shared memory could be a solution to obtain better speedups. This would reduce the number of page-faults incurred by the write invalidate implementation of the shared memory. The granularity has been increased from 4 kilobytes to 64 kilobytes to observe the impact of less requests on the performance of the application. The execution times and speedups with the increased granularity are presented in Table 5.14, and the graph of the speedups is presented and compared with the previous results in Figure 5.18.

As shown in Figure 5.18, increasing the granularity improves a little the performance of the application. However, the speedups are still almost inexistent.

To explain the absence of speedups, three elements must be considered. The first is the delay for serving page-fault requests. This delay and the sum of the data sent by the home for

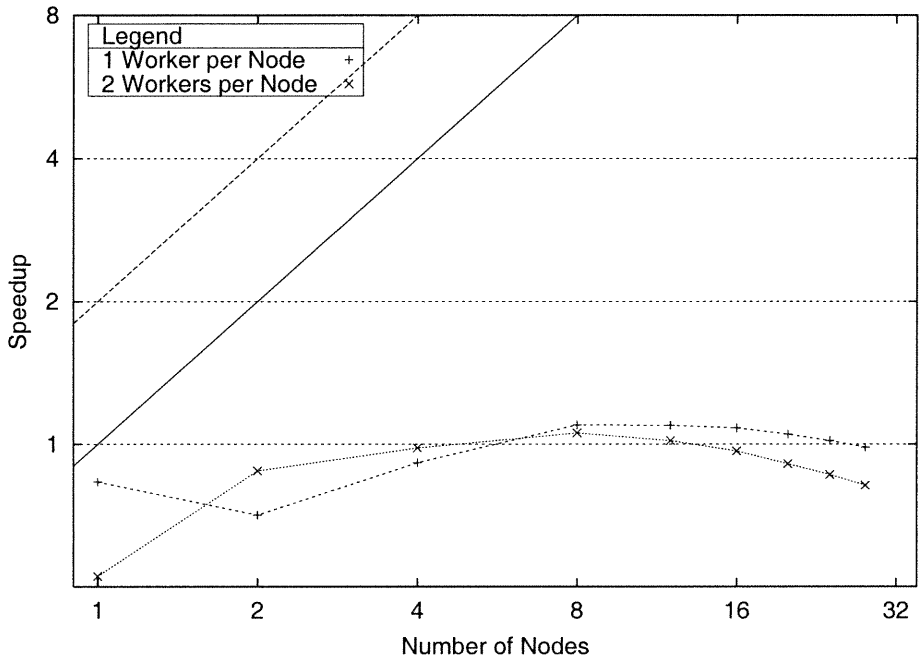

Figure 5.17: Tridiag Speedups.

	Without Increased Granularity				With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	50.3	0.83	79.7	0.53	46.9	0.89	48.5	0.86
2 nodes	59.1	0.71	47.7	0.88	38.7	1.08	36.3	1.16
4 nodes	45.8	0.91	42.7	0.98	29.0	1.45	28.4	1.48
8 nodes	38.1	1.10	39.7	1.06	26.2	1.60	27.4	1.53
12 nodes	38.2	1.10	41.2	1.02	28.0	1.50	30.4	1.38
16 nodes	38.7	1.08	43.3	0.97	27.7	1.51	30.0	1.40
20 nodes	39.9	1.05	46.1	0.91	30.5	1.38	35.6	1.18
24 nodes	41.2	1.02	48.5	0.86	31.4	1.33	37.0	1.13
28 nodes	42.5	0.99	51.1	0.82	32.9	1.27	40.6	1.03

Table 5.14: Tridiag Execution Times and Speedups With Increased Granularity.

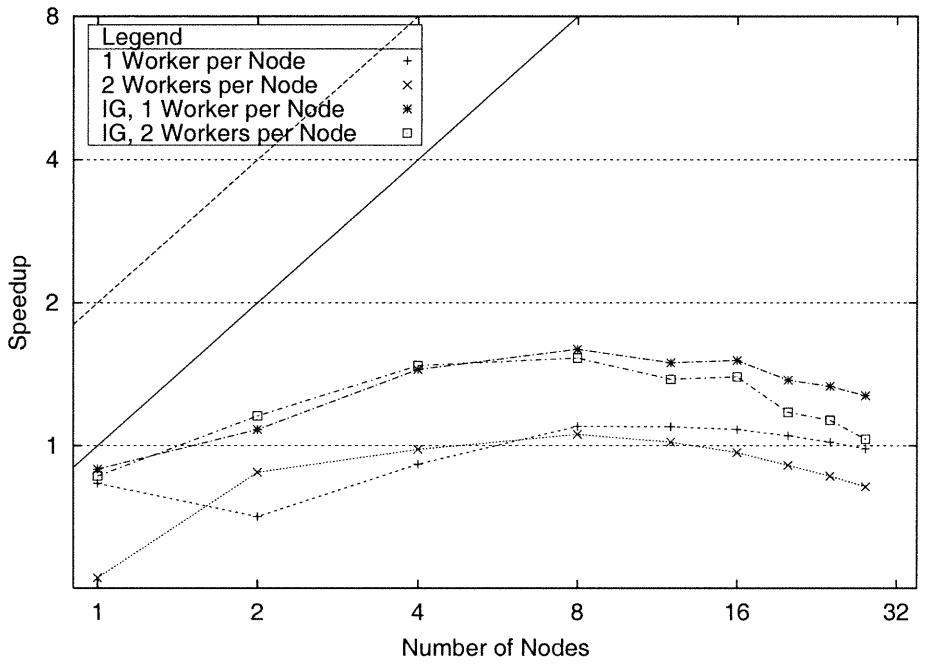

Figure 5.18: Tridiag Speedups With and Without Increased Granularity (IG).

serving page-fault requests for the increased granularity version of the benchmark are presented in Table 5.15. This table shows that these delays (and the corresponding data sent) increase with the number of workers. When comparing these delays with the execution times presented in Table 5.14, the large proportion of the time spent in answering page-fault requests can be noticed. Another element to explain the absence of speedups is the time spent in "releases". Table 5.16 presents the delay for doing "releases" in the increased granularity version of the benchmark. These delays are not as large as the delays for serving data requests, but they are not negligible. Finally, the last element is that some workers receive their data later than others. As these workers begin their iteration later, they will also reach the barrier later. As there are 22 iterations in this version of the benchmark, the difference in the reaching time at each barrier is multiplied by 22, transforming a small delay in a large one. No exact measurement of this delay has been performed, but, as the time spent answering requests is large, this delay is likely to be not negligible.

There is still hope of increasing the performance of this application. One of the reasons explaining the bad speedups is that a single node answers all the data requests from the other nodes (there is a single home for all the shared memory). Using a more decentralized imple-

	1 Worker		2 Workers	
	Delay	Data	Delay	Data
1 node	0.00	0	1.43	125
2 nodes	1.26	125	3.34	272
4 nodes	3.26	272	5.77	409
8 nodes	5.94	409	8.92	548
12 nodes	9.38	544	12.73	765
16 nodes	8.85	548	12.10	710
20 nodes	12.25	726	17.19	1,051
24 nodes	12.91	765	17.85	1,104
28 nodes	14.31	874	20.61	1,306

Table 5.15: Delay for Serving Page-Fault Requests in Tridiag.
(Delay in seconds, data in megabytes)

	1 Worker	2 Workers
1 node	5.96	6.77
2 nodes	5.85	6.25
4 nodes	5.79	6.61
8 nodes	6.70	6.77
12 nodes	7.45	7.50
16 nodes	8.05	8.21
20 nodes	8.59	9.29
24 nodes	9.29	10.00
28 nodes	9.29	11.01

Table 5.16: Delay for Executing Releases in Tridiag.
(Delay in seconds)

mentation of the eager release WI model, as the single-writer version presented in Section 3.3.3, could lead to performance improvements by reducing the delay needed to answer page-fault requests. As the answering of page-fault requests would be accelerated, the difference in the barrier reaching time would be reduced, increasing even more the performance of the benchmark. Finally, by using a more decentralized region implementation, "release" operations would also be more efficient as they would stress less a single worker, the home of the region.

When using two workers per node, we generally have worse results than when using a single worker per node. This is explained by the computing power added not being backed by more network resources. As this benchmark uses the network intensively for importing data, only adding workers sharing the same network resources does not lead to performance improvements.

5.3 Discussion

This section discusses the benchmark results presented in the previous section. First, general conclusions are drawn from the results. Then, a comparison of the static and the BOT partitioning is presented followed by a discussion on the ability of tuning the execution parameters. Finally, future development on YADL drawn from these discussions is presented.

5.3.1 General Conclusions

Generally, the benchmark results are satisfying. They show that YADL can be used to implement parallel applications achieving moderate to good speedups for some problems. The applications that need very little collaboration, such as the matrix multiplication, the ray tracing, the Mandelbrot fractal and the n-queens applications, perform very well using YADL.

When problems require more collaboration between the workers, the speedups are inferior. The tridiag benchmark proves that some applications do not perform well with the current implementation of YADL. However, implementing more specific region types could lead to improvements of the tridiag benchmark. Nevertheless, the performance of tridiag will never be as good as the performance of very well performing applications.

The performance of the matrix exponentiation benchmark is better than the performance of tridiag, but, when using many nodes, the performance begins to flatten. The analysis of

the execution time of this benchmark shows that the flattening is mainly caused by the request forwarding not being implemented for the eager release WU regions. This leads to the conclusion that YADL must still be optimized to achieve better speedups for this application.

5.3.2 Static vs BOT Partitioning

The static partitioning almost always gives better results than the BOT partitioning. This could have been expected as the BOT partitioning introduces overhead. However, the ray tracing, the Mandelbrot fractal and the n-queens benchmarks are three successes of the BOT partitioning: the results obtained are very close to the results with the static partitioning. These good results are mainly explained by the computationally intensive characteristic of these three benchmarks associated with a requirement for little memory, thus not requiring lots of consistency operations.

For the matrix multiplication benchmark, the performance of the first BOT partitioning version of the program was unsatisfying. However, increasing the granularity has improved greatly the performance of the program. Moreover, using less tasks in the BOT also results in an improvement of the performance of this program. These positive results of the optimization of the matrix multiplication benchmark shows that the BOT partitioning could obtain good results in a computationally intensive application that needs lot of shared memory.

For the matrix exponentiation benchmark, the performances of the BOT partitioning version is disappointing. Because it requires too many consistency operations, including the need for more shared memory, the BOT partitioning obtains worse speedups than the static partitioning.

However, achieving better speedups and better load balancing were not the only goal motivating the introduction of the BOT partitioning. Achieving load balancing with different processor speeds and worker addition and removal are features only provided by the BOT partitioning. If a programmer requires these features, he may accept the performance penalty associated with the use of the BOT partitioning.

5.3.3 Tuning Parameters

In Section 5.2, some benchmark results have been presented where some execution parameters have been tuned to try to increase their performances. Tuning correctly the granularity of the

regions has resulted in performance improvements for the static version of the tridiag benchmark and for the BOT version of the matrix multiplication benchmark. The ability to tune the granularity is a feature that must be provided to the advanced programmers to allow them to get as much performance as possible from the SDSM system.

5.3.4 Future Development on YADL

YADL is functional, but some work must still be done to achieve better performance and to obtain a more user-friendly interface. The improvements that could be done are: implementing more region types, implementing more synchronization primitive types, supporting many workers for a single manager, implementing more efficient versions of UDN, supporting task prefetching, supporting worker removal when using the BOT partitioning, increasing portability, and doing more extensive benchmarking.

Implementing More Region Types

More region types, making available more memory consistency modes, are needed to cover all possible memory access patterns. Among unimplemented models, the sequential and lazy release models should be implemented. Moreover, single-writer versions of the eager release model should be implemented to provide the programmer with lower overhead implementation for the cases where a multiple-writers version is not required (the case of the tridiag benchmark).

Also, the current implementation of the regions could be optimized. A primordial optimization is request forwarding in the eager release WU regions as shown by the results of the matrix exponentiation benchmark.

Implementing More Synchronization Primitive Types

Many synchronization primitive types should be added to the system. Among them, barriers, including tree barriers, semaphores and condition variables should be implemented. Some other mutex types should also be implemented to increase the performance of some mutex-based applications by enforcing mutex locality (see Section 4.2.4).

Moreover, the combination of synchronization primitives with "acquire" and "release" should

be done to provide a more user-friendly API.

Supporting Many Workers for a Single Manager

As presented in Section 3.5.1, some optimizations could be done when using multiprocessor nodes. One of these optimizations is to allow many workers to be spawned using the same manager. Implementing this feature should be done to take advantage of multiprocessor nodes. However, to implement this optimization, some modifications must be done to the implementation algorithms of regions and synchronization primitives.

Supporting many workers for a single manager could also be an optimization for uni-processor NOWs. By having many workers on a single node, CPU cycles are not wasted when a worker is blocked, as they can by used to run other workers. A study of the effect of this optimization for uni-processor NOWs would be interesting.

Implementing More Efficient Versions of UDN

The current implementation of UDN is a naive implementation over TCP/IP (see Section 4.2.2). Implementing UDN over some lower overhead or higher performance network layers could result in performance improvements. Among these network layers, UDP/IP, Ethernet and Myrinet should be the next targets. Moreover, a port of UDN over MPI could also be advantageous to use the hardware implementation of MPI provided by some network adapter.

Supporting BOT Task Prefetching

When using too small tasks with the BOT partitioning, the latency of task fetching could be a noticeable overhead. To reduce this overhead, a worker could prefetch some tasks. With task prefetching, the task commitment and fetching could be done while executing another task, thus avoiding the lost of CPU cycles.

Supporting Worker Removal in BOT Mode

The worker addition using the BOT partitioning is almost implemented because the shared memory and synchronization primitive algorithms have been designed to support this feature. Only small modifications to the server are needed to complete the implementation of worker addition. However, graceful removal (not a node failure) is not implemented. The removal of workers with the BOT partitioning would be a nice feature to implement as it would allow to remove workers from nodes that must be shut down for maintenance or that have been requested back by their owners[5]. Implementing this feature requires a lot of work to adapt region and synchronization primitive implementations.

Increasing Portability

YADL was not designed with portability in mind. However, because most system functions used are standard UNIX functions, YADL is probably compatible with other UNIX-flavor OSes. Work should be done identifying sections of code that are not compatible and modifying them to achieve portability.

Doing More Extensive Benchmarks

More extensive benchmarks should be done to identify the most valuable optimizations and to get more intuition about the tuning of execution parameters such as the number of tasks to put in the BOT and the granularity of regions. Moreover, other benchmark applications should be developed to stress different parts of the system and better evaluate the performance of YADL.

[5]Some NOWs are composed of nodes used occasionally by their owners and lent to perform parallel computation.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In Section 1.1, we indicated that our goal was to design and implement a SDSM system that could be used on the Linux OS. This goal has been achieved. We have proposed a functional SDSM system, named YADL, that can be used to develop parallel applications on NOW.

The major contribution of this work is to increase the availability of SDSM systems. As mentioned in Section 1.1, the reason why programmers are not using SDSM systems could be the ignorance of their existence owing to their low availability. By providing this system, we hope that programmers will use it and will discover the effectiveness of programming with SDSM.

Another contribution of this work is to introduce the BOT partitioning to SDSM. The idea of using the BOT partitioning is not new, but, to our knowledge, the way we include it in a SDSM system is new. The BOT partitioning allows to easily achieve load balancing in irregular applications and on NOWs composed of different processor speed nodes. It also theorically allows to add and remove workers during a computation. Moreover, as we will see in Section 6.2, the BOT partitioning could be a first step to fault tolerant SDSM systems.

Our concern about expert programmers has not been addressed in previous SDSM systems. Allowing experts to tune the performance of their application by specifying execution parameters such as the granularity of the regions and the diff unit of the release memory consistency models

is not a feature that we found while studying previous SDSM systems. In Section 5.2, we pointed out that increasing the granularity improves the performance of some benchmarks, which is a good optimization. We should eventually add some other advanced features to allow expert programmers to optimize even more their programs, like user-driven prefetching.

The implementation of YADL is also a starting point for more extensive research on SDSM systems. Before its implementation, no SDSM system was available at University of Montreal to conduct research. With YADL, we can launch more experimental research on SDSM systems.

We think that YADL is not simply a research prototype, but also a complete SDSM system that can be used to develop parallel applications. YADL is actually very suited for computationally intensive applications not requiring a lot of collaboration between workers. It could probably be used in the computer graphics field to perform parallel image synthesis. Moreover, we plan to adapt the system to train neuron network, a machine learning algorithm used in the artificial intelligence field that requires a lot of computing power.

A comparison of YADL and other SDSM systems was not performed due to availability restrictions. One year before publishing this document, a demonstration version of the commercial TreadMarks SDSM system was obtained and briefly evaluated using a different benchmark platform from the one used for the benchmarks presented in Chapter 5. The benchmark programs for this evaluation were matrix multiplication and ray tracing. The maximum number of nodes used was eight due to a limitation in the demonstration version of the TreadMarks library. The speedups obtained were very similar to the one published in Chapter 5. Of course, YADL is not a system as mature as TreadMarks, but we think that YADL can be considered as an alternative to TreadMarks for some applications. When the lazy release memory consistency model will be implemented, YADL will offer a true alternative to TreadMarks.

Of course, we do not expect YADL to replace neither message passing nor multiprocessors. As mentioned in Section 3.6, SDSMs' niche is to produce quickly an elegant parallel solution that can be run on NOWs when a fast and inexpensive solution is required. People needing high performance computing will still spend much effort on the message passing model and will still spend much money on multiprocessors to get the fastest solution available.

With YADL, we do not expect to convert programmers needing high performance computing to use SDSMs. Our target is programmers with limited financial resources and development deadline needing a parallel solution to their problem. These programmers could eventually start

using SDSM to obtain elegant parallel programs that can be run on NOWs. However, the low availability of SDSM libraries was an obstacle to their usage. Now, with the introduction of YADL, this obstacle will hopefully be removed and programmers will start to use SDSM.

6.2 Future Work

Future research should be mainly oriented on the BOT partitioning. One of the needed optimizations to the BOT partitioning is a distributed bag of tasks management to remove the centralized implementation of the BOT in the server.

One of the major drawbacks of the BOT partitioning is the explosion of the number of consistency operations. Performing consistency operations while executing the next task could result in improvements. By implementing more aggressive versions of memory consistency models, a worker could start a new task while the consistency operations related to the previous task are done in the background, reducing the idle time of processors. Combining aggressive implementations of memory consistency models and the BOT partitioning is a promising avenue of research to improve its performance.

Work on the implementation of the lazy release memory consistency model into the BOT partitioning should also be done. In mutex based applications, the BOT partitioning should work very well with the lazy release model. However, as in barrier based applications, "acquire" and "release" operations are merged with barriers. As barriers using the BOT partitioning are implicit (they are managed in the BOT by task substitutions and dependencies), no explicit point in a BOT partitioning program provides a way to do "acquires" and "releases" like explicit barriers do in a static partitioning program. A special task could be added in the BOT to do a barrier with consistency operations, but we think that a more elegant solution to this problem can be found.

Moreover, guidelines about the BOT partitioning must also be developed. Correctly choosing the number of tasks to put in the BOT is one of the important elements that should be clarified. Moreover, techniques using adequately task substitution to dynamically partition the computation according to run-time parameters should be developed.

Finally, the BOT partitioning could be used as a means to develop a fault-tolerant task

partitioning system. Fault tolerance implies that both the computation organization and the data are protected against failures. The data fault tolerance can be achieved by replication, but a fault-tolerant computation organization is more difficult to achieve. The BOT partitioning could be used as a mean to achieve a fault-tolerant computation organization. To do so, a task would be considered as a transaction that, on completion, cannot be lost. On a task commit, the data modified while executing the task would be replicated, thus achieving fault tolerance. However, the development of a fault-tolerant BOT and fault-tolerant data storage must still be carried out.

Bibliography

[ACG86] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[Car94] John B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency.* PhD thesis, Rice University, January 1994.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of MUNIN. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[Cd+99] Alan L. Cox, Eyal de Lara, , Charlie Hu, and Willy Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols for software shared memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 279–283, January 1999.

[CG90] Nicholas Carriero and David Gelernter. *How to Write Parallel Programs: A First Course.* MIT Press, 1990.

[CKK95] John B. Carter, Dilip Khandekar, and Linus Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings of the 5th Workshop on Hot Topics in Operating Systems*, pages 119–123, May 1995.

[Cor02] Intel Corporation. *A-32 Intel Architecture Software Developers Manual Volume 3: System Programming Guide*, chapter 5, pages 44–46. Intel Corporation, 2002. http://developer.intel.com/design/pentium4/manuals/245472.htm, Visited on 2002-08-13.

[FMST00] Brett D. Fleisch, Heiko Michel, Sachin K. Shah, and Oliver E. Theel. Fault tolerance and configurability in DSM coherence protocols. *IEEE Concurrency*, 8(2):10–21, April-June 2000.

[FP89] Brett D. Fleisch and Gerald J. Popek. Mirage: a coherent distributed shared memory design. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 211–223, December 1989.

[FvDFH96] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addisson-Wesley, second edition, 1996.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[Goo91] James R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, University of Wisconsin-Madison, February 1991.

[GYF95] James Griffioen, Rajendra Yavatkar, and Raphael Finkel. Unify: A scalable approach to multicomputer design. *IEEE Computer Society Bulletin of the Technical Committee on Operating Systems and Application Environments*, 7(2), July 1995.

[Hal96] Fred Halsall. *Data Communication, Computer Networks and Open Systems.* Addison-Wesley, fourth edition, 1996.

[HP96] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufmann, second edition, 1996.

[HST99] Weiwu Hu, Weisong Shi, and Zhimin Tang. Home migration in home-based software DSMs. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, June 1999.

[Hu99] Weiwu Hu. Reducing message overhead in home-based software DSMs. In *Proceedings of the 1st Workshop on Software Distributed Shared Memory*, June 1999.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 213–228, December 1995.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[KCG+95] Anne-Marie Kermarrec, Gilbert Cabillic, Alain Gefflaut, Christine Morin, and Isabelle Puaut. A recoverable distributed shared memory integrating coherence and recoverability. In *Proceedings of the 25th International Symposium on Fault Tolerant Computer Systems*, pages 289–298, June 1995.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kel95] Peter Keleher. *Lazy Release Consistency for Distributed Shared Memory.* PhD thesis, Rice University, January 1995.

[Kel96a] Pete Keleher. *CVM: The Coherent Virtual Machine*, November 1996.

[Kel96b] Peter J. Keleher. The relative importance of concurrent writers and weak consistency models. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 91–99, May 1996.

[Kha] Dilip Khandekar. Quarks: Portable distributed shared memory on UNIX. Included in Quarks distribution: http://www.cs.utah.edu/flux/quarks.html, Visited on 2002-08-13.

[Kha95] Dilip Khandekar. *Application Programming with Quarks*, 1995.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[Li88] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[lin] Scientific computing associates. http://www.lindaspaces.com/, Visited on 2002-08-13.

[man] Fractal eXtreme: Fractal theory. http://www.cygnus-software.com/theory/theory.htm, Visited on 2002-08-13.

[Mat88] Friedemann Mattern. Virtual time and global states of distributed systems. pages 215–226, October 1988.

[MKB97] Christine Morin, Anne-Marie Kermarrec, and Michel Banâtre. An efficient and scalable approach for implementing fault tolerant dsm architectures. Technical Report 3103, Institut National de Recherche en Informatique et en Automatique (INRIA), February 1997.

[mpi] Message passing interface (MPI) forum home page. http://www.mpi-forum.org/, Visited on 2002-08-13.

[Mue97] Frank Mueller. On the design and implementation of dsm-threads. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 315–324, June 1997.

[Mue00] Frank Mueller. Decentralized synchronization for multi-threaded DSMs. In *Proceedings of the 2nd Workshop on Software Distributed Shared Memory*, May 2000.

[myr] Myricom home page. http://www.myri.com/, Visited on 2002-08-13.

[NTA96] Mohamed Naimi, Michel Trehel, and Andri Arnold. A log(N) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34(1):1–13, April 1996.

[PCD+00] Eduardo Pinheiro, DeQing Chen, Sandhya Dwarkadas, Srinivasan Parthasarathy, and Michael L. Scott. S-DSM for heterogeneous machine architecture. In *Proceedings of the 2nd Workshop on Software Distributed Shared Memory*, May 2000.

[RDF+00] Michael D. Rogers, Christopher Diaz, Raphael Finkel, James Griffioen, and James E. Lumpp Jr. BTMD: Small, fast diffs for WAN-based DSM. In *Proceedings of the 2nd Workshop on Software Distributed Shared Memory*, May 2000.

[Sco00] Michael L. Scott. Is S-DSM dead? Invited Keynote Address: 2nd Workshop on Software Distributed Shared Memory, May 2000.

[SD86] Christopher Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June 1986.

[SGG02] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating Systems Concepts.* John Wiley & Sons, sixth edition, 2002.

[sin] The UNIX system. http://www.unix-systems.org/, Visited on 2002-08-13.

[Sta97] William Stallings. *Operating Systems: Internals and Design Principles.* Prentice Hall, third edition, 1997.

[Ste94] Richard W. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, first edition, 1994.

[Ste97] Richard W. Stevens. *UNIX Network Progrmaming, Volume 1: Networking API: Sockets and XTI.* Prentice-Hall, second edition, 1997.

[Ste98] Richard W. Stevens. *UNIX Network Progrmaming, Volume 2: Interprocess Communications.* Prentice-Hall, second edition, 1998.

[TMK96] *Concurrent Programming with TreadMarks*, 1996.

[tri] Data-parallel algorithms: Recursive doubling techniques. http://www.qmw.ac.uk/~cgaa260/BUILDING/DATAPARA/RECURDBL/TRIDIAG.HTM, Visited on 2002-08-13.

[WLF01] David Watson, Yan Luo, and Brett D. Fleisch. Experiences with oasis+: A fault tolerant storage system. In *Proceedings of the IEEE International Conference on Cluster Computing*, pages 29–36, October 2001.

[YLLM01] Hee-Chul Yun, Sang-Kwon Lee, Joonwon Lee, and Seungryoul Maeng. An efficient lock protocol for home-based lazy release consistency. In *Proceeding of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 527–532, May 2001.

[ZIL96] Yuanyuan Zhou, Liviu Iftode, and Kai Li. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, pages 75–88, October 1996.

Appendix A

Page-Faults at User Level

The possibility of receiving VM page-faults at user level is important to the implementation of a SDSM system relying on the VMH to detect memory accesses. The page-faults are trapped in the OS kernel in the VM module, which is not accessible to user processes.

Fortunately, with UNIX, it is possible to protect memory and receive the corresponding page-faults in a user program. To protect memory, the UNIX function `mprotect` must be used. `mprotect` allows the protection of memory at a granularity of VM pages by allowing none, read-only, write-only, or read-write permissions[1]. If an invalid access is performed, the program receives a `SIGSEGV` signal. The default handler for this signal usually halts the program execution and dumps a "core" file in the working directory of the program.

The default `SIGSEGV` handler can be replaced by a custom handler to manage the page-faults. When the execution of the handler is completed, the faulting memory access is replayed. If the permissions on memory are changed while executing the handler, the access could perform correctly and the program execution can resume. If the permissions are not changed, the handler will be called again, causing a potential infinite loop.

To register a custom handler to the `SIGSEGV` signal, the UNIX function `sigaction` can be used. To get the address that causes the access fault, the three arguments signal handler should be used. When using this handler, the second argument is a `siginfo_t` structure containing a

[1] It is also possible to allow or restrict execution permission on the memory, but this is not very relevant to SDSMs.

`void *si_addr` field which points to the memory location causing the fault (see the `sigaction` documentation for more details). However, this feature is not yet implemented on Linux. To get the faulting address under Linux, we must rely on an undocumented feature.

Under x86 Linux[2], adding a `struct sigcontext` parameter to the one argument signal handler allows to get information about the context of a signal. One of the fields of this structure is `unsigned long cr2` which corresponds to the faulting address. To know if the fault is a read or a write fault, the second bit of the field `unsigned long err` of the signal context structure must be consulted: if the bit is set, the fault is a write fault, if not, it is a read fault. For more informations about the Intel page-fault exception, the reader is referred to [Cor02]. An example of code to obtain a faulting address and the type of fault follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/mman.h>

#define PAGE_SIZE 4096
#define IsWriteFault(SCP) (SCP.err & 2)

void handler(int nSignal, struct sigcontext SCP) {
  printf("--In Handler--\n");
  printf("  Faulting address: %p\n", (void*)SCP.cr2);
  printf("  Type of fault: %s.\n", (IsWriteFault(SCP) ? "Write" : "Read"));

  if (IsWriteFault(SCP)) mprotect((void*)SCP.cr2, PAGE_SIZE, PROT_READ | PROT_WRITE);
  else mprotect((void*)SCP.cr2, PAGE_SIZE, PROT_READ);
  printf("--Out Handler--\n");
}

int main(int argc, char *argv[]) {
  char *str;

  signal(SIGSEGV, handler);

  str = malloc(2*PAGE_SIZE);
  str = (char*)(((int) str + PAGE_SIZE-1) & ~(PAGE_SIZE-1));

  *str = 'a';

  mprotect(str, PAGE_SIZE, PROT_NONE);

  printf("--Try to read, should fault--\n");
  printf("The Value is: %c.\n", *str);

  printf("--Try to write, should fault--\n");
  *str = 'b';

  printf("--Try to read, should NOT fault--\n");
  printf("The Value is: %c.\n", *str);
```

[2]This feature is probably not compatible with other computer architecture Linux versions and with other UNIX-flavor OSes.

```
  return 0;
}

/* Execution result:
   --Try to read, should fault--
   --In Handler--
   Faulting address: 0x804a000
   Type of fault: Read.
   --Out Handler--
   The Value is: a.
   --Try to write, should fault--
   --In Handler--
   Faulting address: 0x804a000
   Type of fault: Write.
   --Out Handler--
   --Try to read, should NOT fault--
   The Value is: b.
*/
```

Appendix B

Some YADL Programs

This appendix presents some YADL programs. The emphasis was not placed on performance, but on ease of development when using a SDSM system. The five programs presented are the matrix multiplication, the Mandelbrot fractal, the fast matrix exponentiation, n-queens and tridiag. These codes are not the exact benchmark codes that were used to produce the results presented in Chapter 5, but simplified version of these benchmarks.

The Matrix Multiplication Program

This section presents a matrix multiplication program using the static partitioning. The two matrices to be multiplied are identity matrices multiplied by two. The resulting matrix should be an identity matrix multiplied by four, which is verified by the program. The region types used are write-once VMH home for the matrices to be multiplied and eager release VMH home write-invalidate for the resulting matrix. This program can be run on two nodes using the command line: `"MatMult -s 128 -- -h node0 -h node1"`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "DSM.h"

int main(int argc, char *argv[]) {
  float *pfA, *pfB, *pfC;
  int c, nSize, id0, id1, nPerW, nLeft, ns, ne, i, j, k;
```

```
DSMRegionData_t RD;

nSize = 128;
while ((c = getopt(argc, argv, "s:")) != -1)
  switch(c) {
    case 's':
      nSize = atoi(optarg); break;
  }

DSM_Start(argc, argv);

if (!DSM_GetWorkerIndex()) {
  RD.nRegionNumber = 1;
  RD.Size          = nSize * nSize * sizeof(float);
  RD.Type          = DSM_REGION_WOVH;
  RD.Granularity   = DSM_REGION_GRAN_4096;
  DSM_CreateRegion(&RD);

  RD.nRegionNumber = 2;
  DSM_CreateRegion(&RD);

  RD.nRegionNumber = 3;
  RD.Type          = DSM_REGION_ERVHWI;
  RD.SpecificData.ERVHWIData.nDiffUnit = sizeof(float);
  DSM_CreateRegion(&RD);

  DSM_MapRegion(1, (void**)&pfA);
  DSM_MapRegion(2, (void**)&pfB);

  for(i = 0; i < nSize * nSize; i++) pfA[i] = pfB[i] = 0;
  for(i = 0; i < nSize * nSize; i += nSize+1) pfA[i] = pfB[i] = 2;

  DSM_UnMapRegion(2);
  DSM_UnMapRegion(1);
}

DSM_WaitInit(1);

id0   = DSM_GetWorkerIndex();
id1   = id0 + 1;
nPerW = nSize / DSM_GetNumberOfWorkers();
nLeft = nSize % DSM_GetNumberOfWorkers();
ns    = id0 * nPerW + id0 * nLeft / DSM_GetNumberOfWorkers();
ne    = id1 * nPerW + id1 * nLeft / DSM_GetNumberOfWorkers();

DSM_MapRegion(1, (void**)&pfA);
DSM_MapRegion(2, (void**)&pfB);
DSM_MapRegion(3, (void**)&pfC);

for(i = ns; i < ne; i++)
  for(j = 0; j < nSize; j++) {
    pfC[i * nSize + j] = 0;
    for(k = 0; k < nSize; k++)
      pfC[i*nSize + j] += pfA[i*nSize + k] * pfB[j*nSize + k];
  }

DSM_Release();

DSM_UnMapRegion(3);
DSM_UnMapRegion(2);
DSM_UnMapRegion(1);
```

```
  DSM_WaitInit(2);

  if (!DSM_GetWorkerIndex()) {
    DSM_MapRegion(3, (void**)&pfC);

    for(i = 0; i < nSize - 1; i++) {
      if (pfC[0] != 4) printf("Error.\n");
      for(j = 0; j < nSize; j++)
        if (pfC[j + 1] != 0) printf("Error.\n");
      pfC += nSize + 1;
    }
    if (pfC[0] != 4) printf("Error.\n");

    DSM_UnMapRegion(3);
  }

  DSM_WaitInit(3);

  DSM_Stop();

  return 0;
}
```

The Mandelbrot Fractal Program

This section presents a Mandelbrot fractal program using the BOT partitioning. The region type used to store the resulting image is eager release VMH home write-invalidate. The task data that must be given to the program is: "Width:Height:A:B:WidthA:NbIter:OutputFile:NbTasks". "Width" and "Height" give the width and height of the resulting image. "A" and "B" are the position in the complex space of the lower left corner of the image. "WidthA" is the width in the "A" axis of the image. "NbIter" is the number of iterations needed to determine if a complex number is in the Mandelbrot set. "OutputFile" is the file where the resulting image is saved in raw RGB format. "NbTasks" is the number of tasks placed in the BOT. As the output file is saved in raw RGB format, it must be converted to BMP or JPEG with the UNIX utility "convert" before visualization. This program can be run on two nodes using twenty tasks with the command line: `"Mandelbrot -- -d 512:512:-2:-2:4:50:out.rgb:20 -h node0 -h node1"`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
#include <unistd.h>

#include "DSM.h"

struct color_t {
```

```
  uint8_t r, g, b, h; /* h to pad to 4 bytes. */
};

struct Task101_t {
  double a, b, dWidthA;
  int nWidth, nHeight, nNbIterMax, nNbTasks;
};

struct Task102_t {
  int nWidth, nHeight;
};

struct Task103_t {
  double a, b, dDeltaA, dDeltaB;
  int nWidth, nHeight, nNbIterMax, nMyTask, nNbTasks;
};

int main(int argc, char *argv[]) {
  int bLoop = 1;

  while (getopt(argc, argv, "") != -1) ;

  DSM_Start(argc, argv);

  while (bLoop) {
    DSMTask_t Task;

    switch(DSM_GetTask(&Task)) {
      case DSM_OK:
        switch(Task.nTaskType) {
          case 100: {
            struct Task101_t *pT101;
            struct Task102_t *pT102;
            DSMAddTask_t pAddTasks[2];
            DSMRegionData_t RD;

            DSM_InitAddTask(pAddTasks + 0, 101, NULL, 0, -1);
            DSM_InitAddTask(pAddTasks + 1, 102, NULL, 0, 0);

            pT101 = (struct Task101_t*)pAddTasks[0].TaskData;
            pT102 = (struct Task102_t*)pAddTasks[1].TaskData;

            pT101->nWidth     = atoi(strtok(Task.TaskData, ":"));
            pT101->nHeight    = atoi(strtok(NULL, ":"));
            pT101->a          = atof(strtok(NULL, ":"));
            pT101->b          = atof(strtok(NULL, ":"));
            pT101->dWidthA    = atof(strtok(NULL, ":"));
            pT101->nNbIterMax = atoi(strtok(NULL, ":"));
            strcpy((char*)&pT102[1], strtok(NULL, ":"));
            pT101->nNbTasks   = atoi(strtok(NULL, ":"));

            pT102->nWidth  = pT101->nWidth;
            pT102->nHeight = pT101->nHeight;

            RD.nRegionNumber = 1;
            RD.Size          = pT101->nHeight * pT101->nWidth * sizeof(struct color_t);
            RD.Type          = DSM_REGION_ERVHWI;
            RD.Granularity   = DSM_REGION_GRAN_4096;
            RD.SpecificData.ERVHWIData.nDiffUnit = sizeof(struct color_t);
            DSM_CreateRegion(&RD);

            DSM_ReplaceTask(&Task, pAddTasks, 2);
```

```
  break;
}
case 101: {
  struct Task101_t *pT101;
  struct Task103_t T103;
  DSMAddTask_t *pAddTasks;

  pT101 = (struct Task101_t*)Task.TaskData;

  pAddTasks = malloc(pT101->nNbTasks * sizeof(DSMAddTask_t));

  T103.a          = pT101->a;
  T103.b          = pT101->b;
  T103.dDeltaA    = pT101->dWidthA / pT101->nWidth;
  T103.dDeltaB    = T103.dDeltaA;
  T103.nWidth     = pT101->nWidth;
  T103.nHeight    = pT101->nHeight;
  T103.nNbIterMax = pT101->nNbIterMax;
  T103.nNbTasks   = pT101->nNbTasks;

  for(T103.nMyTask = 0; T103.nMyTask < pT101->nNbTasks; T103.nMyTask++)
    DSM_InitAddTask(pAddTasks + T103.nMyTask, 103, (char*)&T103, sizeof(T103), -1);

  DSM_ReplaceTask(&Task, pAddTasks, pT101->nNbTasks);
  free(pAddTasks);

  break;
}
case 102: {
  int i;
  struct color_t *pImage;
  struct Task102_t *pT102;
  FILE *F;

  pT102 = (struct Task102_t*)Task.TaskData;

  DSM_MapRegion(1, (void**)&pImage);

  F = fopen((char*)&pT102[1], "w");
  for(i = 0; i < pT102->nHeight * pT102->nWidth; i++)
    fprintf(F, "%c%c%c", pImage[i].r, pImage[i].g, pImage[i].b);
  fclose(F);

  DSM_UnMapRegion(1);

  DSM_CommitTask(&Task);

  break;
}
case 103: {
  double ca, cb, za, zb, zaSquared, zbSquared, dT;
  int id0, id1, nPerW, nLeft, ns, ne, i, j, nNbI;
  struct color_t *pImage;
  struct Task103_t *pT103;

  pT103 = (struct Task103_t*)Task.TaskData;

  id0   = pT103->nMyTask;
  id1    = id0 + 1;
  nPerW = pT103->nHeight / pT103->nNbTasks;
  nLeft = pT103->nHeight % pT103->nNbTasks;
```

```
            ns    = id0 * nPerW + id0 * nLeft / pT103->nNbTasks;
            ne    = id1 * nPerW + id1 * nLeft / pT103->nNbTasks;

            DSM_MapRegion(1, (void**)&pImage);

            for(i = ns; i < ne; i++) {
              cb = pT103->b + pT103->dDeltaB * (pT103->nHeight - i);

              for(j = 0; j < pT103->nWidth; j++) {
                ca = pT103->a + pT103->dDeltaA * j;
                za = ca; zb = cb;
                zaSquared = za * za;
                zbSquared = zb * zb;

                for(nNbI = 0; nNbI < pT103->nNbIterMax && zaSquared + zbSquared <= 4.0; nNbI++) {
                  zb = 2 * za * zb + cb;
                  za = zaSquared - zbSquared + ca;

                  zaSquared = za * za;
                  zbSquared = zb * zb;
                }

                dT = 1.0 - (double)nNbI / pT103->nNbIterMax;

                pImage[i * pT103->nWidth + j].r = 0;
                pImage[i * pT103->nWidth + j].g = 255.0 * dT;
                pImage[i * pT103->nWidth + j].b = 255.0 * dT;
              }
            }

            DSM_Release();
            DSM_UnMapRegion(1);

            DSM_CommitTask(&Task);

            break;
          }
        }
        break;
      case DSM_NO_TASK_AVAILABLE:
        usleep(1000); break;
      case DSM_NO_MORE_TASK:
        bLoop = 0; break;
      default:
        printf("Error.\n"); bLoop = 0; break;
    }
  }

  DSM_Stop();

  return 0;
}
```

The Fast Matrix Exponentiation Program

This section presents a fast matrix exponentiation program using the static partitioning. The matrix to be exponentiated is the identity matrix multiplied by two. The resulting matrix should be an identity matrix multiplied by 2^{exp}, which is verified by the program. The region types used are write-once VMH home for the matrix to be exponentiated and eager release VMH home write-update for the resulting matrix. This program can be run on two nodes using the command line: `"MatExp -e 10 -s 128 -- -h node0 -h node1"`.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "DSM.h"

void MatExp(float *pfA, float *pfB, float *pfC, float *pfD,
            int nSize, int nExp, int ns, int ne) {
  int i, j, k;

  if (nExp > 1) {
    if (nExp % 2) {
      MatExp(pfA, pfB, pfC, pfD, nSize, nExp - 1, ns, ne);
      pfC = pfB;
    }
    else {
      MatExp(pfA, pfB, pfC, pfD, nSize, nExp/2, ns, ne);

      for(i = 0; i < nSize; i++)
        for(j = 0; j < nSize; j++)
          pfC[j * nSize + i] = pfA[i * nSize + j];

      DSM_WaitInit(4);
    }

    for(i = ns; i < ne; i++) {
      for(j = 0; j < nSize; j++)
        pfD[j] = pfA[i * nSize + j];

      for(j = 0; j < nSize; j++) {
        pfA[i * nSize + j] = 0;
        for(k = 0; k < nSize; k++)
          pfA[i * nSize + j] += pfD[k] * pfC[j * nSize + k];
      }
    }

    DSM_Release();
    DSM_WaitInit(5);
  }
  else {
    for(i = ns; i < ne; i++)
      for(j = 0; j < nSize; j++)
        pfA[i * nSize + j] = pfB[j * nSize + i];

    DSM_Release();
    DSM_WaitInit(6);
```

```
  }
}

int main(int argc, char *argv[]) {
  float *pfA, *pfB, *pfC, *pfD, fExpected;
  int c, nExp, nSize, id0, id1, nPerW, nLeft, ns, ne, i, j;
  DSMRegionData_t RD;

  nExp  = 10;
  nSize = 128;
  while ((c = getopt(argc, argv, "e:s:v")) != -1)
    switch(c) {
      case 'e':
        nExp = atoi(optarg); break;
      case 's':
        nSize = atoi(optarg); break;
    }

  DSM_Start(argc, argv);

  pfC = malloc(nSize * nSize * sizeof(float));
  pfD = malloc(nSize * sizeof(float));

  if (!DSM_GetWorkerIndex()) {
    RD.nRegionNumber = 1;
    RD.Size          = nSize * nSize * sizeof(float);
    RD.Type          = DSM_REGION_ERVHWU;
    RD.Granularity   = DSM_REGION_GRAN_4096;
    RD.SpecificData.ERVHWUData.nDiffUnit = sizeof(float);
    DSM_CreateRegion(&RD);

    RD.nRegionNumber = 2;
    RD.Type          = DSM_REGION_WOVH;
    DSM_CreateRegion(&RD);

    DSM_MapRegion(2, (void**)&pfB);

    for(i = 0; i < nSize * nSize; i++) pfB[i] = 0;
    for(i = 0; i < nSize * nSize; i += nSize + 1) pfB[i] = 2;

    DSM_UnMapRegion(2);
  }

  DSM_WaitInit(1);

  id0   = DSM_GetWorkerIndex();
  id1   = id0 + 1;
  nPerW = nSize / DSM_GetNumberOfWorkers();
  nLeft = nSize % DSM_GetNumberOfWorkers();
  ns    = id0 * nPerW + id0 * nLeft / DSM_GetNumberOfWorkers();
  ne    = id1 * nPerW + id1 * nLeft / DSM_GetNumberOfWorkers();

  DSM_MapRegion(1, (void**)&pfA);
  DSM_MapRegion(2, (void**)&pfB);

  MatExp(pfA, pfB, pfC, pfD, nSize, nExp, ns, ne);

  DSM_UnMapRegion(2);
  DSM_UnMapRegion(1);

  DSM_WaitInit(2);
```

```
  if (!DSM_GetWorkerIndex()) {
    fExpected = pow(2, nExp);

    DSM_MapRegion(1, (void**)&pfA);

    for(i = 0; i < nSize - 1; i++) {
      if (pfA[0] != fExpected) printf("Error.\n");
      for(j = 0; j < nSize; j++)
        if (pfA[j + 1] != 0) printf("Error.\n");
      pfA += nSize + 1;
    }
    if (pfA[0] != fExpected) printf("Error.\n");

    DSM_UnMapRegion(1);
  }

  DSM_WaitInit(3);

  free(pfD);
  free(pfC);

  DSM_Stop();

  return 0;
}
```

The N-Queens Program

This section presents a n-queens program using the static partitioning. The only shared data required is the number of solutions which is put in an eager release VMH home write-invalidate region. To protect the access to this variable, a mutex is used. This program can be run on two nodes with the command line: `"NQueens -n 15 -l 3 -- -h node0 -h node1"`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "DSM.h"

void PlaceQueens(int nRow, int nNbQueens, int nCol, int nD1, int nD2, uint64_t *pnNbSols) {
  int i, nThisCol;

  if (nRow >= nNbQueens) pnNbSols[0]++;
  else {
    nD1 >>= 1;
    nD2 <<= 1;

    for(i = 0, nThisCol = 1; i < nNbQueens; i++, nThisCol <<= 1)
      if (!((nCol | nD1 | nD2) & nThisCol))
        PlaceQueens(nRow+1, nNbQueens, nCol | nThisCol,
                    nD1 | nThisCol, nD2 | nThisCol, pnNbSols);
  }
}
```

```
void PlaceQueensTasks(int nRow, int nNbQueens, int nLimitPlaced, int nCol, int nD1, int nD2,
                      int *pnNext, int nJump, int *pnTaskNumber, uint64_t*pnNbSols) {
  int i, nThisCol;

  if (nRow >= nLimitPlaced) {
    if (pnTaskNumber[0] == pnNext[0]) {
      pnNext[0] += nJump;
      PlaceQueens(nRow, nNbQueens, nCol, nD1, nD2, pnNbSols);
    }

    pnTaskNumber[0]++;
  }
  else {
    nD1 >>= 1;
    nD2 <<= 1;

    for(i = 0, nThisCol = 1; i < nNbQueens; i++, nThisCol <<= 1)
      if (!((nCol | nD1 | nD2) & nThisCol))
        PlaceQueensTasks(nRow+1, nNbQueens, nLimitPlaced, nCol | nThisCol, nD1 | nThisCol,
                         nD2 | nThisCol, pnNext, nJump, pnTaskNumber, pnNbSols);
  }
}

int main(int argc, char *argv[]) {
  int c, nNext, nTaskNumber, nNbQueens, nLimitPlaced;
  uint64_t nNbSols, *pnNbSols;
  DSMMutexData_t MD;
  DSMRegionData_t RD;

  nNbQueens    = 8;
  nLimitPlaced = 3;
  while ((c = getopt(argc, argv, "n:l:")) != -1)
    switch(c) {
      case 'n':
        nNbQueens = atoi(optarg); break;
      case 'l':
        nLimitPlaced = atoi(optarg); break;
    }

  DSM_Start(argc, argv);

  if (!DSM_GetWorkerIndex()) {
    RD.nRegionNumber = 1;
    RD.Size          = sizeof(uint64_t);
    RD.Type          = DSM_REGION_ERVHWI;
    RD.Granularity   = DSM_REGION_GRAN_4096;
    RD.SpecificData.ERVHWIData.nDiffUnit = sizeof(uint64_t);
    DSM_CreateRegion(&RD);

    DSM_MapRegion(1, (void**)&pnNbSols);

    pnNbSols[0] = 0;

    DSM_Release();
    DSM_UnMapRegion(1);

    MD.nMutexNumber = 1;
    MD.Type         = DSM_MUTEX_SHL;
    DSM_CreateMutex(&MD);
  }

  DSM_WaitInit(1);
```

```
  nNbSols     = 0;
  nTaskNumber = 0;
  nNext       = DSM_GetWorkerIndex();
  PlaceQueensTasks(0, nNbQueens, nLimitPlaced, 0, 0, 0, &nNext,
                   DSM_GetNumberOfWorkers(), &nTaskNumber, &nNbSols);

  DSM_MapRegion(1, (void**)&pnNbSols);
  DSM_LockMutex(1, DSM_FLAGS_NONE);

  pnNbSols[0] += nNbSols;

  DSM_Release();
  DSM_UnlockMutex(1);
  DSM_UnMapRegion(1);

  DSM_WaitInit(2);

  if (!DSM_GetWorkerIndex()) {
    printf("There were %d tasks.\n", nTaskNumber);

    DSM_MapRegion(1, (void**)&pnNbSols);
    printf("There are %lld solutions.\n", pnNbSols[0]);
    DSM_UnMapRegion(1);
  }

  DSM_WaitInit(3);

  DSM_Stop();

  return 0;
}
```

The Tridiag Program

This section presents a program solving a tridiagonal equation systems using the static partitioning. The equation system is built by choosing random values for the variables and using only one as the coefficients. After the computation, the solutions found are compared with the randomly chosen values. The region type used is eager release VMH home write-invalidate for the system and the solutions. This program can be run on two nodes using the command line: `"Tridiag -s 128 -- -h node0 -h node1"`.

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "DSM.h"

int main(int nNbArgs, char *strArgs[]) {
  float *pdA, *pdAT, *pdB, *pdC, *pdCT, *pdT, *pdX, *pdY, *pdYT;
  int c, i, id0, id1, nPerW, nLeft, ns, ne, nSize, nOff;
```

```
DSMRegionData_t RD;

nSize = 128;
while ((c = getopt(nNbArgs, strArgs, "s:")) != -1)
  switch(c) {
    case 's':
      nSize = atoi(optarg); break;
  }

DSM_Start(nNbArgs, strArgs);

if (!DSM_GetWorkerIndex()) {
  RD.nRegionNumber = 1;
  RD.Size          = 8 * nSize * sizeof(float);
  RD.Type          = DSM_REGION_ERVHWI;
  RD.Granularity   = 12;
  RD.SpecificData.ERVHWIData.nDiffUnit = sizeof(float);
  DSM_CreateRegion(&RD);

  DSM_MapRegion(1, (void**)&pdT);

  for(i = 0; i < 8 * nSize; i++) {
    pdT[i] = 0;
  }

  pdA = pdT + nSize;
  pdC = pdT + 3 * nSize;
  pdY = pdT + 5 * nSize;
  pdB = pdT + 7 * nSize;

  pdX  = malloc(nSize * sizeof(float));

  srand(0);
  for(i = 0; i < nSize; i++) {
    pdA[i] = pdB[i] = pdC[i] = 1;
    pdX[i] = rand() % 10;
  }
  pdA[0]       = 0;
  pdC[nSize-1] = 0;

  pdY[0] = pdX[0] + pdX[1];
  for(i = 1; i < nSize-1; i++) {
    pdY[i] = pdX[i-1] + pdX[i] + pdX[i+1];
  }
  pdY[nSize] = pdX[nSize-2] + pdX[nSize-1];

  DSM_Release();

  DSM_UnMapRegion(1);
}

DSM_WaitInit(1);

id0   = DSM_GetWorkerIndex();
id1   = id0 + 1;
nPerW = nSize / DSM_GetNumberOfWorkers();
nLeft = nSize % DSM_GetNumberOfWorkers();
ns    = id0 * nPerW + id0 * nLeft / DSM_GetNumberOfWorkers();
ne    = id1 * nPerW + id1 * nLeft / DSM_GetNumberOfWorkers();

pdAT = malloc((ne - ns) * sizeof(float));
pdCT = malloc((ne - ns) * sizeof(float));
```

```
  pdYT = malloc((ne - ns) * sizeof(float));

  DSM_MapRegion(1, (void**)&pdT);

  pdA = pdT + nSize + ns;
  pdC = pdT + 3 * nSize + ns;
  pdY = pdT + 5 * nSize + ns;
  pdB = pdT + 7 * nSize + ns;

  for(nOff = 1; nOff <= nSize; nOff *= 2) {
    for(i = 0; i < (ne-ns); i++) {
      pdA[i] /= pdB[i];
      pdC[i] /= pdB[i];
      pdY[i] /= pdB[i];
    }

    for(i = 0; i < (ne-ns); i++) pdB[i]  = 1.0 - pdA[i] * pdC[i-nOff] - pdC[i] * pdA[i + nOff];
    for(i = 0; i < (ne-ns); i++) pdYT[i] = pdY[i] - pdA[i] * pdY[i-nOff] - pdC[i+nOff];
    for(i = 0; i < (ne-ns); i++) pdCT[i]  = -1.0 * pdC[i] * pdC[i+nOff];
    for(i = (ne-ns)-1; i >= 0; i--) pdAT[i] = -1.0 * pdA[i] * pdA[i-nOff];

    DSM_WaitInit(2);

    for(i = 0; i < (ne-ns); i++) {
      pdA[i] = pdAT[i];
      pdC[i] = pdCT[i];
      pdY[i] = pdYT[i];
    }

    DSM_Release();

    DSM_WaitInit(3);
  }

  DSM_UnMapRegion(1);

  if (!DSM_GetWorkerIndex()) {
    DSM_MapRegion(1, (void**)&pdT);

    pdY = pdT + 5 * nSize;
    pdB = pdT + 7 * nSize;

    for(i = 0; i < nSize; i++)
      if (fabs(pdX[i] * pdB[i] - pdY[i]) > 0.0001)
        printf("Error.\n");

    free(pdX);

    DSM_UnMapRegion(1);
  }

  DSM_WaitInit(4);

  DSM_Stop();

  free(pdAT);
  free(pdCT);
  free(pdYT);

  return 0;
}
```

Appendix C

An Example of Annotated YADL Program

This appendix presents the matrix exponentiation program using annotated versions of the memory. It is supplied to show the reader how to use the annotated version of the write-once and eager release WU region types. It basically is the same example as presented in Appendix B, but using annotated versions of the write-once and eager release write-update regions. Again, this program can be run on two nodes using the command line: "`MatExp -e 10 -s 128 -- -h node0 -h node1`".

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "DSM.h"

void MatExp(float *pfA, float *pfB, float *pfC, float *pfD,
            int nSize, int nExp, int ns, int ne) {
  int i, j, k;
  float *pfT;

  if (nExp > 1) {
    if (nExp % 2) {
      MatExp(pfA, pfB, pfC, pfD, nSize, nExp-1, ns, ne);
      pfC = pfB;
    }
    else {
      MatExp(pfA, pfB, pfC, pfD, nSize, nExp/2, ns, ne);

      for(i = 0; i < nSize; i++)
```

```
        for(j = 0; j < nSize; j++)
          pfC[j * nSize + i] = pfA[i * nSize + j];

      DSM_WaitInit(4);
    }

    DSM_BeginAccess(1, ns * nSize * sizeof(float), (ne-ns) * nSize * sizeof(float),
                    DSM_REGION_ACCESS_WRITE, (void**)&pfT);

    for(i = ns; i < ne; i++) {
      for(j = 0; j < nSize; j++)
        pfD[j] = pfA[i * nSize + j];

      for(j = 0; j < nSize; j++) {
        pfA[i * nSize + j] = 0;
        for(k = 0; k < nSize; k++)
          pfA[i * nSize + j] += pfD[k] * pfC[j * nSize + k];
      }
    }

    DSM_EndAccess(1, ns * nSize * sizeof(float), (ne-ns) * nSize * sizeof(float),
                  DSM_REGION_ACCESS_WRITE);
    DSM_Release();

    DSM_WaitInit(5);
  }
  else {
    DSM_BeginAccess(1, ns * nSize * sizeof(float), (ne-ns) * nSize * sizeof(float),
                    DSM_REGION_ACCESS_WRITE, (void**)&pfT);

    for(i = ns; i < ne; i++)
      for(j = 0; j < nSize; j++)
        pfA[i * nSize + j] = pfB[j * nSize + i];

    DSM_EndAccess(1, ns * nSize * sizeof(float), (ne-ns) * nSize * sizeof(float),
                  DSM_REGION_ACCESS_WRITE);
    DSM_Release();

    DSM_WaitInit(6);
  }
}

int main(int nNbArgs, char *strArgs[]) {
  float fExpected;
  float *pfA, *pfB, *pfC, *pfD;
  int c, i, j, id0, id1, nPerW, nLeft, ns, ne, nExp, nSize;
  DSMRegionData_t RD;

  nExp  = 10;
  nSize = 128;
  while ((c = getopt(nNbArgs, strArgs, "e:s:")) != -1)
    switch(c) {
      case 'e':
        nExp = atoi(optarg); break;
      case 's':
        nSize = atoi(optarg); break;
    }

  DSM_Start(nNbArgs, strArgs);

  pfC = malloc(nSize * nSize * sizeof(float));
  pfD = malloc(nSize * sizeof(float));
```

```
if (!DSM_GetWorkerIndex()) {
  RD.nRegionNumber = 1;
  RD.Size          = nSize * nSize * sizeof(float);
  RD.Type          = DSM_REGION_ERAHWU;
  RD.Granularity   = 12;
  RD.SpecificData.ERAHWUData.nDiffUnit = sizeof(float);
  DSM_CreateRegion(&RD);

  RD.nRegionNumber = 2;
  RD.Size          = nSize * nSize * sizeof(float);
  RD.Type          = DSM_REGION_WOAH;
  RD.Granularity   = 12;
  DSM_CreateRegion(&RD);

  DSM_MapRegion(2, (void**)&pfB);
  DSM_BeginAccess(2, 0, nSize * nSize * sizeof(float),
                  DSM_REGION_ACCESS_READ_WRITE, (void**)&pfB);

  for(i = 0; i < nSize * nSize; i++) pfB[i] = 0;
  for(i = 0; i < nSize * nSize; i += nSize + 1) pfB[i] = 2;

  DSM_EndAccess(2, 0, nSize * nSize * sizeof(float), DSM_REGION_ACCESS_READ_WRITE);
  DSM_UnMapRegion(2);
}

DSM_WaitInit(1);

id0   = DSM_GetWorkerIndex();
id1   = id0 + 1;
nPerW = nSize / DSM_GetNumberOfWorkers();
nLeft = nSize % DSM_GetNumberOfWorkers();
ns    = id0 * nPerW + id0 * nLeft / DSM_GetNumberOfWorkers();
ne    = id1 * nPerW + id1 * nLeft / DSM_GetNumberOfWorkers();

DSM_MapRegion(1, (void**)&pfA);
DSM_MapRegion(2, (void**)&pfB);

DSM_BeginAccess(2, 0, nSize * nSize * sizeof(float), DSM_REGION_ACCESS_READ, (void**)&pfB);

MatExp(pfA, pfB, pfC, pfD, nSize, nExp, ns, ne);

DSM_EndAccess(2, 0, nSize * nSize * sizeof(float), DSM_REGION_ACCESS_READ);

DSM_UnMapRegion(2);
DSM_UnMapRegion(1);

DSM_WaitInit(2);

if (!DSM_GetWorkerIndex()) {
  fExpected = pow(2, nExp);

  DSM_MapRegion(1, (void**)&pfA);

  for(i = 0; i < nSize - 1; i++) {
    if (pfA[0] != fExpected) printf("Error.\n");
    for(j = 0; j < nSize; j++)
      if (pfA[j + 1] != 0) printf("Error.\n");
    pfA += nSize + 1;
  }
  if (pfA[0] != fExpected) printf("Error.\n");
```

```
    DSM_UnMapRegion(1);
  }

  DSM_WaitInit(3);

  free(pfD);
  free(pfC);

  DSM_Stop();

  return 0;
}
```

Appendix D

Other Benchmark Results

	Static Without Increased Granularity				Static With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1115.7	1.06	567.9	2.07	1118.5	1.05	566.2	2.08
2 nodes	568.4	2.07	290.4	4.06	568.4	2.07	290.1	4.06
4 nodes	288.6	4.08	151.7	7.76	288.6	4.08	151.8	7.76
8 nodes	148.6	7.93	81.5	14.44	148.3	7.94	81.2	14.50
12 nodes	102.4	11.51	58.5	20.12	101.8	11.57	58.6	20.10
16 nodes	79.2	14.87	47.8	24.65	78.9	14.94	47.5	24.78
20 nodes	65.3	18.03	41.3	28.49	65.1	18.10	40.9	28.81
24 nodes	56.3	20.90	36.7	32.10	55.9	21.07	36.7	32.11
28 nodes	50.1	23.53	34.0	34.64	49.6	23.76	33.3	35.35

Table D.1: Matrix Multiplication Exec. Times and Speedups With Increased Granularity.

	Static Without Increased Granularity				Static With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1621.2	0.99	812.3	1.97	1621.8	0.99	812.7	1.97
2 nodes	811.9	1.97	407.3	3.93	812.3	1.97	407.2	3.93
4 nodes	406.9	3.93	204.5	7.82	407.1	3.93	204.6	7.81
8 nodes	204.3	7.82	103.3	15.48	204.4	7.82	103.3	15.47
12 nodes	137.6	11.62	70.8	22.57	137.6	11.62	70.8	22.58
16 nodes	103.2	15.49	52.6	30.38	103.3	15.48	52.7	30.33
20 nodes	83.9	19.06	43.1	37.12	83.9	19.05	43.1	37.06
24 nodes	70.7	22.61	36.8	43.47	70.7	22.61	36.8	43.48
28 nodes	61.5	25.97	32.7	48.94	61.3	26.09	32.7	48.93

Table D.2: Ray Tracing Static Exec. Times and Speedups With Increased Granularity.

	BOT Without Increased Granularity				BOT With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1598.6	1.00	812.3	1.97	1596.2	1.00	809.7	1.97
2 nodes	806.2	1.98	408.9	3.91	802.3	1.99	406.9	3.93
4 nodes	406.3	3.93	206.6	7.74	404.2	3.96	204.9	7.80
8 nodes	205.8	7.77	105.6	15.14	203.8	7.84	104.5	15.30
12 nodes	138.7	11.52	72.2	22.14	137.0	11.67	71.3	22.44
16 nodes	105.5	15.15	55.1	28.99	104.3	15.33	55.0	29.04
20 nodes	85.2	18.76	44.4	36.04	83.8	19.09	43.6	36.63
24 nodes	72.2	22.14	38.1	41.95	70.9	22.55	37.2	42.96
28 nodes	63.5	25.16	34.2	46.72	61.6	25.93	33.3	47.96

Table D.3: Ray Tracing BOT Exec. Times and Speedups With Increased Granularity.

	BOT With 1024 Tasks				BOT With 256 Tasks			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1598.6	1.00	812.3	1.97	1595.0	1.00	803.6	1.99
2 nodes	806.2	1.98	408.9	3.91	801.0	2.00	403.1	3.97
4 nodes	406.3	3.93	206.6	7.74	403.0	3.97	203.1	7.87
8 nodes	205.8	7.77	105.6	15.14	202.8	7.88	102.3	15.63
12 nodes	138.7	11.52	72.2	22.14	140.3	11.40	72.2	22.13
16 nodes	105.5	15.15	55.1	28.99	103.0	15.52	52.6	30.39
20 nodes	85.2	18.76	44.4	36.04	84.5	18.93	47.0	34.05
24 nodes	72.2	22.14	38.1	41.95	72.3	22.12	40.7	39.24
28 nodes	63.5	25.16	34.2	46.72	65.7	24.34	34.1	46.93

Table D.4: Ray Tracing Exec. Times and Speedups With Less Tasks.

	Static Without Increased Granularity				Static With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1722.4	0.93	856.9	1.87	1722.9	0.93	858.0	1.87
2 nodes	862.7	1.86	431.4	3.72	862.4	1.86	430.4	3.73
4 nodes	432.4	3.71	217.1	7.39	432.2	3.71	216.4	7.41
8 nodes	217.0	7.39	109.0	14.71	217.0	7.39	109.3	14.67
12 nodes	147.3	10.89	74.3	21.58	147.3	10.89	74.4	21.54
16 nodes	109.6	14.63	55.6	28.84	109.6	14.64	55.7	28.81
20 nodes	90.2	17.78	45.9	34.92	90.1	17.79	45.9	34.97
24 nodes	74.6	21.51	39.6	40.51	74.6	21.50	39.6	40.46
28 nodes	64.6	24.81	34.6	46.32	64.5	24.85	34.6	46.34

Table D.5: Mandelbrot Static Exec. Times and Speedups With Increased Granularity.

	BOT Without Increased Granularity				BOT With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1622.8	0.99	827.7	1.94	1620.6	0.99	829.6	1.93
2 nodes	817.6	1.96	414.8	3.87	815.3	1.97	413.2	3.88
4 nodes	412.9	3.88	209.4	7.66	410.2	3.91	209.4	7.66
8 nodes	208.9	7.68	106.4	15.07	207.5	7.73	106.0	15.13
12 nodes	141.2	11.36	72.9	22.00	139.8	11.47	72.1	22.24
16 nodes	107.7	14.89	55.9	28.67	106.0	15.12	54.9	29.19
20 nodes	87.3	18.37	45.7	35.07	85.5	18.76	44.7	35.91
24 nodes	73.7	21.76	39.0	41.08	71.9	22.31	38.3	41.85
28 nodes	64.7	24.80	34.2	46.85	62.2	25.76	33.6	47.80

Table D.6: Mandelbrot BOT Exec. Times and Speedups With Increased Granularity.

	BOT With 1024 Tasks				BOT With 256 Tasks			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1622.8	0.99	827.7	1.94	1619.2	0.99	816.5	1.96
2 nodes	817.6	1.96	414.8	3.87	812.8	1.97	408.7	3.92
4 nodes	412.9	3.88	209.4	7.66	409.7	3.91	206.2	7.78
8 nodes	208.9	7.68	106.4	15.07	205.2	7.81	104.2	15.39
12 nodes	141.2	11.36	72.9	22.00	142.5	11.26	73.3	21.89
16 nodes	107.7	14.89	55.9	28.67	104.0	15.43	54.0	29.68
20 nodes	87.3	18.37	45.7	35.07	85.7	18.72	47.6	33.69
24 nodes	73.7	21.76	39.0	41.08	73.1	21.94	41.3	38.86
28 nodes	64.7	24.80	34.2	46.85	67.1	23.90	36.0	44.53

Table D.7: Mandelbrot Exec. Times and Speedups With Less Tasks.

	Static Without Increased Granularity				Static With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1123.9	1.02	574.3	1.99	1118.0	1.02	573.7	1.99
2 nodes	581.8	1.96	295.0	3.87	581.0	1.97	295.1	3.87
4 nodes	296.0	3.86	154.6	7.39	296.1	3.86	154.6	7.39
8 nodes	154.1	7.41	85.9	13.30	154.0	7.42	85.8	13.32
12 nodes	107.5	10.63	64.8	17.63	107.6	10.62	64.8	17.64
16 nodes	84.5	13.51	55.3	20.67	84.6	13.50	55.0	20.79
20 nodes	71.2	16.05	50.2	22.78	71.3	16.04	50.4	22.69
24 nodes	62.6	18.26	47.2	24.20	62.6	18.25	47.4	24.12
28 nodes	56.8	20.11	46.0	24.82	56.9	20.08	46.0	24.81

Table D.8: Matrix Exp. Static Exec. Times and Speedups With Increased Granularity.

	BOT Without Increased Granularity				BOT With Increased Granularity			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1157.4	0.99	583.9	1.96	1155.5	0.99	582.7	1.96
2 nodes	587.2	1.95	301.9	3.78	588.6	1.94	301.5	3.79
4 nodes	300.6	3.80	162.6	7.03	299.8	3.81	162.8	7.02
8 nodes	160.2	7.13	99.2	11.52	159.4	7.17	99.4	11.49
12 nodes	116.7	9.79	85.8	13.32	116.4	9.82	85.6	13.34
16 nodes	95.8	11.93	83.4	13.71	95.6	11.96	83.1	13.75
20 nodes	85.6	13.35	86.3	13.25	85.6	13.35	86.3	13.23
24 nodes	80.8	14.14	94.0	12.15	80.5	14.19	92.4	12.36
28 nodes	78.3	14.58	100.8	11.34	77.8	14.69	100.5	11.37

Table D.9: Matrix Exp. BOT Exec. Times and Speedups With Increased Granularity.

	BOT With 1024 Tasks				BOT With 256 Tasks			
	1 Worker		2 Workers		1 Worker		2 Workers	
	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1 node	1157.4	0.99	583.9	1.96	1152.1	0.99	581.4	1.97
2 nodes	587.2	1.95	301.9	3.78	581.8	1.96	300.3	3.80
4 nodes	300.6	3.80	162.6	7.03	299.6	3.81	162.0	7.05
8 nodes	160.2	7.13	99.2	11.52	159.2	7.18	98.9	11.55
12 nodes	116.7	9.79	85.8	13.32	116.1	9.84	83.4	13.70
16 nodes	95.8	11.93	83.4	13.71	96.4	11.85	80.0	14.28
20 nodes	85.6	13.35	86.3	13.25	85.8	13.31	83.1	13.76
24 nodes	80.8	14.14	94.0	12.15	80.6	14.18	88.0	12.98
28 nodes	78.3	14.58	100.8	11.34	78.3	14.60	94.6	12.08

Table D.10: Matrix Exponentiation Exec. Times and Speedups With Less Tasks.