Techniques to Improve the Performance of Software-based Distributed Shared Memory Systems

by

Churngwei Chu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
New York University
August 1998

Approved: _____

Professor Zvi Kedem
Research Advisor

To my grandfather, Cheng-yuan

Acknowledgments

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; and by the National Science Foundation under grant number CCR-94-11590. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

I am very grateful for the guidance of my advisor, Professor Zvi Kedem. I always bring my biggest problems to him. I am especially indebted to Peter Piatko who brought to my attention the heat flow parallel application and who always had time for discussions with me. I also owe many thanks to Arash Baratloo and Karp Jeong who helped me solve several technical problems. I would like to thank my friends, Renu Boonoeb and Shau-Di Du who shared complaints, frustration, and joy with me. Lastly, I would like to thank my parents and my brother. Without their support, this thesis would not have been possible.

Abstract

Software distributed shared memory systems are able to provide programmers with the illusion of global shared memory on networked workstations without special hardware support. This thesis identifies two problems in contemporary software distributed shared memory systems: (1) poor application programming interfaces for programmers who need to solve complicated synchronization problems and (2) inefficiencies in traditional multiple writer protocols. We propose a solution to both of these problems. One is the introduction of *user-definable high level synchronization primitives* to provide a better application programming interface. The other is the *single-owner* protocol to provide efficiency. In order to accommodate user-definable high level synchronization primitives, a variant of release consistency is also proposed.

User-definable high level synchronization primitives provide a paradigm for users to define their own synchronization primitives instead of relying on traditional low level synchronization primitives, such as barriers and locks. The single-owner protocol reduces the number of messages from $O(n^2)$ messages (the number of messages needed in the *multiple-owner* protocol) to $\Theta(n)$ messages when there are first n writers writing to a page and then n readers reading the page. Unlike some multiple-owner protocols, in the single-owner protocol garbage collection is performed asynchronously, and the size of a message for doing memory update is smaller in most cases.

We also evaluate the tradeoffs between the single-owner protocol and multiple-owner protocols. We have found that in most cases the single-owner protocol uses fewer messages than multiple-owner protocols, but there are some computations which may perform better with some multiple-owner protocols. In order to combine the advantages of both

protocols, we propose a *hybrid owner* protocol which can be used to increase the efficiency in an adaptive way, with some pages managed by the single-owner protocol and some by a multiple-owner protocol.

Finally, five applications are evaluated using the single-owner protocol and a particular multiple-owner protocol called the lazy invalidate protocol. The performance of these two protocols is compared. We also demonstrate the use of user-definable high level synchronization primitives on one of the applications, and compare its performance against the same application constructed using only low-level synchronization primitives.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

High performance computing will increasingly utilize distributed platforms consisting of standard workstations connected by high-speed networks, called a network of workstations (NOW). In order to make such platforms practical for utilization by a wide community of users, an extensive research effort by computer scientists has been undertaken. The approaches taken include message passing, software distributed shared memory, shared tuple space, and remote procedure calls.

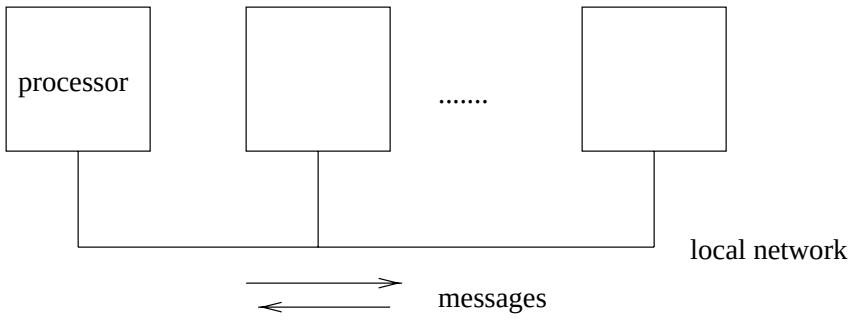

Figure 1.1: Message passing model

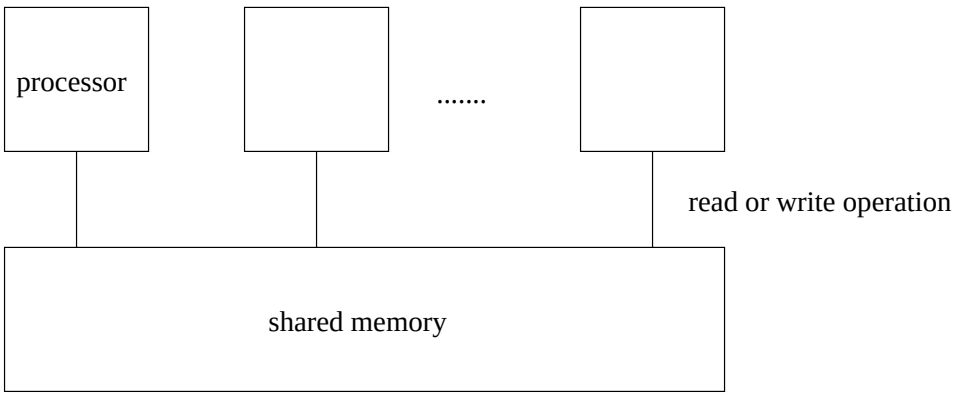

Figure 1.2: Distributed shared memory

Two of these approaches are currently most viable for providing the software environment to enable the utilization of networked workstations for parallel computation. One is based on the *message passing* model, Fig. 1.1, embodied in, e.g. PVM and MPI [BDG+91, GLS94]. Although this approach is the most popular now, it is rather low level and makes programming very difficult. The other approach, that of providing the programmer with the illusion of global shared memory, by means of physically distributed memory (see Fig. 1.2), is currently a very promising approach for the next generation of software environments.

When programs execute on a distributed shared memory (DSM) system, the low level details of data movement are handled dynamically by the system itself without the intervention of the application programmer.

Of course, the underlying system still implements the shared memory by means of message passing. However, the programmer is not aware of this and does not need to control it. The key issue of efficiency depends heavily on the synchronization required by the computation and the amount of messages

required.

The interest in the potential of software DSM-based systems is evidenced by the vigorous program of research and by several prototypes that have been developed in recent years, including Midway, Munin, TreadMarks, and Quarks [BZS93, CBZ95, KCDZ94, CKK95].

DSM systems, however, have not yet reached the maturity of the message passing systems. In fact, researchers have identified bottlenecks in the performance of software DSM systems and have proposed techniques for removing them [CBZ91, CK96, KCDZ94, LDCZ97]. In this chapter we will give a brief introduction to the different models of distributed shared memory, discuss implementation issues, and then talk about the key problems of existing implementations.

1.2 Memory Consistency Models

A memory consistency model is a contract between programmers and shared memory which specifies how the memory operations of a program will be executed. Computer scientists have proposed different memory models to enhance distributed shared memory systems. In this section, we will give an introduction to those memory models and some terminology used in this thesis.

1.2.1 Strict Consistency

Strict consistency is the most stringent memory model. It requires any read operation to a memory location to return the latest write. This definition uses a global time to define what a read operation can get from the memory. This model mimics the memory behavior in a single processor.

1.2.2 Sequential Consistency

A global clock is hard to capture in a distributed system. Each processor in the distributed system may have its own local clock with a different view of time. The idea of recent time can be inconsistent in the system. [Lam79] proposed another model, called *sequential consistency*, to extend the idea of the strict consistency model.

Definition 1 *A system is sequentially consistent* [Lam79] *iff the result of any execution is the same as if the operations of all processors were executed in some total order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

Sequential consistency provides a view of a single global shared memory which receives read and write operations from processors in its program order [AG95]. The global shared memory serializes memory operations. Each processor can not proceed to the next memory operation until all previous ones are performed. A read operation does not return the value of a write until the value is visible to all other processors.

Even though sequential consistency is the canonical memory consistency model, [DSB86, DSB88, DS90, SD88, Sch89] have described the difficulties of implementing sequential consistency in many systems. In order to describe non-atomic memory operations in distributed systems, they also define the notion of an operation being *performed with respect to a processor*, *performed*, and *globally performed*.

A write operation *is performed with respect to a processor* when it has been observed by the processor, i.e., no future read of the processor to the same location can return the value of a previous write. A read operation *is performed with respect to a processor* when no future write of the processor

can affect the value returned by the read. A write or a read operation *is performed* when it is performed with respect to all processors. A write *is globally performed* when it is performed. A read *is globally performed* when it is performed and when the write whose value it reads is performed. Thus, a write or a read is globally performed when the value written or read is observed by all processors [Adv93].

1.2.3 Processor Consistency

Processor consistency [Goo89, GLL+90] was proposed to relax the program order constraints in the case of a write followed by a read operation to a different location. It allows the read operation to bypass the write before the write is serialized or made visible to other processors [AG95].

Definition 2 *A system is processor consistent* [GLL+90] *iff*

1. *Before a load is allowed to perform with respect to any other processor, all preceding load accesses must be performed.*
2. *Before a store is allowed to perform with respect to any other processor, all preceding accesses (loads and stores) must be performed.*

1.2.4 Weak Consistency

One family of relaxed memory models requires programmers to distinguish between data and synchronization operations. Accesses to synchronization variables are strongly ordered (for example, totally ordered), but data accesses follow a weaker order. Weak consistency was the first hybrid memory model proposed by [DSB86, DS90, Sch89].

Definition 3 *A system is weakly consistent* [DS90] *iff*

1. *Accesses to synchronization variables are strongly ordered.*

2. *No access to a synchronization variable is issued by a processor before all its previous data accesses are performed.*

3. *No access is issued by a processor before previous accesses to a synchronization variable are performed.*

"Previous" access in the definition means previous operations in the program order. A synchronization access works like a fence. All data accesses have to be performed before their subsequent synchronization operation is performed. Weak consistency may provide better performance than sequential consistency by allowing operations between two synchronization operations to be reordered, executed in parallel, and non-atomically [Adv93].

1.2.5 Release Consistency

Release consistency [GLL+90] is an extension of weak consistency. Release consistency classifies operations on shared memory into two categories, *special* and *ordinary*. *Special operations* also are classified into *sync* and *nsync*. *Sync* operations are either *release* operations or *acquire* operations. *Ordinary operations* refer to data accesses without conflicting with other operations. *Sync* accesses are used to order data accesses such that data operations may not conflict with each other. *Nsync* accesses are asynchronous data accesses. *Release* is a write synchronization operation. *Acquire* is a read synchronization operation.

Definition 4 *A system is release consistent iff*

1. *Before an ordinary load or store access is allowed to perform with respect to any other processor, all preceding acquire accesses must be performed.*

2. *Before a release access is allowed to be performed with respect to any other processor, all preceding ordinary load and store accesses must be performed.*

3. *Special accesses are sequentially consistent with respect to each other.*

Variants of release consistency models differ as to which consistency model special accesses follow. In some memory consistency models, special accesses follow sequential consistency; in other memory models they follow processor consistency [Goo89]. Those memory models are denoted as $\mathrm{RC_{SC}}$ and $\mathrm{RC_{PC}}$ respectively [Adv93].

1.2.6 Lazy Release Consistency

Even though the conventional release consistency allows ordinary accesses to be postponed until a release operation is executed, it still requires all ordinary accesses to be performed with respect to all processes. [KCZ92] and [Kel95] proposed the *lazy release consistency* which allows ordinary accesses to be performed with respect to some processes. (Because the lazy release consistency was implemented as a software distributed shared memory system, the term of process is used instead of processor.)

Definition 5 *A system is lazy release consistent* [Kel95] *iff*

1. *Before an ordinary read or write access is allowed to perform with respect to another process, all previous acquire accesses must be performed with respect to that other process.*

2. *Before a release access is allowed to perform with respect to any other process, all previous ordinary read and write accesses must be performed with respect to that other process.*

3. *Sync accesses are sequentially consistent with respect to one another.*

Lazy release consistency requires only that ordinary accesses to be performed with respect to other processes as subsequent releases become visible to them [Kel95]. It potentially provides better performance than the traditional release consistency by reducing the number of processes that must see the changes to shared memory. Only those processes that acquire the value which the writer process releases need to see the changes.

1.2.7 Message-driven Relaxed Consistency

In order to exploit both the message passing model and the shared memory model, [KFJ94] proposed *message-driven relaxed consistency* that combines those two mechanisms into a single system. Messages carrying explicit causality annotations are exchanged to trigger memory coherence actions. In addition to shared memory, message passing is another mechanism provided by the system to exchange information among processes.

Specifically, if a process sends a synchronization message to another process, the modifications in shared memory the sending process made are visible to the receiving process after the receiving process gets the message. If all messages are synchronization messages, the ordering of memory events is consist with the "happened before" relation, as defined by [Lam79].

1.3 Related Techniques for Implementing DSM Systems

In this section, we are going to give an brief introduction to the implementation techniques of software distributed shared memory systems related to this thesis.

1.3.1 Single Writer Protocol

[Li88, LH89] uses the *page-based* mechanism to implement a software distributed shared memory system called IVY. The pages of the shared memory are cached in all processes. Each cache in a process is protected by the operating system. Any access to an invalid cache by the application generates a segment fault. The fault handler then obtains a valid copy of the page for the process.

IVY implements sequential consistency. Each process may cache a copy of a page of shared memory. All copies of a page cached in the various processes have to be identical. Only one process is allowed to write to the page at any point in the execution.

When a process attempts to read a page, it gets a valid copy of the page from another process. Then the mode of the cache of the page is set to read-only. If there is another process that is writing to that page, then the writer's cache is also set to read-only.

Before a process writes to the local cache of the page, it invalidates all the existing copies of the page in all other processes and gets a copy of the page from another process if it does not have a current version of the page. The access mode of the local cache is then changed to read-write.

1.3.2 Multiple Writer Protocol

In order to ensure the total order and atomicity of *sequential consistency*, the value of a write operation to a variable in shared memory needs to be visible to the read operations on the variable following the write operation. The single writer protocol satisfies this requirement by invalidating all the existing copies of a page in other processes before a process writes to the page. The subsequent readers read the copy of the page which the writer process

writes. For a write operation, several messages are needed to invalidate all other copies of the page. The size of each response to a read request is the size of a physical page in memory.

Since release consistency does not require the written value of a write operation to be visible to other processes until the writer executes a release operation, it simulates another type of mechanism, called *multiple writer* protocol, which allows multiple writers write to the same page simultaneously.

Write-shared Data Protocol

Munin [Car93, CBZ95, CBZ91, Car95] proposed a *write-shared data* protocol to implement the conventional release consistency. The write-shared data protocol uses the page-based mechanism. However, it allows more than one process to write to the same page simultaneously. Inconsistent copies of a page may exist in the system. An ordinary write operation is performed by sending the final value of the variable to all the processes caching the variable before the release operation is performed.

For example, process p and process q write to the same page but different addresses. Then they wait for each other to finish on a barrier. Before they begin to wait for each other, they execute release operations. When process p executes a release operation, it gives the changes of the page to process q. When process q executes a release operation, it gives the changes of the page to process p. After process p and q both execute the release operations, their caches of the page are identical. The changes to a page before a release operation are called a *diff* of the page. A diff consists of the addresses and values of the changes in a page.

In contrast with the single writer protocol, the write-shared data protocol does not generate any invalidation messages when a process writes to a page. Only the diff of the page is sent in the write-shared data protocol instead of

sending the whole page to the reader as is done in the single writer protocol.

Lazy Invalidate Protocol

As we mentioned in Section 1.2.6, the conventional release consistency requires all ordinary accesses to be performed with respect to all processes even though some processes do not even access those variables. TreadMarks [KCZ92, Kel95, ACD+96] proposed *lazy release consistency* and its implementation, called *lazy invalidate* protocol.

The execution of an application in the lazy release consistency is partitioned into intervals. The intervals of different processes are partially ordered:

1. Intervals in a single process are totally ordered by its program order.

2. An interval of process p precedes an interval of process q if the interval of q begins with the acquire operation corresponding to the release operation that concluded the interval of p.

The lazy invalidation protocol is also a multiple writer protocol. It invalidates the caches of the shared memory according to *write notices*. A *write notice* is created for a written page by the writer when the process executes a release operation. Each write notice lists the information about the page number and the specific interval when the page was modified. There is a diff associated with each write notice. The diff keeps the changes of the page which the process writes since the local copy of the page in the process is made valid [KCDZ94].

A process performs an acquire operation on a variable by sending an acquire request to the last process which performed a release operation on the variable. The releasing process responds to the acquiring process with a set

of write notices. These write notices were created in the intervals preceding the acquiring process's new interval.

The acquiring process invalidates its local copy of a page if there is a write notice for that page and the write notice was not used to invalidate the page before. When the process attempts to access an invalid cache, it first checks whether it has kept all the diffs corresponding to the write notices which invalidated the cache. If not, the process sends messages to some of the processes which created the write notices. The reason why it need not send messages to all of the processes can be shown from the following example. The interval when a write notice is created by process p may precede the interval of another write notice created by process q. Process q must keep process p's diff because process q needs p's diff to make the local cache valid before process q can write to it. The process subsequently accessing the page needs only to send a requesting message to the last writer process, process q. After the accessing process receives all the responses, the accessing process applies the diffs to the cache in the partial order. Then the computation resumes. All the processes keep these write notices and diffs locally until garbage collection is performed.

1.3.3 Diff Creation

Multiple writer protocols allow inconsistent copies of a page to exist in the system. Instead of sending a whole page around, multiple writer protocols send the changes of a page (diffs) to other processes. In this section, we are going to discuss methods to obtain diffs.

Twin Page Method

The *twin page* method [CBZ95] is used to obtain the changes of a written page. Each valid cache is initially protected from write operations. Before a

process writes to the page, the process duplicates a copy of the page in its local memory. When the process executes a release operation, it compares the differences between the after value of the page and the original value of the page to obtain the diff of the page.

Software Write Detection Mechanism

The *software write detection* mechanism [ZSB94] is another method to collect the changes to a written page. In a process, each shared address in shared memory has a dirty bit to indicate whether the variable has been written by the process. After each write to the shared address, the program sets the dirty bit associated with the modified address. A precompiler is used to emit the call to set the dirty bit after each write to the shared memory.

1.3.4 Synchronization Primitives

The synchronization primitives provided by most of software distributed shared memory systems are limited to locks and barriers. Synchronization operations are separate from operations on shared memory and implemented differently.

Barriers

A barrier is usually implemented by a centralized manager. A waiting process sends a message to the barrier manager and awaits the response when it reaches the barrier. After the barrier manager receives all the messages from the waiting processes, it responds to each waiting process. All the processes then resume their computation.

Locks

Munin uses the *probable owner* mechanism to implement locks. Each process maintains its observations about which process might own the lock. If the lock is not available locally, a requesting message is sent to the probable owner. If the probable owner does not have the lock, the probable owner forwards the requesting message to its probable owner. The message is passed along the probable owner chain to the last lock holder. If the lock is free, the last lock holder gives the lock to the requesting process and sets its observation of the probable owner to the requesting process.

TreadMarks uses a distributed queue to implement a lock. Each lock has a specific manager which knows the most recent process, p, requesting the lock. A global waiting queue is maintained. When the manager receives a lock request from a process, q, it forwards this request to p. The manager also sets the most recent process to q. p passes the lock and invalidation information to q when p releases the lock.

1.4 Problems

1.4.1 Synchronization Primitives in Software Distributed Shared Memory Systems

Poor Application Programming Interfaces for Solving Synchronization Problems

The synchronization operations of *release consistency* are restricted to *release* and *acquire*, which are write and read operations on shared memory. The overhead of the strong memory consistency is not only high at run time in NOW but it is hard to program using just release and acquire.

Instead of implementing a strong memory consistency model, most of contemporary software distributed shared memory systems offer higher level

synchronization primitives, such as locks and barriers, and implement them by synchronization managers, see Section 1.3.4.

In order to conform with the definition of release consistency, the developers of distributed shared memory systems usually need to spend some effort in associating locks and barriers with release and acquire operations. For example, getting a lock is an acquire operation and returning a lock is a release operation [Kel95]. However, interpreting a barrier is not as intuitive as a lock. Therefore, using the notions of release and acquire to describe synchronization accesses is problematic.

Moreover, the complexity of using these basic synchronization operations of locks and barriers to solve some synchronization problems is known to be quite complicated for programmers and prone to errors. This is a classic discussion appearing in many operating system text books, for example [Tan92].

Performance Issue

[Car93] and [CBZ95] identify a class of applications that do not perform well on some distributed shared memory systems. For example, a parallel version of the traveling salesperson problem uses a branch-and-bound algorithm to find the shortest path. The algorithm uses a priority queue to store incomplete paths. The priority queue is protected by a lock. As stated in [Car93]:

> "The major source of overhead for these DSM versions was the amount of times spent waiting on the lock protecting the work queues ... These lock waiting times are large because the DSM versions must ship the work queue, a sizable data structure, to the acquiring process before that process can perform any operation on the work queue."

A *function shipping mechanism* was proposed by [Car93] such that the priority queue remains attached to a specific process. Accesses to the priority queue by other processes are performed by remote procedure calls. However, there was no systematic way to incorporate such features into their system.

Message-driven relaxed consistency (see Section 1.2.7) can implement the RPC-server for the priority queue without too much overhead by creating a process which receives requests from other processes in the form of messages. However, this approach inherits the disadvantages of the message passing models, which are difficult for users to program.

1.4.2 Weaknesses of Multiple Writer Protocols

The write-shared data protocol of Munin and the lazy invalidate protocol of TreadMarks are called *multiple-owner* protocols because in the implementations of both systems the reader needs to contact some writers which own a piece of the current data.

Following is a list of problems in multiple-owner protocols.

1. *The need for large number messages to maintain consistency of a page which is written by some processes and read by the processes.*

   When a page is written by n processes and all the processes want to read the same page later (in the case of the write-shared data protocol n processes are caching the same page), it takes $O(n^2)$ messages for all the reader processes to get current version of the page [CK96]. Even though those writers just write a small piece of the page, $O(n^2)$ messages are needed.

2. *Diff accumulation for pages accessed exclusively.*

   In the lazy invalidate protocol processes cannot tell whether two diffs for the same page overlap. If two processes modify a whole array ex-

clusively, the third process needs to obtain diffs generated from both processes to make the page up to date. This scenario is called *diff accumulation* [LDCZ95]. The size of the response to a read request increases when the number of processes increases.

For example, assume processes p, q, r, ... need to read an array and then write to the whole array exclusively. The array is protected by a lock. Process p first reads the initial value of the array and then writes its result into the array. Process q obtains the lock and then gets the diffs of the array, which process p made, from process p. After process r obtains the lock, it needs to get diffs of the array, which process p and q made, from process q. The more the processes there are, the larger the diffs needed to be sent. If there are n processes in the system, the last process need to get $n-2$ versions of the diffs of the array to update the page. Since the diffs created by a process overlap with those diffs created by others, the process only reads the values of the diffs created by the previous lock holder.

3. *Garbage collection.*

   In the lazy invalidate protocol, a process needs to keep its diffs and write notices until garbage collection. Garbage collection is performed by stopping execution of all processes and making active pages current in each process. Both stopping processes from execution and storing diffs locally can also slow down the system (if the diffs consume too much space).

1.5 Contributions

This thesis identifies two weaknesses of contemporary software distributed shared memory systems: poor application programming interfaces for pro-

grammers who need to solve complicated synchronization problems and inefficiencies in the traditional multiple writer protocols. We propose two methods, user-definable high level synchronization primitives and single-owner protocol. We also define a variant of release consistency, release consistency with user-definable high level synchronization primitives ($\mathrm{RC_{HS}}$) to accommodate with high level synchronization primitives.

In order to allow users to define high level synchronization primitives, we provide a paradigm, synchronization class (C++ like class), for users to define their own synchronization primitives and to associate synchronization operations with our memory model. Instead of relying on traditional low level synchronization primitives, such as barriers and locks, user-definable high level synchronization primitives provide a better application programming interface.

The single-owner protocol is a new multiple writer protocol. It reduces the number of messages from $O(n^2)$ messages (the number of messages needed in multiple-owner protocols) to $\Theta(n)$ messages when there are n writers writing to a page and then n readers reading the page. Unlike multiple-owner protocols, in the single-owner protocol garbage collection is performed asynchronously, and there is no diff accumulation.

We also evaluate the tradeoffs between the single-owner protocol and multiple-owner protocols. We have found that in most cases the single-owner protocol uses fewer messages than multiple-owner protocols. But there are some computations which may perform better with multiple-owner protocols. In order to combine the advantages of both protocols, we propose a *hybrid owner* protocol which can be used to increase the efficiency in an adaptive way, with some pages managed by the single-owner protocol and some by a multiple-owner protocol.

Finally, five applications are evaluated using the single-owner protocol

and the lazy invalidate protocol. The performance of these two protocols is compared. We also demonstrate the use of user-definable high level synchronization primitives on one of the applications, and compare its performance against the same application constructed using only low-level synchronization primitives. The result matches our analysis.

1.6 Outline of this Thesis

This chapter has briefly introduced the background and problems of modern distributed shared memory systems.

Chapter 2 presents the paradigm of user-definable high level synchronization primitives and defines RC_{HS}. We also outline the implementation at the end of this chapter.

Chapter 3 discusses the various problems of multiple writer protocols. We also present the new memory protocol, single-owner protocol in this chapter. In addition, the tradeoffs between the single-owner protocol and multiple-owner protocols are discussed. In conclusion a hybrid owner protocol is proposed.

Chapter 4 details the implementation of the single-owner protocol.

Chapter 5 contains the performance evaluation of five different applications.

Chapter 6 discusses related work.

Chapter 7 concludes this thesis and proposes future work.

Chapter 2

Release Consistency with User-definable High Level Synchronization Primitives

Release consistency with user-definable high level synchronization primitives (RC_{HS}) provides a paradigm in which users can define their own synchronization primitives, called *synchronization classes*. RC_{HS} also constrains the execution of synchronization primitives but any synchronization primitive that follows the paradigm can be used in this memory model. The paradigm not only provides a better interface for the programmer to implement synchronization algorithms but also improves the performance of some applications. These high level synchronization primitives are designed for a software distributed shared memory system in a high latency network, where the cost of traditional atomic operations, for example *fetch&add* and busy waiting for the purpose of synchronization is considerable, and has an adverse effect on the performance of the computation.

2.1 Programming Model

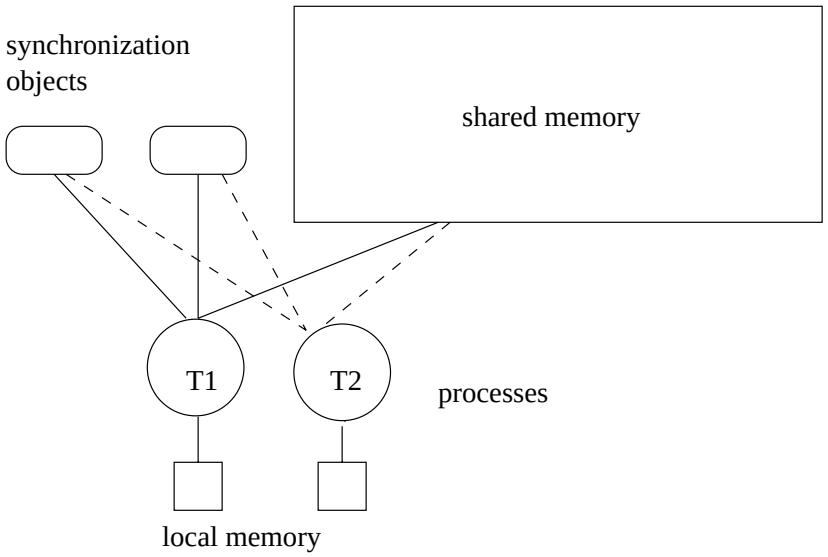

Figure 2.1: Programming model

There are two types of shared objects that processes use to communicate with each other in RC_{HS}: shared memory and synchronization objects (see Fig. 2.1). The shared memory consists of a contiguous memory array in virtual memory. Two types of operations are allowed in shared memory, read and write. Anything a process writes in shared memory may be visible to other processes. In addition to shared memory, processes can also communicate with each other via synchronization objects. Each process accesses synchronization objects only by calling operations defined in synchronization classes. Synchronization objects can not access other shared objects.

Each synchronization operation may be annotated with one of following attributes, *release*, *acquire*, *acquire_release*, or *release_acquire*. These attributes are used to define the visibility of the values in shared memory. Informally,

the annotation *release* may be thought of as meaning that the process "puts" its visible shared memory writes onto the synchronization object, and *acquire* may be thought of as meaning that the process "gets" the visible writes from the synchronization object. For example, suppose that process p writes to shared variable X and then executes an operation provided by synchronization object S with attribute *release*. S can see the newly written value of X (even though S is not allowed to access other shared objects). If process q subsequently executes an operation of S with attribute *acquire*, q obtains what S can see. So q can read the new value of X which p has written. Similarly, *acquire_release* and *release_acquire* may be thought of as a combination of the two. In the context of following discussion, *release operations* are operations with attribute *release*, *acquire_release* or *release_acquire*. *Acquire operations* are operations with attribute *acquire*, *release_acquire*, or *acquire_release*.

We consider an example of a simplified version of the producer and consumer problem [Tan92]. Products produced by producers are stored in shared memory. Consumers read products from the shared memory. We assume there is infinite memory to store products. We use a synchronization object called `buffer` to coordinate producers and consumers. The definition of the synchronization object is shown in Fig. 2.2.

The synchronization object, `buffer`, works as a server which keeps pointers of available products for consumers. After a producer writes its product to shared memory, it calls the method `PutItemPtr(...)` and passes the pointer of the product in shared memory to `buffer`. Since the product in shared memory needs to be visible to other consumers, `PutItemPtr(...)` is annotated with attribute *release*. A consumer acquires the pointer of a ready product by calling `GetItemPtr(...)`. In order to read the part of the shared memory written by producers, `GetItemPtr(...)` is annotated with attribute *acquire*.

```
SyncClass Buffer{
public:
  Buffer();
  ~Buffer();
  release void PutItemPtr(ItemPtr item_ptr);
  acquire ItemPtr GetItemPtr();
private:
  Queue_class<int> *empty; /* keep process ids when the buffer is empty */
  Queue_class<ItemPtr> *buffer;  /* keep products' pointers*/
  ......
};

Buffer_class *buffer;

process_Consumer(){
  ItemPtr p;

  while ((p = buffer->GetItemPtr()) != NULL) consume(p);
}
process_Producer(){
  ItemPtr p;

  while ((p = produce()) != NULL) buffer->PutItemPtr(p);

}
initGlobalEvironment(){
  buffer = new Buffer;
}
```

Figure 2.2: Synchronization Class

Our system provides two basic synchronization classes, semaphores and barriers. Semaphores have two operations, `P(k)` and `V(k)`, where `k` is the number to increase or decrease the counter of the semaphore. `P(k)` and `V(k)` are annotated with *acquire* and *release* attributes respectively. The `P` and `V` operations of binary semaphores usually correspond to operations on locks, which are used to protect a critical section. When a process gets a lock using the `P` operation, the *acquire* annotation specifies that the previous lock holder's writes become visible to the process. Similarly, when the process leaves the critical section using the `V` operation, the *release* annotation specifies that subsequent processes will see its writes.

A barrier has one operation, `WaitForBarrier(proc)`, where `proc` is the number of waiting processes. The attribute of `WaitForBarrier(proc)` is *release_acquire*. Intuitively, the *release_acquire* attribute can be thought of specifying a *release* annotation and then an *acquire* annotation. In this case, the calling process will be putting its writes (i.e. making them visible) to the synchronization object when it calls `WaitForBarrier(...)` and will be getting other writes (i.e. collecting currently visible writes) from the synchronization object when the call returns. The synchronization object of the barrier does not respond to the calling processes until k processes execute `WaitForBarrier(k)`. Therefore, the waiting processes can see all the writes other waiting processes made in shared memory before they executed `WaitForBarrier(...)`.

2.2 Synchronization Objects and Synchronization Classes

Defining synchronization classes is very similar to defining regular classes in the C++ programming language. Instead of *class* in C++, synchronization classes start with *SyncClass*.

Synchronization classes do not have the inheritance properties of regular object oriented languages. The operations declared in the public section of the synchronization class are the only operations which can be called by processes. Each operation can have at most one parameter. In addition, each of the public operations may be tagged with a synchronization attribute of either *release*, *acquire*, *release_acquire*, or *acquire_release*. These attributes are used to define the visibility of processes' writes to shared memory. The private section of the synchronization class is used to define procedures and data structures in synchronization objects.

Synchronization objects can be accessed only by calling operations provided by them. The computation of a synchronization object is like that of a server servicing a remote procedure call called by processes. The execution of a synchronization object is sequential. The synchronization object may decide not to reply immediately to the caller and receive other requests. The caller is stalled until the synchronization object sends a response back.

2.3 Execution of a Program

In order to simplify the discussion, we are going to use a directed acyclic graph (DAG) to specify the execution of a program.

2.3.1 Phases and Events

The execution of a process or a synchronization object is sequential and consists of phases. Performing a synchronization operation is an event of a process. Receiving a request from a process and replying to a process are events of a synchronization object. The computation between two events, including the ending event, is a phase. The execution of a process or a synchronization object is a sequence of phases in program order.

Each phase of a process is identified by a unique time stamp. The time stamp of the initial phase is 1. The phase with time stamp i is followed by the phase with time stamp $i + 1$. The phase of a synchronization object starting with a requesting event from a process is called a *receiving phase*, even though the requesting event is not in the phase. The phase of a synchronization object ending with a replying event to a process is called a *replying phase*.

2.3.2 The Execution in Parallel Computation

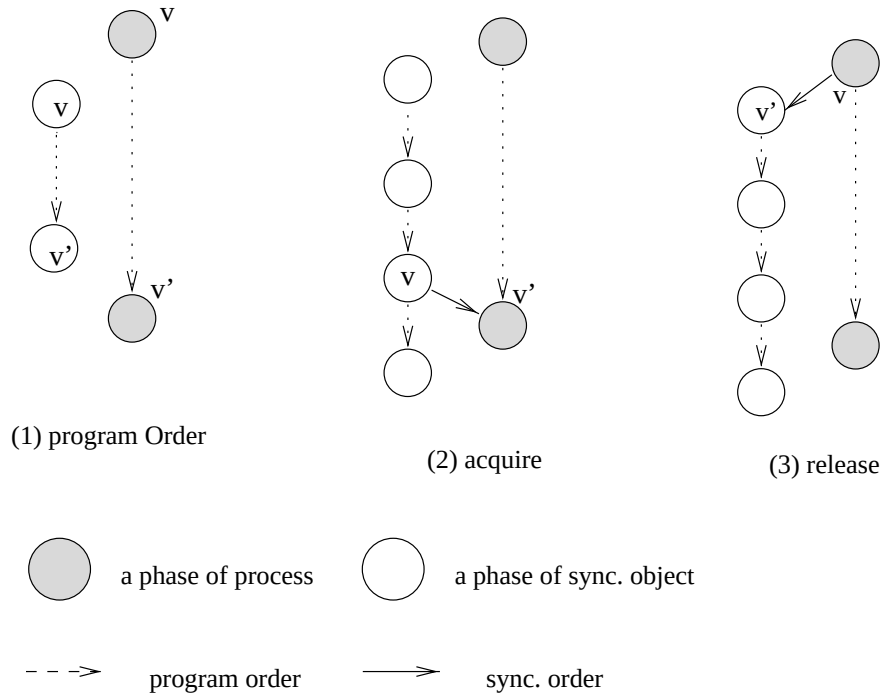

Figure 2.3: A directed acyclic graph of a computation.

The execution of a parallel program can be represented by a directed acyclic graph, $G = \{V, E\}$, where V is the set of phases. The visibility of a written value in shared memory is defined by G. There is an edge $e_{vv'}$ from

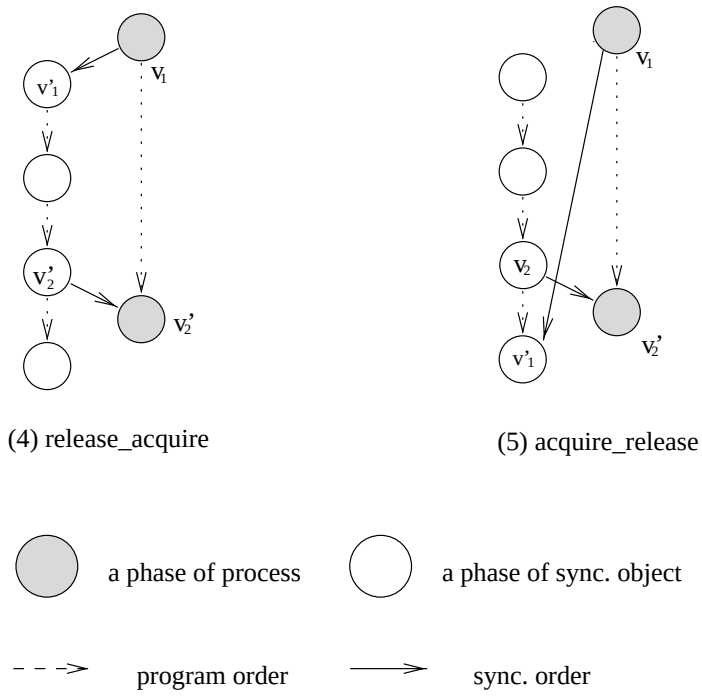

Figure 2.4: A directed acyclic graph of a computation.

vertex v to v' if and only if one of following conditions holds.

1. v' and v are phases from the execution of the same process or synchronization object and v' immediately follows v in program order. See Fig. 2.3(1).

2. A process invokes a synchronization operation with the *acquire* attribute. v is the replying phase of the synchronization object, and v' is the phase after the process invokes the synchronization operation. See Fig. 2.3(2).

3. A process invokes a synchronization operation with the *release* attribute.

v is the phase ending with the synchronization operation, and v' is the receiving phase of the synchronization object. See Fig 2.3(3).

4. A process invokes a synchronization operation with the *release_acquire* attribute. This case involves two edges.

   (a) v is the phase of the process ending with the synchronization operation, and v' is the receiving phase of the synchronization object (see edge $e_{v_1 v'_1}$ in Fig 2.4(4)).

   (b) v is the replying phase of the synchronization object and v' is the phase after the process invokes the synchronization operation (see edge $e_{v_2 v'_2}$ in Fig 2.4(4)).

5. A process invokes a synchronization operation with the *acquire_release* attribute. In this case there are two edges:

   (a) v is the phase of the process ending with the synchronization operation, and v' is the phase immediately after the replying phase of the synchronization object (see edge $e_{v_1 v'_1}$ in Fig 2.4(5)).

   (b) v is the replying phase of the synchronization operation, and v' is the phase after the process invokes the synchronization operation (see edge $e_{v_2 v'_2}$ in Fig 2.4(5)).

A phase p' is reachable from p, denoted by $p \prec p'$, if there is a path from p to p'. (We can also say phase p reaches phase p'.) Two phases are concurrent if there is no path between them. *Competing accesses* are two operations accessing the same shared variable in concurrent phases and one of the operations is write.

Assume two operations o and o' are executed in two phases, p and p' respectively. o' is reachable from o (noted as $o \xrightarrow{\text{DAG}} o'$) iff any of following conditions is satisfied : (1) $p = p'$ and o is executed before o', or (2) $p \prec p'$.

Visibility of a Written Value on a Shared Variable

A written value of a write operation, w, on shared variable x in phase p is visible to a read operation on x in phase p' if and only if one of following situations holds

1. if $p = p'$, the write operation is the last write operation on x before the read operation.

2. if $p \neq p'$ and p' is reachable from p, w is the last write on x in phase p and there is no other phase on the path from p to p' which has write operation on x.

3. p and p' are concurrent.

The set of written values visible to the read operation *op* on X is called the *visible set* of X.

From the programmer's point of view, an *acquire* action takes place when a synchronization object replies to a requesting process. All the updates in shared memory that are visible to the synchronization object before the synchronization object replies are also visible to the process after the process receives the response from the synchronization object.

The time instance when *release* acts depends on how conservatively the updates in shared memory are expected to be propagated. If the attribute *release_acquire* or *release* is used, the updates of the requesting process are visible to the synchronization object when the object receives the request. If the operation is annotated with *acquire_release*, the updates are made visible

to the synchronization object after the object replies to the process. The attribute *acquire_release* can be used for atomic updates to the synchronization object. For example, in the consumer producer problem (see Sec. 2.1), if the queue `buffer` in the synchronization object `buffer` is full, the producer needs to be suspended until some products are taken by consumers. The updates in shared memory by the producer do not have to be seen by others until the products are stored in the queue `buffer`. The function `PutItemPtr(...)` may be annotated with the attribute *acquire_release*. A synchronization operation can have no attribute. The updates of the process are not visible to such a synchronization object when it executes the synchronization operation.

2.4 Release Consistency with User-definable Hight Level Synchronization Primitives

In traditional release consistency, synchronization accesses read and write shared memory to enforce memory consistency. Release Consistency with user-definable hight level synchronization primitives ($\mathrm{RC_{HS}}$) does not have synchronization accesses to shared memory but instead synchronization operations provided by synchronization objects. Synchronization operations are annotated with attributes which denote how visible writes are passed explicitly. We will first describe the memory consistency using our notation. Later in the section we will compare $\mathrm{RC_{HS}}$ to other release consistency models.

2.4.1 View

Shared memory consists of a set of shared variables. In order to simplify the discussion, we assume without loss of generality that each variable an integer. Let X be a variable. A visible set of X to a process is a set of values the process may get when it reads X. V_X will denote the visible set of X. Initially $V_X = \{0\}$ for all processes. The collection of V_X for all X in shared

memory forms a view of the process. The initial view of a synchronization object is a collection of $V_X = \{0\}$ for all X in shared memory.

2.4.2 Merging Views

A process' visible set of shared variable X changes dynamically during the execution between synchronization operations, issued by the process itself. V_X may change during this period in two cases. The first case occurs when another process writes b to X, causing V_X to become $V_X \bigcup \{b\}$. In the second case, when the process itself writes a to X, V_X is set to $\{a\}$. However, after the process writes to X, V_X may change if another process writes to X. If the process attempts to read X, any value in V_X is a legal returned value. Executions with race conditions are allowed in $\mathrm{RC_{HS}}$.

The visible set of variable X to a synchronization object changes only when the object merges its view with a process's view. Let V'_X be the visible set of variable X to the synchronization object and V_X is the visible set from a process. V'_X gets updated if V_X is not equal to V'_X. For each value a in $V_X \bigcup V'_X$, a is associated with a set of write accesses, OP_a, where each op_a in OP_a writes a to X. The merging process for the synchronization object picks value a as an element of the newly merged visible set if there is no other value b in $V_X \bigcup V'_X$ such that $op_a \xrightarrow{\mathrm{DAG}} op_b$ for any op_a and any op_b.

For example, assume $V'_X = \{1, 2\}$ and $V_X = \{3, 4\}$, and $op_1 \xrightarrow{\mathrm{DAG}} op_3$ for all op_1 in OP_1 and all op_3 in OP_3. The newly merged visible set becomes $\{3, 2, 4\}$. Processes follow the same merging procedure as synchronization objects.

When a process executes a *release* operation of synchronization object S, it also passes its current view to S. S then updates its view by merging its own view with the process's view. When the object responds to a request for an *acquire* operation, S passes its view to the requesting process. Then the

process updates its view by merging its current view with S's view.

2.4.3 When Attributes Become Effective

The *release* attribute of a synchronization operation becomes effective when the synchronization object merges its view with the requesting process's view. The *acquire* attribute of a synchronization operation becomes effective when the synchronization object responds to the requesting process and the process merges its view with the object's view.

The time instance when the synchronization object updates its view and gives its view to the requesting process depends on the semantics and the attribute of the synchronization operation.

If the synchronization object receives a request with a *release* or *release_acquire* attribute, the object updates its current view immediately. If the object receives a request with an *acquire_release* attribute, the object merges its view with the requesting process's view after it responds to the process. If the process executes an *acquire* operation, it merges its view with the view of the synchronization object when it receives the reply. When a synchronization object responds to an *acquire* operation it always sends its current view to the requesting process.

2.4.4 Conventional Notation

Even though this new memory consistency model does not have synchronization accesses on shared memory, we can still use the terminology of conventional release consistency to describe it. Each synchronization object is considered as a special shared variable. When the *release* attribute of a synchronization operation becomes effective, a *release* access to the synchronization object is performed by the requesting process. Similarly, when the *acquire* attribute of a synchronization operation becomes effective, an *acquire*

access to the synchronization object is performed by the requesting process. *Release* and *acquire* accesses to the same synchronization object are executed in total order. We may use the traditional notation for release consistency to define $\mathrm{RC_{HS}}$. This notation is only suitable for programs without race conditions.

Definition 6 *A system is* $\mathrm{RC_{HS}}$ *iff*

1. *Before an ordinary load or store access is allowed to perform with respect to any other process, all preceding acquire accesses must be performed.*
2. *Before a release access is allowed to perform with respect to another process, all ordinary load and store accesses must be performed with respect to that process.*
3. *Synchronization accesses follow sequential consistency. The order of synchronization accesses depends on the semantics of synchronization objects.*

We may mimic *release* and *acquire* operations of the release consistency on shared memory by creating a synchronization class, whose operations are `release void put(Parameter parameter)` and `acquire Parameter get()`. The synchronization object also keeps a local variable with type `Parameter`. When the synchronization object receives a request for the function `put(...)`, it stores the parameter in its local memory. When the synchronization object receives a request for function `get`, it responds to the requesting process with the value in its local memory.

2.5 Implementation

2.5.1 System Architecture

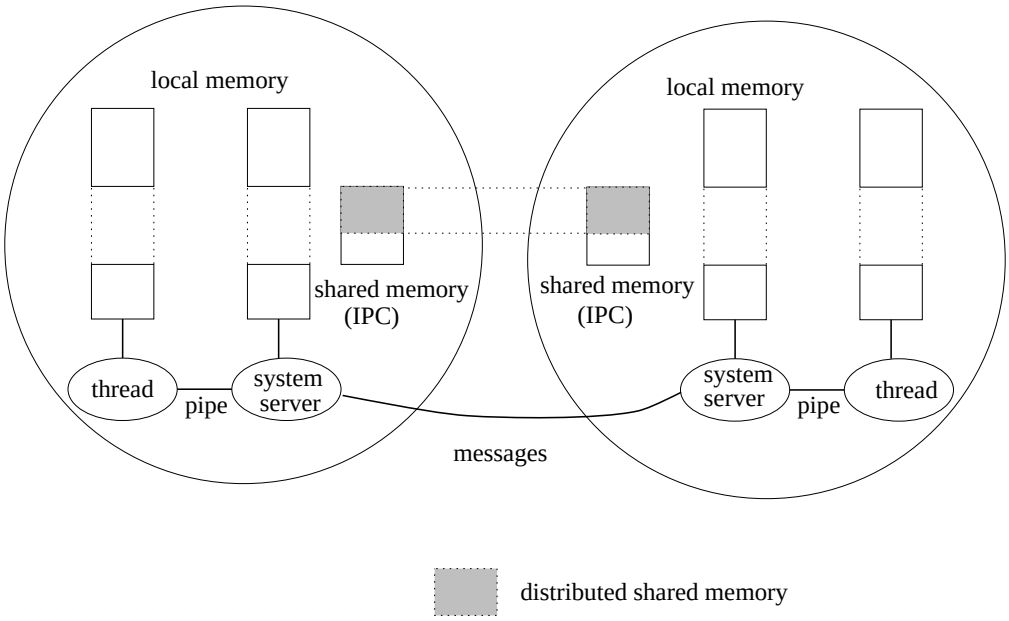

Figure 2.5: Architecture

The system consists of several machines. Each has its own processor and local memory. The machines are connected by a local network. We will use the term "processor" for the machine from now on. Distributed shared memory is located in the same contiguous space in each processor's virtual memory. Any access to that piece of memory is monitored and controlled by the system.

In the current implementation, each processor has two processes in charge of the computation (see Fig. 2.5). One process, the thread, is used for the computation of applications. The other process, the system server, serves the requests for synchronization and memory consistency. These two processes communicate with each other via pipe and shared memory provided by IPC [Ste90]. System servers communicate with each other via messages.

2.5.2 Synchronization Objects

Synchronization objects are embedded in system servers. Each synchronization object has a unique id number and each method in the public section of the synchronization class is assigned an id number. When a thread accesses a method of a synchronization object, the thread itself sends a message to its local system server notifying it of the object id and the operation it is accessing. The local system server processes the request if the synchronization object resides locally. Otherwise, it forwards the request to the remote system server where the object resides.

In our current implementation, creating code for synchronization servers is done manually. However, it is not difficult for a preprocessor to do this. *SyncClass*'es are written by programmers. Two classes are generated, one for processes and the other for system servers. The class for threads contains only the methods declared in the public section of the synchronization classes. The body of each method composes a message and sends it to the system server. The method waits for the response from the server if necessary.

The class for system servers is almost the same as the class defined by programmers. We insert three new methods into the class, `GetCliId()`, `SyncReply(process id, result)`, `Master( caller's id, operation id, parameter)`. After the system server receives a synchronization request, it passes the parameter to the synchronization object's `Master(...)` function, which calls the public method mapped to the operation id. When the object decides to reply to a requesting thread, it calls `SyncReply(process id, returned value)`. `SyncReply(...)` composes the replying message and sends it to the calling system server.

2.5.3 Memory Protocol

We employ a multiple writer protocol to implement our shared memory system. Each processor may cache a page in shared memory. The various caches of a page are allowed to be inconsistent. A cache is invalid if the value of a variable in the cache is not in the variable's visible set. A modified lazy invalidate protocol [KCZ92] is used to propagate invalidation information.

A page invalid to the local thread is protected by the operating system. When the local thread attempts to read the page, a segment fault is raised. The signal handler then sends a read request to the system server. After the system server obtains a valid copy of the page, the local thread resumes its execution.

Write Notices and Invalid Pages

Following the lazy invalidate protocol, when a thread issues a *release* operation immediately after a phase, it generates a set of write notices. Every write notice in the set indicates a page that has been written since the last phase when the thread executed a *release* operation. A write notice consists of the writer's identification number, the written page number and the time stamp of the phase.

Propagating Write Notices

Systems servers follow the edges of the DAG described in Sec. 2.3.2 to propagate write notice sets. When a thread performs a *release* operation, the system server sends sufficient write notice sets to the system server where the synchronization object is. When a synchronization object responds to a thread on behalf of an *acquire* operation, the object requests its local system server to send sufficient write notice sets to the system server where the re-

questing thread is. In contrast, the original lazy invalidate protocol does not have the concept of synchronization objects. Write notices are propagated from threads to threads.

Invalidating a Stale Page

A thread invalidates pages after it executes an *acquire* operation. After its system server receives the response from the synchronization object, the server finds all the pages which were valid at the last phase but will be invalid at the next phase because of newly arriving write notices. The server informs the thread about the invalid pages. Then the thread sets protection on those pages. There are several ways for the run time system to obtain a valid copy of the page. We will postpone until Chapter 4 the description of how a valid copy is obtained.

Chapter 3

Multiple Writer Protocol

In the beginning of this chapter, we will discuss the weaknesses of conventional multiple writer protocols. We then propose a new multiple writer protocol, called *single-owner* protocol, which can outperform conventional multiple write protocols in some cases. We will also show in some cases conventional multiple writer protocols can outperform the single-owner protocol. In order to exploit two different protocols, a *hybrid owner* protocol is proposed.

3.1 False Sharing in Distributed Shared Memory Systems

It has been recognized that *false sharing* is one of the key impediments to high performance in DSM systems. Programmers, address their data on the *logical variable* basis. The underlying operating system handles the data in the *physical page* basis. Consider a simple example of two variables x and y residing on a single page. If one processor works on x and another on y, as far as the operating system is concerned they both work on the same physical variable, the page in which x and y are both stored. In the worst case, the page is *shuttling* between the two processors, severely delaying the progress of the computation. A protocol behaving as described above, is generally

referred to as a *single writer* protocol.

One of the ways to reduce such shuttling is to employ a *multiple writer* protocol in conjunction with a specific memory consistency model [CBZ95, KCDZ94]. This protocol allows more than one processor to write to the same page at the same time. However the page does not shuttle among the processors. Instead of passing a copy of the page nondeterministicly among processors, each processor collects its diffs [Car93, ZSB94] from a written page and then propagates them to other processors. This is an effective way of improving performance in many cases.

We will show that this approach has certain weaknesses and therefore has to be used with caution.

3.2 Excess Messages Caused by Multiple-owner Protocol

Figure 3.1: Heat flow problem

We describe the problem by using the example of a simple canonical application, that of the *heat flow* problem. Our example is grossly simplified in order to highlight the underlying problem and our approach to solving it.

We are given a uniform square plate with time-invariant values of the temperature on the boundary and initial zero temperature in the interior. Using a standard iterative technique, we are to calculate the steady state temperature. The plate is partitioned into squares, each represented by a location in an array T. The computation proceeds in *time steps*. In each step the next value of $T[i,j]$ is computed based on the current values of $T[i-1,j]$, $T[i,j+1]$, $T[i+1,j]$, $T[i,j-1]$, and $T[i,j]$. When the *stopping condition* of the form $|\text{new } T[i,j] - \text{old } T[i,j]| < \epsilon$ for all i and j holds, the steady state temperature is assumed to have been reached.

The parallel DSM version employs some number of concurrently executing threads, say n. Each thread executes in a separate processor. It is responsible for computing the temperature of a horizontal "slice," as shown in Fig. 3.1. When a global stopping condition is reached, the execution is completed.

For our discussion assume that there is a *stopping vector* S of n elements. At each time step, after thread k computed the new values of its slice, it checks whether the stopping condition is satisfied for its slice. If yes, it writes 1 in $S[k]$, otherwise it writes 0 in $S[k]$. At the beginning of a step, it finds out whether all threads "agree" to stop. This is indicated by all locations of S being 1. Thus all threads complete after the same number of steps.

Let us examine now the performance of the algorithm in a *multipe-writer* protocol. For simplicity, we can assume that a row of the matrix T in general fits in a single page. The vector S in general will fit in a single page.

First, at the end of each step (other than the final one), each thread needs to obtain from each of its neighbors the values of one row. (In our example, Fig. 5.2, thread k, in order to compute $T[i,j]$, needs to obtain the value of $T[i,j-1]$ as computed by thread $k-1$). Thus each thread needs to send about 2 messages (although this could be higher, though still a small constant—we will continue with the value of 2 in our example). So the total number of

messages required to accomplish this is (say) $2n$.

Furthermore, each thread needs to notify all the other threads whether steady state has been reached in its slice (adaptive notification is not practical due to the lack of global knowledge). As is currently generally done, this will require $2n(n-1)$ messages. In the underlying implementation of lazy invalidate protocol [KCDZ94], each thread k will in effect contact each of the threads 0, 1, ..., $k-1$, $k+1$, ..., and $n-1$. Each will respond by sending the single entry of array S it wrote. For write-shared data protocol [CBZ95], each writer aggressively propagates updates to threads currently holding S. So each thread needs to send the single entry of S that it wrote, to all other $n-1$ threads. We thus see that the communication costs in each step are swamped by what seems like a minor part of the overall computation.

Since in these protocols a page can be owned by many threads and a page owner needs to contact other page owners to get current updates, we call them multiple-owner protocols.

3.3 Single-owner Protocol

We propose a single-owner protocol to solve this problem. Each page will have a "home processor", which will be referred as the *owner* of the page. All the modifications to the page will be sent to the owner of the page, and each thread will request the new values from the page's owner. To apply this protocol to our example, the pages of the matrix T are owned by the processors computing them, thus minimizing communication. (If a page contains elements spanning two slices, then one "non-owner" processor will be updating the elements in the page.) Using this protocol the number of messages is greatly reduced.

Boundary information is exchanged between processors resulting in cn (for a small constant c) messages. Vector S is owned by one (or maybe two

processors if S spans more than one page). Thus updating it takes n messages and the threads require n messages to read the new value of S. The total number of messages is linear in n with a small multiplicative constant, and does not grow quadratically with the number of processors, as in the previous description.

Let us return to the synchronization employed in this algorithm. There is one synchronization per step. Machines *wait* until they finish a step, read the updates they need and proceed to the next step.

One could try to reduce the number of messages from quadratic to linear while still employing the standard protocols. For instance, one could have a *centralized stopping detector* by employing a single thread to obtain all the values of S, determine if all of them are 1, and then notify the slices-computing threads whether the global steady state has been reached. In Munin this will still require a quadratic number of messages, but in TreadMarks, this can be done using a linear number of messages. However, this centralized stopping detection suffers from the following drawback: it requires an additional synchronization point per step.

We note that the above was a simplified example showing how some aspects of the current protocol for the management of shared pages need to be further developed to improve performance.

3.4 Multiple-owner Protocol

However, in some cases multiple-owner protocols can still outperforms the single-owner protocol. Quicksort can be used for illustration.

The standard parallel version of quick sort uses an array in shared memory to keep keys and a queue to keep the indexes of the boundaries of the subarrays that have not been partitioned. The queue is protected by a lock so that only

one thread can access it at any time. Each thread gets from the queue a pair of indexes that define the subarray for it to "quicksort." If the size of subarray is below some threshold, it sorts the subarray locally. Otherwise, it finds the pivot in the subarray and partitions the array into two subarrays. The thread puts the indexes defining the boundaries of one of the subarrays into the queue. Then the thread starts working on the other subarray (in a recursive manner).

Note, that in each step (other than the first) a thread reads data it has previously written—it has generated the subarray which is its input for the current step. In fact, the thread is the only writer and reader of the subarray in any given step. For the single-owner protocol, employing release consistency, each thread has to send out the updates to the subarray to page owners, even though the major part of the subarray is only read by itself. For the multiple-owner protocol [KCDZ94], modifications are kept locally until acquiring threads send requesting messages. Thus, no messages are passed until any thread gets the indexes of a subarray from the queue.

3.5 Tradeoffs Between Single-owner Protocol and Multiple-owner Protocol

In this section, we are going to explore different scenarios when multiple-owner protocols use fewer messages than the single-owner protocol. This happens in very limited cases. Message counts can be a fair estimation for the performance of software distributed shared memory systems because sending/receiving a message imposes large overhead in a high latency network.

Assume there are w writer threads writing to a page and r reader threads reading that page later. If writers are also readers, the total number of messages required for all the readers to get updates produced by the writers is

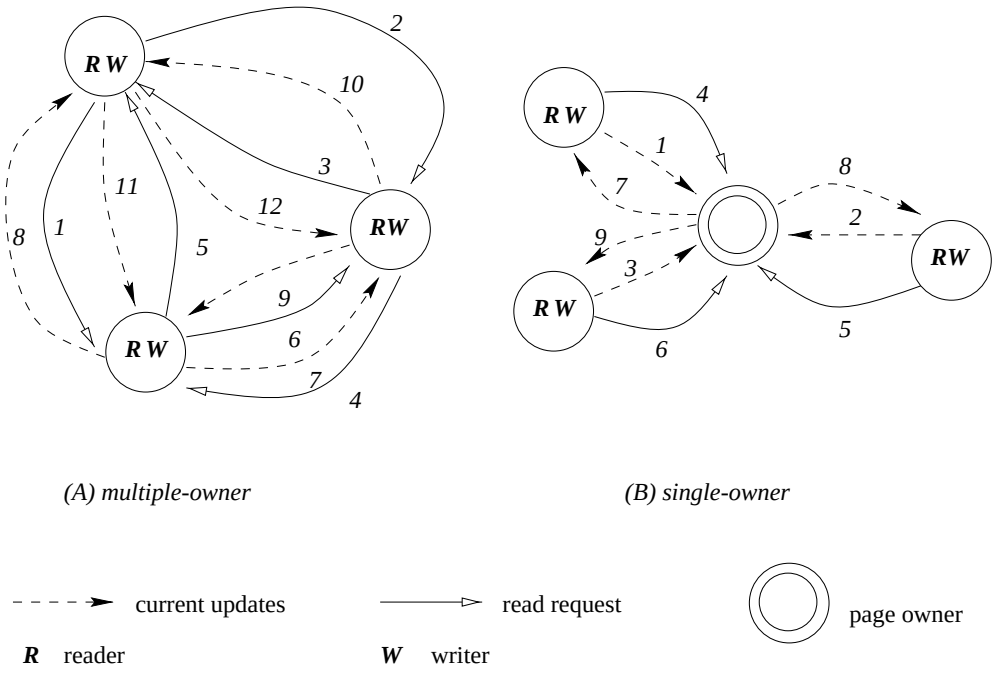

Figure 3.2: Messages needed to get three readers updated with three writers in the system

$2r(w-1)$ for a multiple-owner protocol in the best case. (These formulas assume the lazy invalidate protocol. For write-shared data protocol, more messages are required. In such case, the set of readers the protocol considers is a superset of actual readers.) In the single-owner protocol, at most $2r+w$ messages are required.

Therefore we examine the values of r and w when $2r(w-1) \geq 2r+w$. The equation is always true when $r=1$ and $w \geq 4$ or when $r>1$ and $w \geq 3$. As long as r and w satisfy one of above conditions, the single-owner protocol uses fewer messages than the multiple-owner protocol. Fig. 3.2 illustrates the case when three writers and three readers are in the system. In the multiple-owner protocol it takes 12 messages to make all the readers updated.

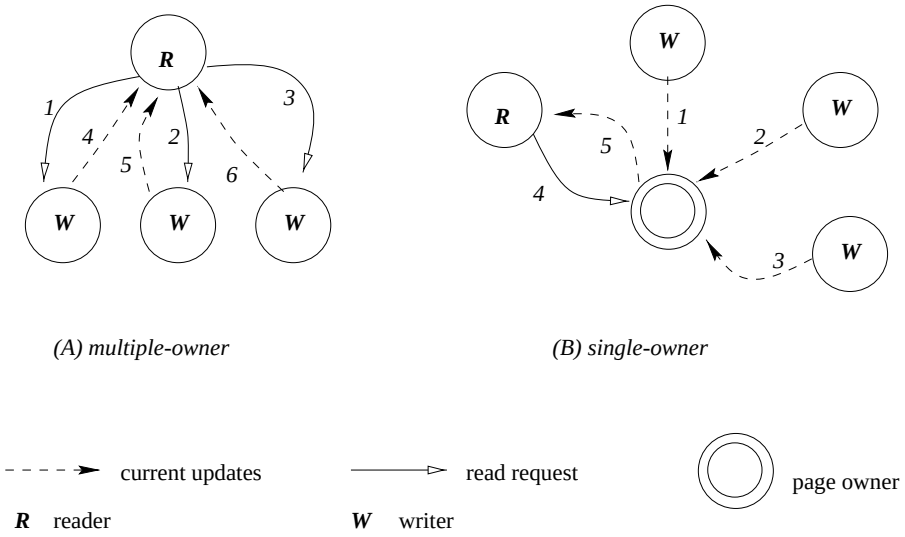

Figure 3.3: Messages needed to get two readers updated with one writer in the system

Since the page owner collects all the updates, the single-owner protocol just takes 8 messages. So, when $r \geq 1$ and $w \leq 2$ or when $r = 1$ and $w = 3$, the multiple-owner protocol may use fewer messages.

We are going to discuss these cases in detail:

1. No reader ($r = 0$).

   If there is no reader, in the multiple-owner protocol the diff stays in writers' processors. No messages are sent. In the single-owner protocol, the behavior depends on where the writers reside. If there is only one writer and the writer is at the page owner, no messages are sent. However, if there is more than one writer and one of the writers resides at the page owner, $w - 1$ messages are sent. w messages are sent if none of the writers resides at the page owner. The multiple-owner protocol always uses fewer messages in this case.

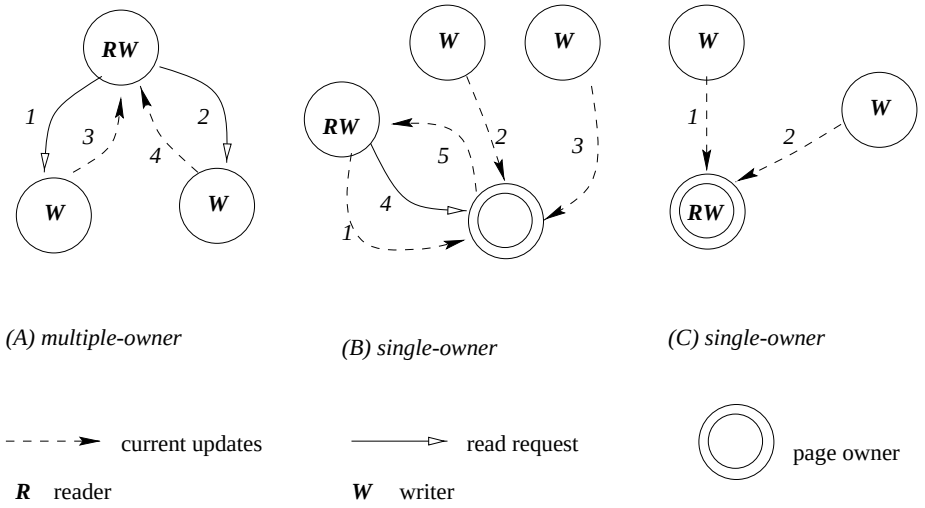

Figure 3.4: Messages needed to get two readers updated with one writer in the system

2. There are three writers and only one reader ($w = 3$ and $r = 1$).

   (a) Assume none of the writers are in the reader's processor (i.e the reader is not a writer. See Fig. 3.3.). In the multiple-owner protocol, the reader needs to contact all the writers to get all the updates. This takes 6 messages (see Fig. 3.3(A)). In the single-owner protocol, all the writers send all the diffs to the page owner and then the reader gets all the diffs from the page owner (see Fig. 3.3(B)). This takes 5 messages. The single-owner protocol uses one message fewer than the multiple-owner protocol.

   (b) Assume the reader resides in the same processor as one of the writers (i.e the reader is also a writer. See Fig. 3.4.). The multiple-owner protocol requires the reader to send read

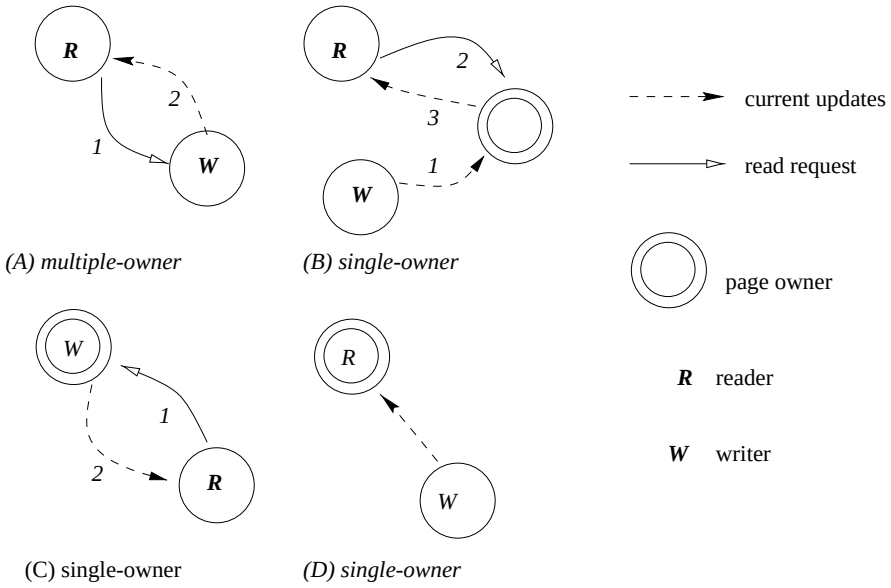

Figure 3.5: Messages needed to get two readers updated with one writer in the system

requests to the other two writers and gets diffs from them (see Fig. 3.4(A)). This takes 4 messages. In the single-owner protocol, if none of the writers are at the page owner, all the writers send their diffs to the page owner and then the reader retrieves all the diffs from the page owner (see Fig. 3.4(B)). This takes 5 messages. However, when the reader is at the page owner, it takes only 2 messages for all the writers to send the updates to the page owner (see Fig 3.4(C)). The reader does not send any read request since it is at the page owner.

3. There is at least one reader and exactly one writer ($r \geq 1$ and $w = 1$).

   Assume the writer is not a reader. In the multiple-owner protocol,

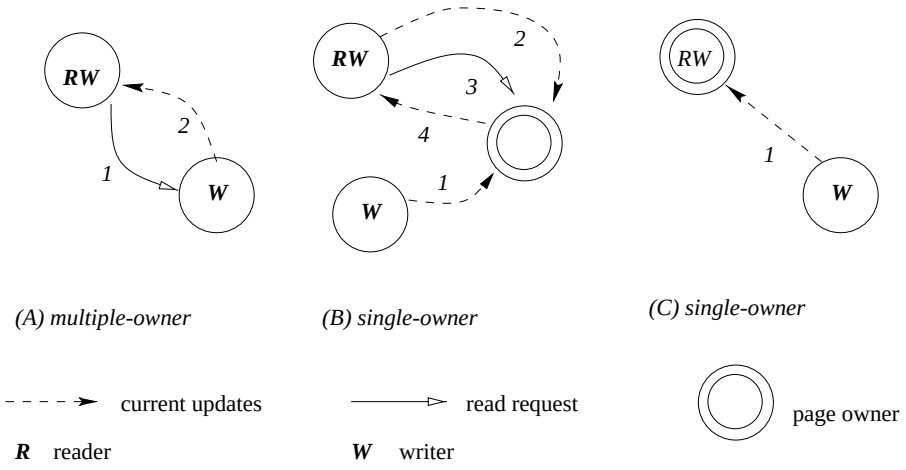

Figure 3.6: Messages needed to get two readers updated with one writer in the system

all the readers send read requests to the writer and get the current updates from it. This takes $2r$ messages (see Fig. 3.5(A)). In the single-owner protocol, if neither the writer nor the readers are at the page owner, the writer sends an extra message to the page owner and then all the readers get the current updates from the page owner (see Fig. 3.5(B)). This takes $2r + 1$ messages. If the writer is at the page owner (see Fig. 3.5(C)), it saves one message to update the page owner. In this case, the single-owner protocol takes $2r$ messages. If one of the readers is at the page owner (see Fig. 3.5(D)), that reader does not need to send a read request to the page owner but obtains the current data locally. This takes $2r - 1$ messages in total.

4. There are at least one reader and two writers ($r \geq 1$ and $w = 2$).

   (a) Assume neither the readers nor the writers reside in the same pro-

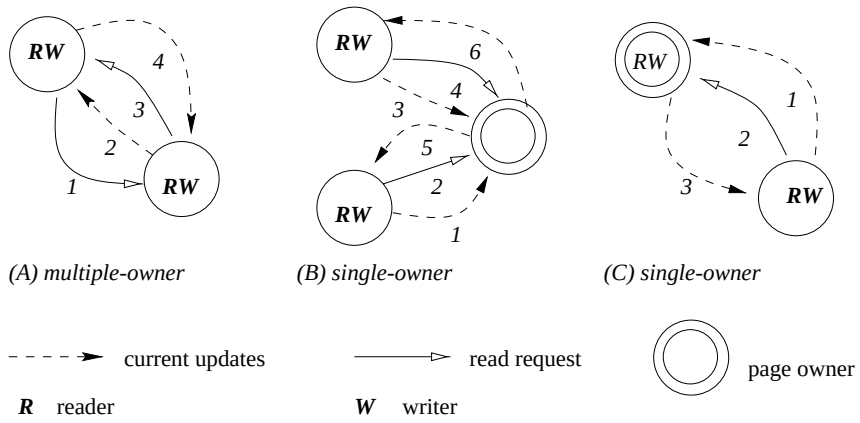

Figure 3.7: Messages needed to get two readers updated with one writer in the system

cessor. In the multiple-owner protocol, all the readers contact both writers to obtain current updates. This takes $4r$ messages. In the single-owner protocol, in the worst case if the readers and the writers are not at the page owner, it takes $2r + 2$ messages. In such case, the single-owner protocol always uses fewer messages when $r > 1$.

(b) Assume one of the readers is also a writer (i.e. a reader and a writer reside in the same processor.) In the multiple-owner protocol, that reader saves 2 messages to get the current updates because it itself is a writer (see Fig. 3.6(A)). This takes $4r - 2$ messages. In the single-owner protocol, in the worst case if neither the readers nor the writers are at the page owner, the reader (who is also a writer) still needs to contact the page owner to obtain the other writer's updates. This takes $2r + 2$ messages (see Fig. 3.6(B)). The multiple-owner protocol uses fewer messages only when there

is just one reader. The multiple-owner protocol takes 2 messages. The single-owner protocol takes 4 messages. However, if the page owner resides in the processor where the reader/writer resides, that reader just takes one message to get updates because the other writer sends the diff to the reader automatically (see Fig. 3.6(C)).

(c) Assume there are at least two readers and two of the readers are also the writers. In the multiple-owner protocol, all the other readers, which are not writers, contact both writers to get all updates. The readers which are also the writers just send one read request to each other. This takes $4r - 4$ messages (see Fig. 3.7(A)). If neither the readers nor the writers are at the page owner in the single-owner protocol, it takes $2r + 2$ messages to update all the readers (see Fig. 3.7(B)). So when there are only two readers, the single-owner protocol uses more messages (6 messages) than the multiple-owner protocol (4 messages). But when one of the readers (also a writer) is at the page owner, the reader saves 3 messages for updating the page owner and getting updates (see Fig. 3.7(C)). So the single-owner protocol takes only 3 messages. In such case, the single-owner protocol uses fewer messages.

From the above discussion, there are only a few cases where the multiple-owner protocol uses fewer messages. However, in these cases the single-owner protocol may reduce the number of messages by assigning the page owner to a processor where one of the readers resides. The application mentioned in Sec. 3.4 shows that sometimes it is impossible to know which threads are the readers and writers of a page in advance. Whether a thread becomes the reader or the writer of a page depends on the state of the execution during run time. In such a case the single-owner protocol cannot

take advantage of the knowledge about the location of the page owner. Thus the multiple-owner protocol is favored.

3.6 Hybrid Owner Protocol

When a page has more than two readers and three writers, the single-owner protocol uses fewer messages to get current updates no matter where the page owner is. When there are fewer than three readers and writers, the multiple-owner protocol uses fewer messages than the single-owner protocol if the programmer cannot assign the page owner to a processor where one of the readers resides.

To combine the advantages of both protocols, we propose a *hybrid owner* protocol which can be used to increase the efficiency in an adaptive way, with some pages managed by the single-owner protocol and some by the lazy invalidate protocol, which may capture the dynamic behavior of the program.

When programs allocate memory from the shared memory heap by a system provided function call, `g_malloc`, in addition to the size of memory, they also need to specify which protocol those pages should use. For a page managed by the single-owner protocol the first thread that touches the page becomes the owner of the page. After performing a release operation, updates of the page in the single-owner are sent to the owner. When a thread wants to read the page, it sends a request to its owner, who then returns current data. For a page managed by the lazy invalidate protocol, the reader requests updates from the writers according to write notices it has collected.

Chapter 4

Implementation of the Single-owner Protocol

Multiple writer protocols allow inconsistent copies of a page to exist in distributed shared memory systems until a synchronization point. The write-shared data protocol updates all existing copies in the system at the synchronization point. The lazy invalidate protocol keeps all updates of a page locally but a reader of an invalid page needs to collect current updates from some writers to make the page current. As we mentioned in Chapter 3, these conventional multiple writer protocols need $O(n^2)$ messages for n readers to make an invalid page current if the n readers also are writers to the invalid page.

In order to reduce number of messages to make a page valid, the single-owner protocol was proposed. The single-owner protocol is also a multiple writer protocol. In contrast with conventional multiple writer protocols, each page in the shared memory has a designated page owner. When a writer executes a release operation, the writer creates diffs and sends them to their page owners. Instead of contacting some writers to get the current data, a reader sends a request to the page owner in order to

obtain the current data. The single-owner protocol takes $\Theta(n)$ messages to make all the copies of a written page valid if n threads write to the page and they all attempt to read the page. In this chapter, we present an implementation of the single-owner protocol. We also demonstrate how the single-owner protocol solves the problems of diff accumulation and synchronous garbage collection.

4.1 Diff Creation

Due to limited RAM in our test environment, we use a slightly modified software write detection method [ZSB94] to create the diffs. See Sec. 1.3.3. A diff consists of addresses and final values of a sequence of writes to a page on the shared memory. A piece of code is manually inserted before write operations to shared memory. This piece of code marks a bit which indicates the corresponding address written by the application. The smallest granularity of operations to the shared memory is a word, which is four bytes in our system.

We also partition a physical page, 4 K, equally into 32 pieces of 128 bytes, called *pseudo pages*. Our code also sets a bit of the *pseudo page bit map* if the pseudo page is written.

This definition is different from [CBZ95] in which a diff describes the difference between twin pages. In [CBZ95], there is no diff at all if a thread does not change the value on the twin page even if it writes to the page.

4.2 Managing Write Notices

The system architecture is the same as in Fig. 2.5. Assume there are n threads. As mentioned in Chapter 2, a modified lazy invalidate protocol is used to propagate invalidation information of written pages. Instead of

passing write notice sets from a thread to another thread in the lazy invalidate protocol, this modified protocol passes write notice sets from a thread to a synchronization object or from a synchronization object to a thread. In addition to the written page number and the time stamp when a write notice is created (see Sec. 2.5.3), a write notice includes a pseudo page bit map in the implementation of the single-owner protocol. A write notice indicates which pseudo pages have been modified in the page.

4.2.1 Write Notice Table

Each system server has $n+1$ time stamp vectors [KCDZ94, BM93] of length n, `SiteTV[0..n-1]` and `TableTV`, where n is the number of threads in the system. Each system server also has a write notice table with n entries. Each entry points to a list of write notice sets from the same thread in their time stamp order. The list always stores write notice sets from contiguous phases. (We may consider the execution of a thread as a sequence of phases. Two phases are contiguous if one phase immediately follows the other in the sequence with no intervening phases in between.) `TableTV[j]` indicates the latest write notice set on the jth entry in the write notice table.

`SiteTV[j]` is thread i's conservative estimate of what write notice sets thread j has. From thread i's view point, thread j keeps write notices which were generated by thread k with time stamps smaller than or equal to `SiteTV[j][k]`. `SiteTV` is used to propagate a sufficient set of write notices to other system servers. `SiteTV` is also used to perform garbage collection asynchronously.

4.2.2 Executing a Synchronization Operation

When thread i executes a *release* operation, it informs its system server of the attribute and the parameter of the operation. The system server finds the

location of the synchronization server, say on system server j. System server i collects write notice sets created at the phases which reach the current phase of thread i and which system server j does not have (based on `SiteTV[j]`). System server i sends system server j the write notice sets and the parameters of the synchronization operation, the synchronization object id, the operation id and the operation's attribute, etc. After system server i sends the write notices to system server j, system server i updates `SiteTV[j]` with the latest time stamps in the sent write notice sets. If thread i executes an operation with the *acquire* attribute, the system server does not include write notice sets in the requesting message.

4.2.3 Synchronization Object

When system server j receives the synchronization request with write notice sets from system server i, it puts the write notice sets into its write notice table if the time stamp of the write notice set is greater than the corresponding entry of `TableTV`; otherwise the system server may discard the write notice set. System server j also updates its `SiteTV[i]` based on write notices it receives from system server i. Afterwards, system server j calls the corresponding synchronization object to perform the operation.

Each synchronization object has a time stamp vector, `SyncTV`. `SyncTV[i]` stores the latest phase of thread i which reaches the object. When the synchronization object executes `SyncReply(...)`, the object informs the local system server about its `SyncTV`, the result of the execution and the attribute of the operation. If the request is for an *acquire* operation, the system server collects all write notice sets (created by thread k) whose time stamps are greater than `SiteTV[i][k]` and less than or equal to `SyncTV[k]` for all k. (The phases creating these write notices reach the current phase of the synchronization object but they are not in system server i.) Then, system server

j updates `SiteTV[i]` with the time stamps in the sending write notice sets.

4.3 Making a Page Up to Date

4.3.1 Page Table

System servers are responsible for invalidating the local copies of pages in shared memory. System servers also serve read requests from local threads and other system servers. Each system server also has a page table to monitor the status of the local caches of pages in shared memory. For each page α, the page table maintains two time vectors and a bit map, $\texttt{wPhase}_\alpha$, $\texttt{wnPhase}_\alpha$ and $\texttt{wBitMap}_\alpha$. See Fig. 4.1.

$\texttt{wnPhase}_\alpha\texttt{[j]}$ stores the last reachable phase of thread j in which thread j writes to page α. $\texttt{wPhase}_\alpha\texttt{[j]}$ stores a time stamp, the time when the last update from thread j has been applied to the local copy of α. *Active write notices* of a page are those created after $\texttt{wPhase}_\alpha\texttt{[k]}$ but not later than $\texttt{wnPhase}_\alpha\texttt{[k]}$ for $0 \leq k < n$. If the set of active write notices is not empty, the local copy of the page is invalid. The system server informs the local thread to protect the page from illegal accesses. $\texttt{wBitMap}_\alpha$ is the bitwise-OR of α's pseudo page bit maps from those active write notices. $\texttt{wBitMap}_\alpha$ indicates which pseudo pages were written by other threads since the page became invalid.

4.3.2 Page Owner

Initially, each page in shared memory is owned by some system server. The system server is the initial *page owner* of the page. This information is known to all system servers. Later the system server of the first thread reading or writing the page is designated to be the page owner of the page by the initial page owner.

```
class PageEntry{
public:
  invalidPage(int writer, int _time_stamp, int _bit_map);
  readRequest(int reader, vector &_wnPhase, int _bitmap);
  localReadRequest(int page);
  updatePage(int writer, vector &_wnPhase, int _time_stamp, Diff &_diff);
private:

  /* wnPhase[i] indicates the last reachable phase of thread i to the
  current phase of the local thread when thread i writes to the page.
  */

  vector wnPhase;

  /* wPhase[i] indicates the time stamp when the last update from
  thread i has been applied to the local copy of the page */

  vector wPhase;

  /* wBitMap indicates which pseudo pages have been written after
  wPhase and before wnPhase */

  int wBitMap;

  /* valid indicates whether the page is valid or not */
  boolean valid;


  /* These two queues are used to store read and write requests if
  they arrive too early */

  Queue readRequestQueue;
  Queue writeRequestQueue;
};
```

Figure 4.1: Page table

```
PageEntry::invalidPage(int _writer, int _time_stamp, int _bit_map){

  if ( wPhase[_writer] < _time_stamp) // never obtain the diff before
    wBitMap = wBitMap XOR _bit_map;

  if ( wnPhase[_writer] < _time_stamp) { // new write notice.
    wnPhase[_writer] =  _time_stamp;
    if (valid) {

      valid = FALSE;
      return 1;// inform system server a newly invalided page.
    }else return 0; // the page is invalid already.

  }
}
```

Figure 4.2: Invalidating a page

Initially each system server sends a read request to the initial page owner when its local thread attempts to read the page. The initial page owner either grants the ownership to the requesting system server—The system server then becomes the new owner—or informs the requesting system server of the new owner and then forwards the read request to the new owner. The ownership of a page can be transfered only once in our current implementation.

The ownerships of pages are also propagated with write notices to other system servers in order to reduce the resending of messages for read and write requests. Programs may intentionally touch shared pages at the beginning of execution to find out who owns them, to reduce unnecessary messages passing around afterwards.

4.3.3 Invalidating a Page

When system server i receives the response from a synchronization object for an operation with an *acquire* attribute, it gets several write notices. After it puts them into its write notice table, it collects an *invalidation set* for the

local thread. An invalidation set consists of all write notices created in the phases which reach thread i's new phase.

The system server sets $\texttt{wnPhase}_\alpha\texttt{[j]}$ to s if a write notice for page α in the invalidation set is generated by thread j with time stamp s and s is greater than the old $\texttt{wnPhase}_\alpha\texttt{[j]}$. The system server bitwise-ORs its $\texttt{wBitMap}_\alpha$ with the pseudo page bit map in the write notice if s is greater than local $\texttt{wPhase}_\alpha\texttt{[j]}$ and $i \neq j$. See Fig. 4.2.

A page is invalid if an active write notice for the page exists. The system server collects the pages, whose status become invalid, and sends their page numbers to the local thread. The local thread sets read protection on those pages. Any illegal accesses to those pages generates a segment fault. A memory handler takes an appropriate action to make the page valid.

4.3.4 Updating the Copy of a Page in the Page Owner

When a thread executes a *release* operation, the thread generates diffs of modified pages. The system server bundles all diffs whose page owners are in the same processor with their `wnPhase` time stamp vectors and sends them to the processor.

After the system server receives α's diff and $\texttt{wnPhase}'_\alpha$ from the sender, it checks whether α's diff arrives early by comparing its local $\texttt{wPhase}_\alpha$ against $\texttt{wnPhase}'$. If entry i in its local $\texttt{wPhase}_\alpha$ is greater than entry i in $\texttt{wnPhase}'$ for all $0 \leq i < n$, where n is number of threads, the system server applies the diffs to the local cache of page α. Otherwise, the system server suspends the work by putting the writer's request into a list, `writeRequestQueue`, until the above condition is satisfied. See Fig. 4.3. Thus, all the diffs for α in the page owner are applied in the order of the DAG defined at Sec. 2.3.2.

```
PageEntry::updatePage(int _writer, vector& _wnPhase, int _time_stamp,
                      Diff &_diff){
  if (_wnPhase <= wPhase) {// all reachable writes have arrived.
    apply _diff to the page;
    wPhase[i] = _time_stamp;

    // remove a pending writer, whose wnPhase is smaller than or equal to
    // current wPhase from writeRequestQueue.

    while ((head = writeRequestQueue.getPendingMember(wPhase)) != NULL) {
      apply head.diff to the page;
      wPhase[i] = head.time_stamp;
    }

    // remove a pending reader whose wnPhase is smaller than or equal to
    // current wPhase from readRequestQueue.

    while ((head = writeRequestQueue.getPendingMember(wPhase)) != NULL) {
      reply to the reader with all pseudo pages marked on head.bitmap;
    }

  }else writeRequestQueue.enqueue(_writer, _wnPhase, _time_stamp, _diff);
}
```

Figure 4.3: Updating the page owner

```
PageEntry::readRequest(int reader, vector &_wnPhase, int _bitmap){
  if (_wnPhase <= wPhase) { // all diffs have arrived
    reply to the reader with all pseudo pages marked on _bitmap;
  }else {// some diffs are not arrived
    readRequestList->enqueue(int reader, vector &_wnPhase, int _bitmap);
  }
}
```

Figure 4.4: The procedure page owners use to handle a read request

```
PageEntry::localReadRequest(int page){
  if (itself is the page owner)  readRequest(myid, wnPhase, wBitMap);
  else send a request to the page owner.
}
```

Figure 4.5: The procedure system servers use to handle a read request

4.3.5 Reading an Invalid Page

Read Request

When thread i attempts to read an invalid page α in shared memory, a memory handler catches the memory fault and sends a read request to its system server.

If system server i is not page α's page owner, it sends a read request with its copy of $\texttt{wnPhase}^i_\alpha$ and of $\texttt{wBitMap}^i_\alpha$ to the page owner. See Fig. 4.5.

After the page owner receives a read request, it checks whether all diffs for the read request have arrived by comparing its own local $\texttt{wPhase}_\alpha$ against $\texttt{wnPhase}^i_\alpha$. If $\texttt{wnPhase}^i_\alpha\texttt{[j]}$ is smaller than or equal to $\texttt{wPhase}_\alpha\texttt{[j]}$ for all j, all diffs of thread i's active write notices have arrived already. The page owner sends all pseudo pages marked on $\texttt{wBitMap}^i_\alpha$ to system server i. If the condition does not hold, the page owner puts the read request into a list `readRequestList` until all diffs for thread i's active write notices arrive. See Fig. 4.3 and Fig. 4.4.

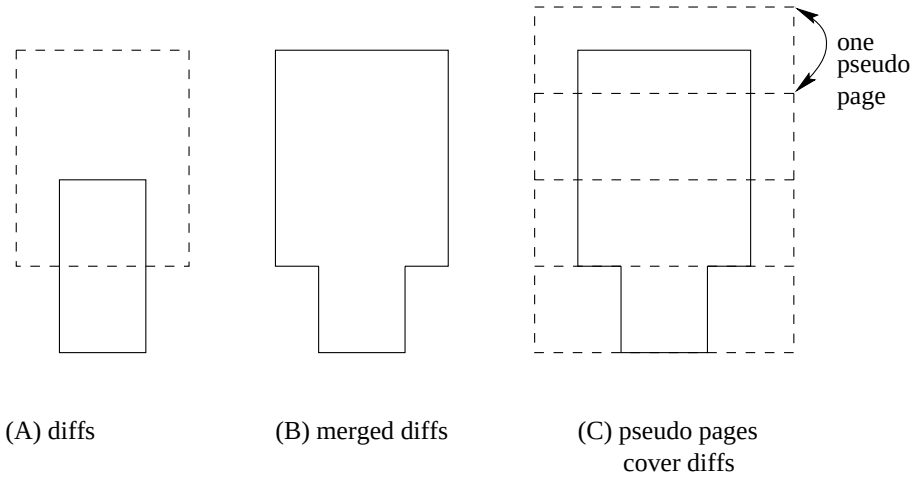

Figure 4.6: Diffs and pseudo pages

Pseudo Pages

Instead of sending a set of diffs, the single-owner protocol responds to a read request with a set of pseudo pages. The single-owner protocol merges all diffs into one entity, called a merged diff. See Fig. 4.6(B). Then it sends the set of pseudo pages (see Fig. 4.6(C)) which cover the merged diff to the requesting system server. The merged diff is obtained by applying the set of diffs to the page in the DAG order. The merged diff consists of all addresses and the final value written by the set of diffs. The set of pseudo pages is a superset of the merged diff.

An exceptional situation may occur when a thread writes to page α and the diff of that page has not been flushed to the page owner. Page α is invalidated after performing a synchronization operation with attribute *acquire*. The pseudo pages acquired from the page owner may clobber what the thread has written on α.

In order to deal with this problem, the thread creates a diff for page α

```
invalidAccess(int page){
  // This procedure is called by memory handler at thread process when
  // a invalid page is accessed.

  if ((the page has been written but the result has not been flushed to
      its page owner) && (the page owner is not local)) {

    obtain diff for what the local thread has written to the page;
  }

  send read request to system server;

  wait until the page is updated by the system server;

  apply the diff back to the page;

}
```

Figure 4.7: The procedure threads use to handle an invalid access to a page

before the system server applies the pseudo pages (from the page owner) to page α. After applying the pseudo pages onto page α, the thread redoes the diff, thus restoring the thread's writes. See Fig. 4.7.

There is no need to keep diffs in local sites because the page owner keeps the current state of α, and each thread knows which pseudo pages should be fetched. Since a superset of the merged diff is sent to the requesting thread, there is no diff accumulation at all. At the end of this section, we will demonstrate that the extra data in the pseudo pages does not affect the memory consistency.

4.3.6 Writing to a Page

The system does not protect a page from write accesses since the thread does not read stale information from the page. Later if the thread attempts to read the written page, a segment fault is raised. The memory handler

creates diff containing whatever the thread has written to the page so far and then sends a read request to its system server. After the system applies the pseudo pages on its local copy, the thread redoes the diff back to the page. Then the computation resumes.

Since each thread reads valid pages and all written values depend on all valid values, delaying the fetching of current data for an invalid page does not affect its correctness. This procedure is essentially equivalent to protecting the page from both write and read access. When a segment fault occurs, the system server acquires a current copy of the page. Then the computation resumes. In some applications threads just write to a page without reading it. This approach saves several messages to make the page current.

4.3.7 Correctness of Pseudo Pages

In Sec. 4.3.5, we stated that the response of a page owner to a reader thread consists of a set of pseudo pages. The set of pseudo pages is a superset of the merged diff. In Sec. 4.3.4, we also showed that all diffs applied to the page of the page owner are in a DAG order. In this section, we will demonstrate that extra data in pseudo pages does not clobber the memory.

Assume that the communication between two processors is in FIFO order. Some pages in the local memory of a processor are reserved for shared memory. Those pages are initialized to zero. If thread i intends to read an invalid page, page α, system server i receives a set of pseudo pages to update its local copy of page α. We now demonstrate that the set of pseudo pages does not clobber the memory based on the algorithm we presented at Sec. 4.3.5.

Let X be a word in page α. X's corresponding value in the pseudo pages (from the page owner) is x_0, but x_0 is not in the merged diff. X's corresponding value in thread i's cache is x_1. P_0 is the phase when x_0 is created. P_1 is the phase when x_1 is created.

In this is thread i's first attempt to read the page, either P_1 is reachable from P_0, $P_0 \prec P_1$, or x_0 is equal to x_1, $P_0 = P_1$, since x_1 is the initial value. X in thread i cannot get clobbered in this case.

Assume this is the first time X in thread i gets clobbered by x_0. That means P_1 is reachable from P_0, $P_0 \prec P_1$.

Lemma 1 *x_1 is written by thread i but not by any other thread if X in thread i gets clobbered by x_0.*

Proof By contradiction, suppose x_1 is written by another thread j. Thread i can only get x_1 from the page owner based on the single-owner protocol. So x_1 is in the page owner before x_0. That is, either P_1 is concurrent with P_0 or P_0 is reachable from P_1, $P_1 \prec P_0$ since all writes are applied to the page owner in the DAG order. X cannot get clobbered in this case. This contradicts our assumption that x_1 is written by thread j and $j \neq i$. 2

There are two scenarios that may take place before thread i attempts to read page α. In the first scenario, thread i has executed a synchronization operation with either a *release*, *release_acquire* or *acquire_release* attribute such that x_1 has been flushed to the page owner before thread i reads page α. So x_1 is in the page owner before x_0. P_1 is either concurrent with P_0 or P_0 is reachable from P_1, $P_1 \prec P_0$. So X is not clobbered.

In the second scenario, thread i only executes synchronization operations with the *acquire* attribute after it writes x_1. By our algorithm, system server i keeps a copy of x_1 before x_0 is written to address X. Afterwards, the system server writes x_1 back to X again. Only when P_0 is reachable from P_1, X can be clobbered. However, there is no *release* operation executed since thread i writes x_1 to X. It is impossible for any phase to be reachable from P_1. It contradicts our assumption that X cannot get clobbered in this scenario. Therefore pseudo pages do not affect memory consistency.

4.4 Garbage Collection

After a thread invalidates pages in shared memory according to its invalidation set, all information is put in the `wnPhase`s and `wBitMap`s fields of the page table. The invalidation set is useless for the thread itself. But its system server keeps the set until it makes sure all other system servers get those write notices. Ideally, system server i may remove write notices from its write notice table as long as the information about those write notices has been put into the page table and their time stamps, generated by thread j, are smaller than `SiteTV[k][j]` for all $k \neq i$. `SiteTV` conservatively estimates what write notices other system servers have.

`SiteTV[j]` is updated when the local system server sends write notices to, or receives write notices from, system server j. Synchronization server j might now have synchronization objects. `SiteTV[j]` does not change during the computation in other synchronization servers. So each system server periodically broadcasts its `TableTV` to all other threads. All system servers update their `SiteTV`s based on the `TableTV`s they receive.

Once the size of the write notice table exceeds a certain threshold, system servers may remove useless write notices asynchronously.

Chapter 5

Performance Evaluation

5.1 Experimental Environment

The platform for the performance evaluation of our current implementation consists of eight Pentium-Pro personal computers with 64 M bytes RAM connected by 100Mbps Ethernet; the operating system is Linux; the communication protocol is TCP/IP; and the reported times are wall clock times.

We pick five applications, EP (embarrassingly parallel benchmark), IS (integer sort) from NAS [BBB+94], a heat flow transformation problem from the homework of the distributed computing class at NYU, Barnes-Hut from SPLASH-2 parallel application suite [WOT+95], and the Mandelbrot set from [GLS94].

We are going to compare the performance of the single-owner protocol to one of the multiple-owner protocols, the lazy invalidate protocol. Both of these protocols implement RC_{HS}. The applications use barriers and locks as their synchronization primitives, with the exception of the Mandelbrot set, where we implement a user-definable synchronization object.

In order to get a fair evaluation, both protocols share the same code for generating diffs and propagating write notices. However, the size of

a write notice in the single-owner protocol is larger than the one in the multiple-owner protocol since a write notice in the single-owner protocol includes its 4 bytes pseudo page bit map. The techniques for updating a stale page are also different between the single-owner protocol and the multiple-owner protocol.

When a thread intends to access an invalid copy of a page in the multiple-owner protocol, the system server sends read requests to a subset of system servers which create the active write notices (see Sec. 1.3.2). Garbage collection is not implemented in our version of the lazy invalidate protocol. The single-owner protocol implements garbage collection. The architecture has been illustrated in Fig. 2.5.

application	problem size	synchronizatoin type	sequential execution time (sec)
EP	$N = 2^{28}$	barrier	2306.4
HFP & HFP $_D$	2048x1024 matrix 30 loops	barrier	44.2
Barnes-Hutt	8192 bodies	barrier	30.7
IS_{128}	max of keys = 128 number of keys = 2^{23}	barrier, semaphore	10.7
IS_{1024}	max of keys = 1024 number of keys = 2^{23}	barrier, semaphore	10.7
Mandelbrot	x = [-2, -1.25] y = [0.5, 1.25]	barrier, semaphore or high level sync. obj.	18.2

Table 5.1: Application Profiles

5.2 Applications

5.2.1 An Embarrassingly Parallel Benchmark (EP)

Brief Statement of the Problem

The benchmark program [BBB+94] generates pairs of Gaussian random deviates and tabulates the number of pairs in successive square annuli. First the algorithm selects pairs of random numbers x, y, which satisfy $t = (2x - 1)^2 + (2y - 1)^2 \neq 0$. Random numbers are generated based on the pseudo code in [BBB+94]. Then it calculates independent Gaussian deviates, X and Y, where $X = x((-2\ln t)/t)^{(1/2)}$, $Y = y((-2\ln t)/t)^{(1/2)}$. It also tabulates Q_l as the count of the pairs (X, Y) that lie in the square annulus $l \leq \max(|X|, |Y|) < l + 1$, where $0 \leq l \leq 9$ and l is an integer. After generating 2^{28} random numbers, it prints the ten Q_l counts and the two sums $\sum X$ and $\sum Y$.

Implementation

In the parallel version, two tables, $Q'_l[0..n-1][0..9]$ and $S[0..n-1][0..1]$, where n is number of threads, are in the shared memory. Each thread generates $2^{28}/n$ random numbers; 2^{28} divides n without lose of generality. After obtaining the Gaussian deviates, the thread adds the result to its local counts for Q_l and local summations for X and Y. At the end of the computation, thread t writes its local result of Q_l into $Q'_l[t][0..9]$, its local summation of X into $S[t][0]$, and its local summation of Y into $S[t][1]$, where $0 \leq t < n$. After all the threads finish, thread 0 sums up values in $Q'_l[0..n-1][0..9]$ and $S[0..n-1][0..1]$ to get Q_l, $\sum X$, and $\sum Y$. Then the results are printed. The page owner of these two tables is in system server 0 in the single-owner protocol.

Performance

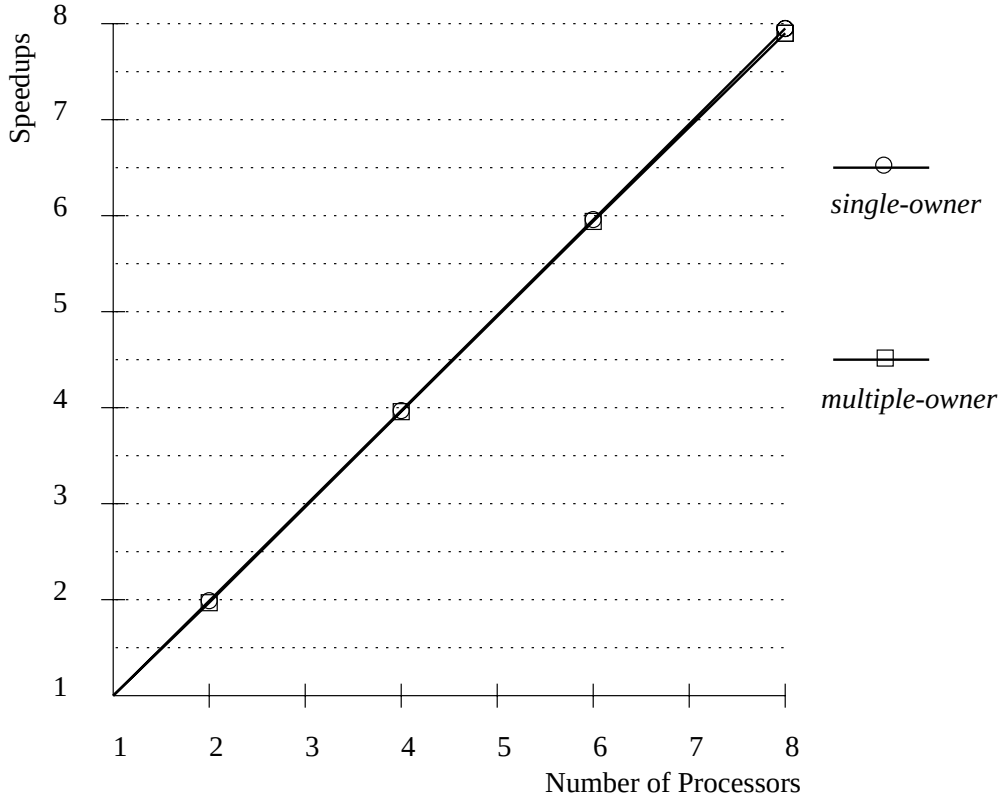

Figure 5.1: Speedup of EP

The sequential execution time of EP is $2,306.4$ seconds. Threads only communicate with thread 0 at the end of computation. The multiple-owner protocol on eight processors takes 291.2 seconds to finish the task. Its speedup is 7.9. The single-owner protocol takes 290.1 seconds. Its speedup is 7.9. Table 5.2 shows the number of messages sent and the total size of messages for the computation. Since the page owner of Q'_l and S is system server 0 and thread 0 is the only thread that reads those two arrays, the number of messages passed and total size of those messages in

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	5	10	15	20	25	30	35
	total size of all messages (K bytes)	0.2	0.6	1.1	1.5	2.1	2.7	3.5
multiple owner protocol	number of messages	6	12	18	24	30	36	42
	total size of all messages (K bytes)	0.2	0.6	1.1	1.7	2.6	3.6	4.9

Table 5.2: Message counts and message size for EP

the single-owner protocol is smaller than in the multiple-owner protocol. The memory access patterns are illustrated in Fig. 3.3(A) and 3.3(C)

Since EP is a coarse-grained computation, the performance of both protocols is close to the best result we can get. The result shows the single-owner protocol does not require too much overhead if threads seldom access the shared memory.

5.2.2 Heat-flow Transferring Problem (HFP)

Brief Statement of the Problem

The computation simulates the transfer of heat energy over a two dimensional square boundary. It is a simple classic PDE problem. We have discussed in Sec. 3.2 the potential difficulties if the program uses the multiple-owner protocol. Suppose there is a square surface whose sides are at a certain temperature (which remains constant over time). The interior of the square starts out with a constant zero temperature. We are interested in whether the temperature inside the square can reach a steady state—i.e.

the change in the temperature of the square is below a threshold over time. Let $T(x, y, t)$ be the temperature inside the square, where (x, y) specify the coordinates inside the square, and t is time. The partial differential equation specifying the heat transfer is $\frac{\partial^2 T}{\partial^2 x} + \frac{\partial^2 T}{\partial^2 y} = \frac{\partial T}{\partial t}$.

To approximate the solution is to divide the plate into a grid and simulate the equation on the grid over time. For a certain grid point (i, j) at time t, we can approximate the new temperature at time $t + \Delta t$ simply by $T(i, j, t + \Delta t) = T(i, j, t) + \frac{\Delta t}{(\Delta d)^2}(T(i-1, j, t) + T(i+1, j, t) + T(i, j-1, t) + T(i, j+1, t) - 4T(i, j, t))$, where Δd is the distance between two neighboring points on horizontal or vertical lines of the grid, and Δt is the incremental time step for the simulation.

Implementation

The program runs in time steps. In the sequential code, we use two 2-dimensional floating-point arrays A_0 and A_1 to store the $N_1 \times N_2$ grid at the even steps and the odd steps respectively. Each entry, $A_x[i][j]$, at step k stores the temperature of a grid $T(i, j, k\Delta t)$, where $x = 0$ if k is an even number and $x = 1$ if k is an odd number. At an even step, the program reads A_1 from the previous step and writes the current temperatures to A_0. At an odd step, the program reads A_0 from the previous step and writes the current temperatures to A_1. Before proceeding to next step, the program checks if $| A_0[i][j] - A_1[i][j] | < \epsilon$, for all i, j and for predetermined ϵ. If the condition is true, the system has reached steady state and terminates. Otherwise, the computation continues.

In the parallel version, thread p computes a submatrix of A_x, from row $(N_1 p)/n$ to row $(N_1(p+1)/n) - 1$, where n is number of threads and N_1 is divisible by n (without loss of the generality). The submatrices are stored in local memory. At each step, the thread checks whether its submatrix is in

steady state. If it is, thread p writes 1 to $D[p]$, otherwise, 0. After a barrier, each thread reads the array D. If $D[i]$ is equal to 1 for all $0 \leq i < n$, all the threads terminate, otherwise they proceed to the next step. There is a barrier between two time steps.

When a thread computes the rows on the boundary of its submatrix, it needs to read rows from other thread's submatrix (Jacobian computation [GLS94]). We use array $B[0..2n-3][0..N_2-1]$ in shared memory to exchange the information. When thread t finishes computing its submatrix, it puts the first row of its submatrix into $B[2p-1]$ and the last row of its submatrix into $B[2p]$, where $p \neq 0$ and $p \neq n-1$. When thread p computes its submatrix, it reads other threads' rows from $B[2(p-1)]$ and $B[2(p+1)-1]$. Thread 0 writes to $B[0]$ and reads from $B[1]$. Thread n writes to $B[2n-3]$ and reads from $B[2n-4]$.

We have two sets of programs. One, HFP_D, writes 1 or 0 into D and check whether the system reaches the steady state. Another, HFP, does not write 1 or 0 to D. We are interested in HFP because its access pattern is very similar to another classic application SOR, successive over-relaxation [Car93].

In the single-owner protocol, the page owner of $B[i]$ is system server $i/2+1$ if i is even otherwise it is system server $(i-1)/2$, where $i \neq 1$ and $i \neq 2n-1$. The reader of $B[i]$ owns the pages of $B[i]$. The system server of thread 0 owns $B[1]$. System server $n-1$ owns $B[2n-4]$.

Performance

The problem size is 2048×1024 with at most 30 loops, if the program cannot reach the stable state. See Table 5.1.

The execution time for the sequential code is 44.2 seconds. The execution time of HFP using the single-owner protocol on eight processors is 6.2 seconds. Its speedup is 7.2. The execution time using the

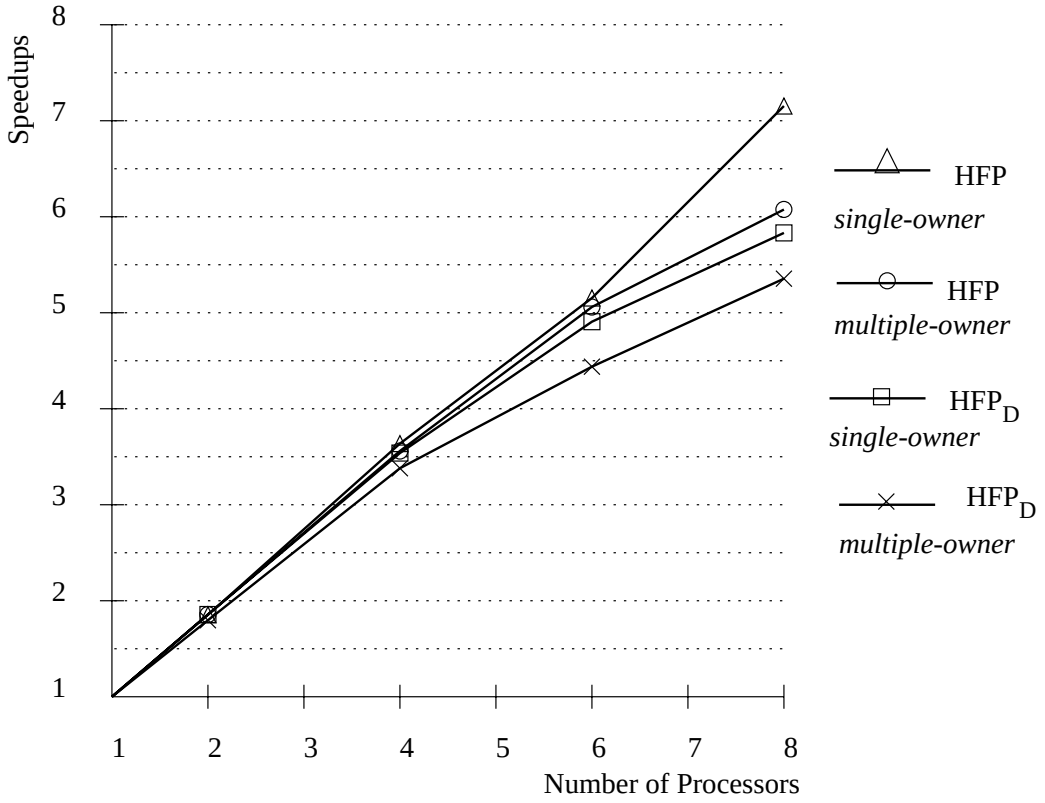

Figure 5.2: Speedup of heat flow problem

multiple-owner protocol is 7.3 seconds. Its speedup is 6.1. Table 5.4 shows the multiple-owner protocol uses more messages but transmits a similar amount of data in the course of computation. The single-owner protocol outperforms the multiple-owner protocol when number of threads is more than three, see Fig. 5.2.

The pattern of accessing matrix B in both protocols can be depicted by Fig. 3.7(A) and (C). The single-owner protocol takes advantage of knowledge about the locations of readers. Writers sends diffs to page owners after they reach barrier. So thread t does not need to send a read request to get B's

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	182	394	606	818	1030	1242	1454
	total size of all messages (M bytes)	0.5	0.9	1.4	1.9	2.5	3	3.5
multiple owner protocol	number of messages	412	940	1564	2262	2999	3746	4916
	total size of all messages (M bytes)	0.5	0.9	1.4	1.9	2.5	3	3.5

Table 5.3: Message counts and message size for HFP_D

rows from threads, $t-1$ and $t+1$ since threads, $t-1$ and $t+1$ eventually send diffs to thread t.

HFP_D using the single-owner protocol takes 7.6 seconds. Its speed up is 5.8. HFP_D using the multiple-owner protocol takes 8.3 seconds on eight processors. Its speedup is 5.4. See Fig. 5.2. The speedup of the multiple-owner protocol does not increase as quickly as the single-owner protocol when there are more than three threads. Even though the amount of data sent by the multiple-owner protocol is very close to the amount sent by the single-owner protocol, the multiple-owner protocol sends many more messages than the single-owner protocol (see Table 5.3). The access patterns of writing a value to D and reading D array can be depicted by Fig. 3.2(A) and (B).

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	121	244	366	488	610	732	854
	total size of all messages (M bytes)	0.5	1.0	1.4	1.9	2.4	2.9	3.4
multiple owner protocol	number of messages	294	588	882	1176	1470	1764	2058
	total size of all messages (M bytes)	0.5	0.9	1.4	1.9	2.4	2.9	3.4

Table 5.4: Message counts and message size for HFP

5.2.3 Barnes-Hut

Brief Statement of the Problem

The Barnes-Huts method [BH86] is one of the well-known hierarchical methods to solve the classical N-body problem. The classical N-body problem models a physical domain as a system of n discrete particles and studies the changes of the system under the influences exerted on each particle by the whole set.

Barnes-Hut method consists of four phases [SHT+95], [WOT+95]. The first phase builds a tree based on the location of each particle. Each leaf in the tree stores properties of a particle such as its mass, position, acceleration, potential, velocity, etc. Each internal node of the tree maintains the information for the center of mass of all particles in its subtree. In the second phase, it computes the center of mass for each tree node. In the third phase the tree is used to compute the forces acting on all the particles. Finally the force acting on a particle is used to update particles' properties.

Implementation

The sequential execution is divided into time steps. At each time step, a tree is created based on the locations of all particles. The tree is traversed bottom up to find the center of mass for each tree node. Then the potential and acceleration of a particle is computed by calculating the influence of all other particles. In this phase, the tree is traversed top down for each particle. If the distance between the tree node and the particle is out of certain range, the influence from all particles under the subtree to the particle can be just calculated according to the property in the tree node. Otherwise, the children of the tree node are visited. In the last phase the new position and the velocity of each particle is calculated.

In the parallel version, we uses *costzones techniques* [SHT+95] to partition particles into subsets. The partitioned subsets are determined at the second time step. The partition of the subsets depends on the location of particles in space, but it is irrelevant to locations in the array which stores their coordinates.

Each thread is in charge of the computation for a subset. Threads work on the same set of particles during the computation. The vectors which maintain locations of particles are stored in the shared memory. All threads know the masses of all the particles. In order to reduce the number of synchronizations at each time step, two arrays are used to keep vectors. In each time step, threads read the array written in the previous step and then write the result to the other array.

In the first phase, a private tree is built [LDCZ95] in each thread. The array keeping the current locations of all particles is read by all threads. Then the threads compute the centers of mass for all internal tree nodes. In the third phase, the threads compute forces acting on the particles in its subset.

After the new position of each particle is computed, the result is stored into the array.

Performance

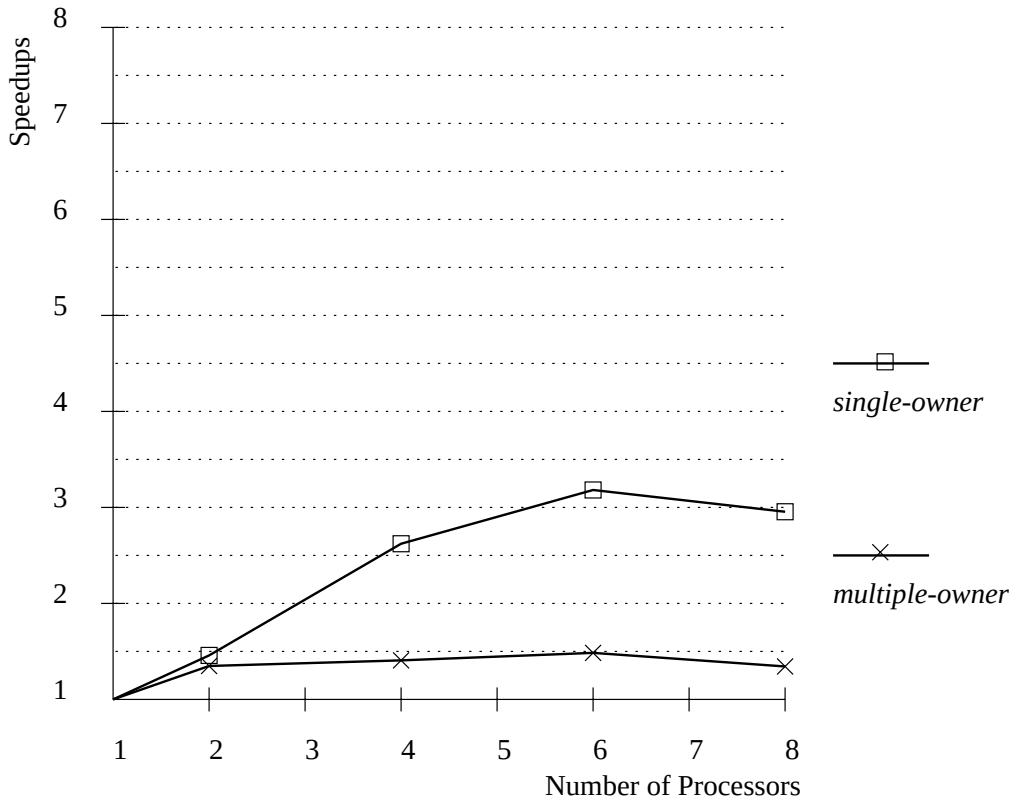

Figure 5.3: Speedup of Barnes-Hut problem

We ran Barnes-Hut with 8192 particles for 7 time steps. The last five iterations are timed to avoid cold start effects. The execution time takes 30.7 seconds. See Table 5.1.

Since the coordinates of the particles a thread computes are irrelevant to the locations where the particles are stored in the shared memory, threads

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	814	1638	2472	3316	4165	5026	5900
	total size of all messages (M bytes)	1.45	3.8	5.5	7.2	8.8	10.4	12.1
multiple owner protocol	number of messages	1598	4309	8148	12392	16406	21624	28208
	total size of all messages (M bytes)	1.6	3.4	5.3	7.2	9.2	11.3	13.5

Table 5.5: Message count and message size of Barnes-Hut problem

may write to the same page simultaneously. In the first phase, all threads read the page. The access patterns are illustrated in Fig. 3.2(A)(B). As we demonstrated in Chapter 3, the single-owner protocol and the multiple-owner protocol transmit almost the same amount of data across the network. But the multiple-owner protocol sends three times more messages than the single-owner protocol when eight threads run the program. The speedup of the single-owner protocol with eight threads is 3.0. See in Fig. 5.3. The speedup of the multiple-owner protocol with eight threads is 1.3.

5.2.4 Integer Sort

Brief Statement of the Problem

The benchmark program integer sort (IS) is selected from NAS [BBB+94]. Integer Sort assumes each of the N inputs keys is an integer in the range from 0 to $B_{\max} - 1$ for some integer $B_{\max}$. Integer sort is based on determining the number of keys smaller than i, for all i, $0 \leq i < B_{\max}$. The information

can be used to place a key directly in its position in the output array. The parameter values suggested in NAS are $N = 2^{23}$ and $B_{\max} = 2^{19}$. Due to limited RAM in our environment, the rank of each key is not stored in the memory, but computed on the fly.

Since the ratio of computation and communication is low, we need to reduce $B_{\max}$ to 1024 and 128 to get better performance. We will refer to the test set for $B_{\max} = 1024$ as IS_{1024} and for $B_{\max} = 128$ as IS_{128}. We are specifically interested in the performance of this application, which exposes the weaknesses of the single-owner protocol and the multiple-owner protocol.

Implementation

The implementation of IS in the parallel version partitions input keys into n subgroups evenly, where n is number of threads. Each thread stores one of the subgroups into its local array. Each thread counts the number of keys equal to i in its subgroup. Then each thread adds its local counts to a global array, G, exclusively by requesting a lock. Before a thread adds its local result to the array, it keeps a local copy of the global array, which is the subtotals of local counts from those threads that accessed the global array previously. After all the threads reach a barrier, each thread reads the final result of the global array. By using the subtotals and final result of the global array all threads rank keys in their subgroup.

Performance

The sequential execution time is 10.7 second. See Table 5.1. The patterns of accessing the shared memory are depicted by Fig. 3.5(A) and (B). The writers in the single-owner protocol need to send an extra message to update the page owners. In contrast, in the multiple-owner protocol thread

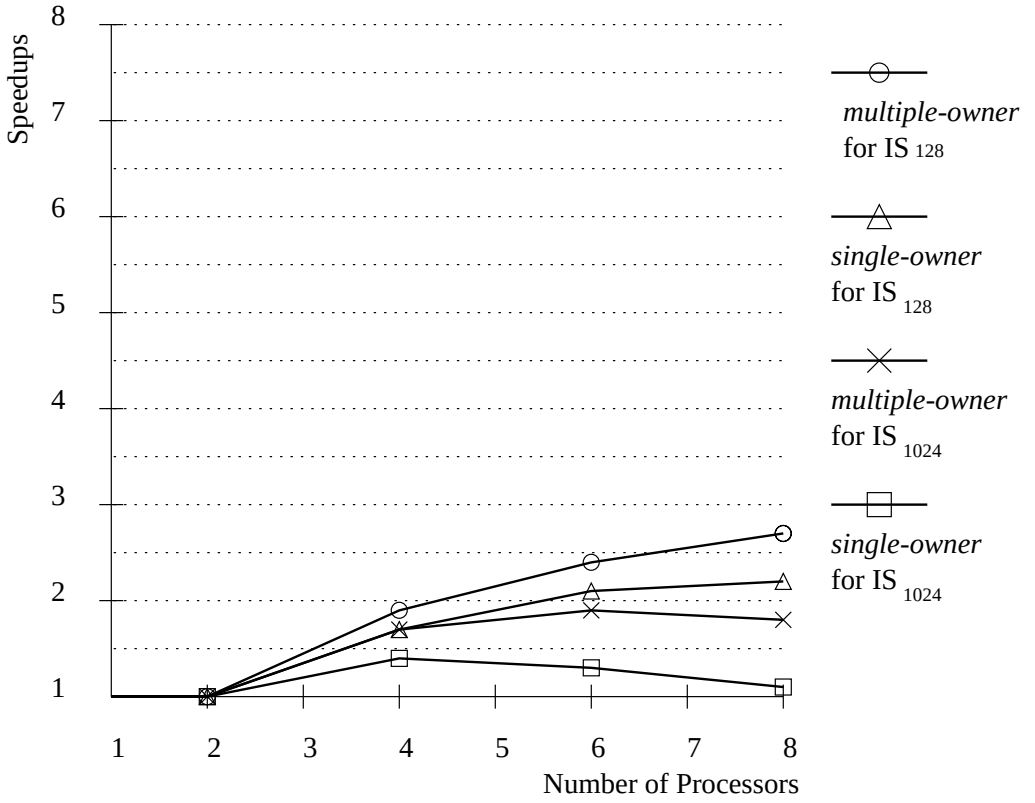

Figure 5.4: Speedup of integer sort

t gets the current G from the last writer, t_l directly. However, thread t always sends the diffs which thread t_l sends and the new diff it created to the next reader of G (diff accumulation). Table 5.7 and 5.6 show that the data sent in the multiple-owner protocol is much larger than the data sent in the single-owner protocol. The single-owner protocol sends slightly more messages than the multiple-owner protocol does. However, the speedup chart in Fig. 5.4 shows IS_{128} and IS_{1024} using the multiple-owner protocol outperform the ones using the single-owner protocol. So sending extra messages to page owners slows down the system much more than sending messages in large size

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	72	164	266	368	470	572	674
	total size of all messages (K bytes)	82.3	206.6	334.4	462.4	591.7	722.9	854.8
multiple owner protocol	number of messages	82	164	256	348	440	532	624
	total size of all messages (K bytes)	82.5	247.5	497.5	829.9	1246.1	1746.9	2332.7

Table 5.6: Message counts and message size for IS_{1024}

caused by diff accumulation.

We can exploit some advantages of the single-owner protocol so that the single-owner protocol outperforms the multiple-owner protocol. For example, thread i may keep an array G_i in the shared memory where system server i is the owner of G_i, for each $0 \leq i < n$. After each thread gets its subtotal, it goes to sleep except for thread 0. Thread 0 writes its subtotal to G_1 and wakes up thread 1. Then thread 0 waits on a barrier. After thread i writes its subtotal to G_{i+1} and wakes up thread $i+1$, thread i waits on barrier for all j, where $0 < j < n-1$. After thread $n-1$ writes its result to G_0 and reach the barrier, all threads wake up and read G_0. The number of messages sent is very close to the number sent in a message passing protocol. However, when the size of G is big as NAS requires, this method requires too much memory.

number of processors		2	3	4	5	6	7	8
single owner protocol	number of messages	72	164	266	368	470	572	674
	total size of all messages (K bytes)	12.3	31.7	54.1	77.1	101.7	127.7	155.6
multiple owner protocol	number of messages	82	164	256	348	440	532	624
	total size of all messages (K bytes)	82.5	37.6	77.5	129.9	196.1	276.7	371.9

Table 5.7: Message counts and message size for IS_{128}

5.2.5 Mandelbrot

Brief Statement of the Problem

The Mandelbrot set is defined as the set of all complex numbers c that satisfy conditions described below.

Define the function $f_c^n(z)$ as the repeated application of $f_c(z) = z^2 + c$, where c and z are numbers on the complex plane. Thus $f_c^1(z) = f_c(z)$ and $f_c^2(z) = f_c(f_c(z))$. If $f_c^n(0)$ stays bounded as $n \to \infty$, then we say that c is a member of the Mandelbrot set.

It can be shown that if $|f_c^n(0)| > 2$ for some n, then f_c^n does not stay bounded at c. For computational purposes, we pick a suitably large number N and calculate $f_c^n(0)$ for $n < N$. If $|f_c^n| \leq 2$ for N steps, then we add it to the set. If not, then we do not add it to the set.

We can characterize each complex number c by a pixel of an image. If f_c stays bounded for N steps we can give it some color, say black. If f_c becomes unbounded after n steps, we can give it another color that is some function of n. Thus, a sequential version of this program can iterate through all points

on the screen, assigning a color to each pixel. In this way, a striking picture develops.

An optimization to the sequential algorithm is described in [GLS94]. The authors note that if the border of any square of pixels are all the same color, then the interior must be of the same color also. Thus our new algorithm starts by checking the boundary of the region, and if it is all the same color, it colors the interior. If not, it breaks the region into equal sized blocks and recursively calls itself on each new region. If a particular block is of a minimum size, it stops breaking the block and colors the interior pixel by pixel. This approach substantially speeds up the computation.

Implementation

The simple parallelization of this algorithm involves putting the blocks in a shared task pool. Each thread accesses the pool, computes the block and does one of two things: it either breaks up the block and puts the new pieces in the shared task pool or it colors the whole block (either because it is of minimum size or because it is all one color).

Because the computation time of each block is small compared to the cost of communication on a typical network of workstations, we adopt a modification of the original scheme to make it slightly coarser grained.

Each thread will have its own local task pool from which it grabs the blocks. Periodically, say after doing N blocks, the thread will consult with the global pool to see whether it should take or add blocks to the global pool to more properly balance the workload. The decision is based on an approximation to the total number of existing blocks. It takes this number and divides it by the number of threads. This result can be assumed to be the number of blocks each thread should be working on (in the ideal case). If the current thread has more than that number of blocks, it takes some from

its local pool and adds it to the global pool. If it has fewer, it takes from the global and adds to the local.

We use two different methods to implement the algorithm. In the first one, we use semaphores and barriers to coordinate threads. The global pool and control information about how many blocks are in the global pool, the approximate number of blocks in the system and how many threads are sleeping, is protected by a binary semaphore, `mutex`. When a thread finds out that the pool is empty and itself does not have any block to work on, it leaves the critical section and sleeps on a semaphore `empty`. If a thread finds out there are more blocks in the pool and some threads asleep on `empty`, the thread wakes up those sleeping threads after it leaves the critical section. In the single-owner protocol, the page owner of the control information and the global pool resides in the same processor. The control information and the global pool are in separate pages.

The second method uses a high level synchronization primitive to manage the control information and synchronize threads. The information of a block is still stored in the global pool in shared memory. A high level synchronization class, `Pool`, is defined. `GetAction(...)`, `Init(...)` and `Done()` are three interface functions to threads. `GetAction()` is annotated with attribute *acquire*. `Done()` is annotated with attribute *release*. `Init(...)` is not annotated with any attribute.

`GetAction(...)` gives the number of blocks in the local memory of the requesting thread to the synchronization object. The synchronization object responds to the requesting thread with the number of blocks that the thread should put into the global pool or get from the global pool. After the thread finishes putting the blocks into the global pool or getting blocks from the global pool, it executes `Done()` to inform the synchronization object that the work is done. However, the synchronization object can ask the thread not

to move any blocks. The thread can keep working on those blocks that it already has.

If there are no blocks in the pool, the synchronization object just postpones the response until some threads put blocks into the pool or until every thread is waiting for a block. In the latter case, the synchronization object sends termination information to all threads. In our current implementation, only one thread can access the global pool at a time. Allowing multiple threads to get blocks from the pool (read the pool) simultaneously is also feasible for this application. But we are more interested in how the small change in the program with the high level synchronization object would affect the performance.

Performance

number of processors		2	3	4	5	6	7	8
single owner protocol with high level sync.	number of messages	103	133	204	257	291	311	375
	total size of all messages (K bytes)	5.2	8.3	14.2	19.2	23.1	27.7	35.7
single owner protocol with basic sync	number of messages	188	254	423	476	593	614	780
	total size of all messages (K bytes)	11.3	19.6	37.6	65.8	90.7	97.7	139.2
multiple owner with basic sync.	number of messages	215	313	412	554	648	719	760
	total size of all messages (K bytes)	10.1	20.8	40.2	67.4	98.5	138.1	163.5

Table 5.8: Message counts and message size for Mandelbrot set

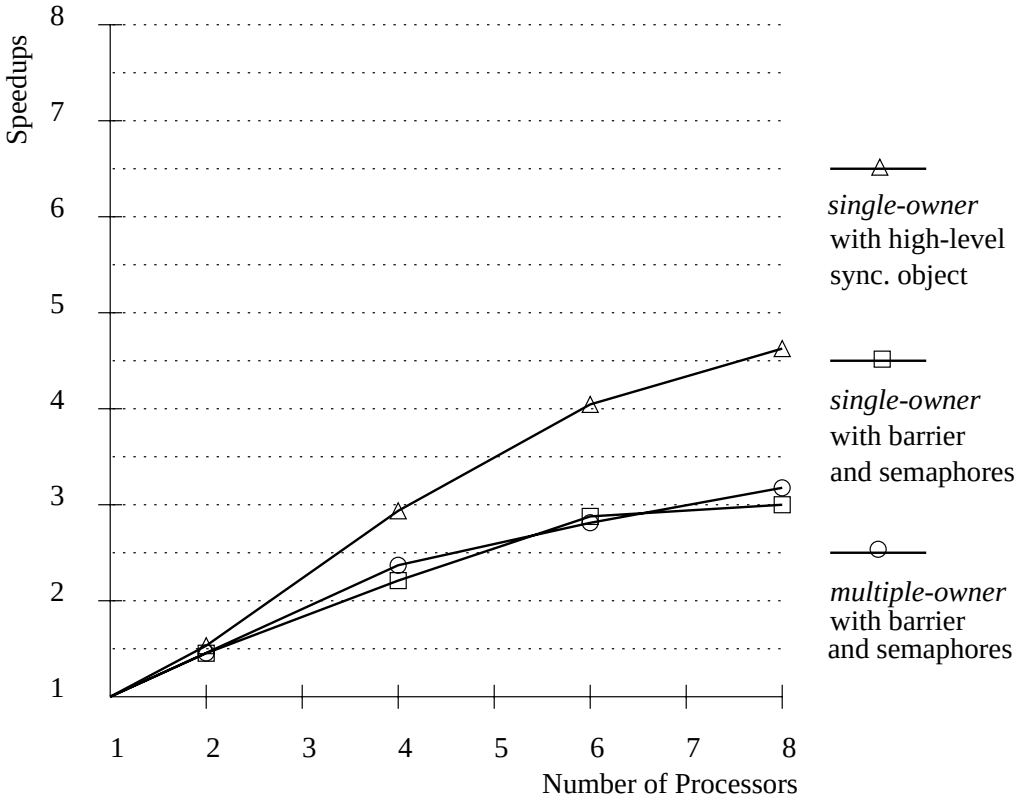

Figure 5.5: Speedup for computing Mandelbrot set

We computed the Mandelbrot set in the plane where $x = [-2, -1.25]$ and $y = [0.5, 1.25]$. Assume there are 720×480 points in the rectangle. Each point in the plane is tested by the iteration function, $f_c^n(z)$ with $z = 0$. If the function stays bounded after 256 iterations, we consider c to be in the Mandelbrot set.

Sequential execution time is 18.2 seconds. The execution time of single-owner protocol, with high level synchronization primitives, on eight processors is 3.95, see Fig. 5.5. Its speedup is 4.6. The execution time of single-owner protocol without high level synchronization primitives is

6.09 seconds on eight processors. Its speedup is 3.0. The execution time of multiple-owner protocol without high level synchronization primitives is 5.75 seconds on eight processors. Its speedup is 3.2.

Table 5.8 shows that the program with the high level synchronization primitive uses fewer messages and the message size is also smaller than the program with basic synchronization primitives. The high level synchronization primitive itself eliminates the need for accessing the page in which control information is located. This information is stored locally in the synchronization server. Putting a thread to sleep is done by postponing the response to the thread.

High-level synchronization primitives also manipulate the waiting queue easily. The server may work on a request from a thread with higher priority. In contrast, with semaphores, the thread always needs to fetch one more page for the control information when it enters the critical section. The thread goes to sleep or wakes up other sleeping threads by calling another semaphore, requiring more messages.

We are also interested in comparing the performance of the single-owner protocol without high-level synchronization primitives to the multiple-owner protocol. The message size and the access pattern for fetching the control information and the global pool are like IS_{128}, Sec. 5.2.4. However, threads may write to the global pool without reading it. In the single-owner protocol, page faults are not raised until threads attempt to read a page. In this case, system servers do not need to send a read request to the page owner of the global pool but send diffs to the page owner. In contrast, the multiple-owner protocol sends read requests to the last writer even though threads just write to the global pool.

Fig. 5.5 shows the performance of the program with the single-owner protocol without high level synchronization primitive is slightly

better than the one with the multiple-owner protocol. Even though the single-owner protocol always needs to send diffs to update page owners, a stale page does not need to get updated if threads just write to the page without reading it. However, when the number of threads increases, sending extra messages to page owners can slow down the system (see Sec. 5.2.4). The speedup of the single-owner protocol starts to fall behind when there are eight processors in the system.

Chapter 6

Related Work

6.1 PVM

PVM (Parallel Virtual Machine) [BDG+91, GBJ+94] is a run time system built on a heterogeneous collection of originally UNIX computers connected by a network. PVM provides a uniform interface as a single distributed memory parallel machine independent of the physical machines. In PVM, process-to-process communication is done with message passing.

The PVM system consists of two parts, the library of PVM interface routines and daemons. The library provides functions for users to pass messages and spawn processes. Daemons reside on all machines and run in the background. Daemons are used to establish first communication between two processes. The details of ports and locations are hidden from user processes. Some run-time functions are also provided by daemons for example, broadcasting a message, synchronization, process control etc.[Sun90].

6.2 Munin

Munin [Car93, CBZ95, CBZ91, Car95] provides a virtual address space that can be accessed by all processes However, Munin supports multiple consis-

tency protocols to improve the performance of the distributed shared memory. Users need to identify the access patterns of the shared variables and annotate each of them with a proper protocol. Those protocols are *conventional*, *read-only*, *migratory* and *write-shared*.

Conventional shared variables are used when multiple processes read the variables and only one process writes to the variables. This type of variables follows the sequential consistency model. *Read-only shared variables* can be written only during initialization. During the rest of execution, read-only shared variables cannot be modified. *Migratory shared variables* are suitable when processes always access the variables exclusively. These variables usually are accessed inside a critical section and protected by locks. The migratory data protocol requires programmers to specify the logical connections between the shared variables and the lock that protect them. The holder of the lock always keeps the most current values of those corresponding variables. *Write-shared variables* are used for an array of variables located on the same page. Multiple processes write to disjoint parts of the array concurrently without using synchronization to coordinate the accesses. The write-shared variables follow the conventional release consistency memory model [GLL+90]. Munin also supports a suite of synchronization primitives such as locks, barriers and condition variables.

Munin uses the page-based implementation. The implementation of Munin's conventional protocol is based on IVY's single writer protocol [LH89] (see Section 1.3.1). Migratory variables are implemented by having the current lock holder send the values of the associated variables to the next lock holder when the current holder passes the ownership of the lock to the next holder. This method eliminates the cost of write misses and invalidating existing copies of the page in the system. The implementation of the write-shared variables is described in Section 1.3.2.

6.3 TreadMarks

TreadMarks [ACD+96] provides a virtual shared address space like IVY and Munin. TreadMarks implement lazy release consistency. A program with a data race condition might get results which programmers do not expect. However, a program without a data race condition runs as if in a sequentially consistent memory model. Unlike Munin, TreadMarks does not have different types of shared variables. All of shared memory follows lazy release consistency. See Section 1.3.2. TreadMarks supports two synchronization primitives, locks and barriers. Section 1.3.4 describes the implementation of locks and barriers.

6.4 Calypso

Calypso [BDK95, DKR95] is a software system for writing and executing parallel programs on a non-dedicated platform. Calypso provides an abstraction of a distributed shared memory model with an unbounded number of virtual processors. Programs are executed by steps. Each step can be either sequential or parallel. In each parallel step, more than one thread works on the computation. The number of threads created in a parallel step can be independent of the number of physical processors in the system. Calypso also provides fault-tolerance and adaptive load balancing functions to avoid execution on very slow processors and to recover from processor failure. Read-write conflicts are illegal amongst concurrent threads. In a parallel step, a process can only read a value written in a previous step. A written value in the current step is not visible to other threads until the following step.

Computation in Calypso is done by one manager and a dynamically changing set of worker processes. The manager is reliable but workers can be slow, fast or even crash. The *two-phase idempotent execution strategy* [KPS90] is

employed. That is, multiple copies of a thread can coexist in the system but the effect is the same as in the case when only one copy of a thread is executed on a non-faulty worker. The manager assigns copies of threads to workers following an eager scheduling strategy [BDK95].

A segment of virtual shared memory is used as a cache of shared memory. Invalid pages are protected. When a worker accesses protected page, the system catches the system fault and sends a read request to the manager. The manager keeps a history of shared memory and gives the last version of the page to the requesting worker. When a worker finishes the execution of a thread, it collects the changes it has made to shared memory and sends them to the manager. The manager accepts the first complete execution of each thread segment and discards subsequent ones.

6.4.1 Orca

Orca [BBH+96] is an object-based distributed shared memory system. Instead of a flat shared memory array, threads in Orca communicate with each other by executing the user defined operations of shared objects, which are instances of abstract data types. All operations on shared objects are executed atomically (as in monitors [Hoa75]). Operations are also allowed to block by using condition synchronization.

There are two strategies to implement shared objects. A shared object stays on one processor and remote threads access the object by using remote procedure calls. Alternatively, all processors have a copy of the object. When a read operation is issued on the shared object, the system gets the data from local copy of the object. Write operations on replicated objects are implemented by broadcasting the operation and updating all copies [Kaa92].

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we first identify some weaknesses of modern software distributed shared memory systems: (1) There are no high level synchronization primitives provided. Programmers have to use basic synchronization primitives, for example, barriers and locks, to solve synchronization problems. It is known to be complex. (2) Current multiple writer protocols suffer from the high cost of making a stale page current if many writers write to the page and then read the page.

In order to alleviate the above problems, we propose two mechanisms, user-definable high level synchronization primitives and the single-owner protocol. The first mechanism provides a friendly paradigm with which users may compose their own synchronization primitives. The relationship between synchronization operations and read/write operations on shared memory is depicted by $\mathrm{RC_{HS}}$.

The single-owner protocol is a new multiple writer protocol. In the single-owner protocol there is a designated processor, called the page owner for each page in shared memory. All writers send their diffs to the page

owner. All readers send read requests to the page owner and obtain a set of pseudo pages which cover the diffs from the page owner.

We have shown the single-owner protocol may reduce the number of messages to update all readers from $O(n^2)$ to $\Theta(n)$, when n writers write to a page and later n readers read the page. However, in a few cases the multiple-owner protocols use fewer messages than the single-owner protocol does. The single-owner protocol can still reduce the number of messages in those cases by assigning the page owner to the processor where one of the readers of the page resides. But there are some applications, for example quicksort, that the readers of a page are always fewer than three and are determined at run time. Thus there is no way for programmers to decide the location of the page owner. In such applications, multiple-owner protocols perform better.

In order to take advantage of both protocols, we propose a hybrid owner protocol, which can be used to increase the efficiency in an adaptive way, with some pages managed by the single-owner protocol and some by the multiple-owner protocol.

In our experiments, we implement the single-owner protocol and a multiple-owner protocol, called the lazy invalidate protocol. Performance results show the single-owner protocol outperforms the multiple-owner protocol in most of cases. In the application of the heat flow problem, even though all threads just write to a small piece of the array on a page, the performance of the multiple-owner protocol degrades when the number of processors increases. However, integer sort shows the deficiency of the single-owner protocol. In this case, there is only one thread reading and writing to the shared memory exclusively at any point in time. Even though the multiple-owner protocol suffers from diff accumulation for each updating message, the effect of the extra messages outweighs the size of

messages in our experimental environment. The multiple-owner protocol performs better than the single-owner protocol does for this application.

We also implement a program, Mandelbrot, using a user-definable high level synchronization primitive. We compare this program against a program using only barriers and semaphores. The program with barriers and semaphores performs similarly when using either the single-owner protocol or multiple-owner protocol. The program with the high level synchronization primitive performs 30% better than the program with barriers and semaphores.

Overall, the result shows the single-owner protocol outperforms the multiple-owner protocol in most cases. A hybrid protocol may be used for those cases in which the multiple-owner performs better. User-definable high level synchronization primitives provide a better application programming interface for programmers to solve complicated synchronization problems. A program may also exploit high level synchronization primitives to get better performance.

7.2 Future Work

The research of shared memory systems used to focus on the hardware implementation. Even though this type of shared memory system promises very good speedups, the cost of building them is very high. Software distributed shared memory systems are built on general purpose workstations (or personal computers) connected by a high latency network. Since local networks, workstations and personal computers are prevalent, software distributed shared memory systems may introduce parallel computation technology to a larger variety of users. Cost efficient systems and a good application programming interfaces are essential for software distributed shared memory

systems.

In order to implement software distributed shared memory systems in such a general purpose environment, there are some factors developers need to be concerned about. These machines are prone to fail [BDK95], and are shared by several users. If a thread is running on a heavily loaded machine, it may slow down the computation. An economical way to tolerate faults and to adapt the system to dynamic load change can make software distributed shared memory systems more practical.

Some parts of the system we have developed may be improved. One is to use compiler analysis to reduce the cost for creating diffs. Our current approach inserts a procedure call before each write to the shared memory. The procedure call takes the written address and the size of variable as its parameters. It marks bit maps for the written pages based on the information. However, operations writing to contiguous addresses or writing to the same address repeatly in shared memory may be merged to one procedure call. In the more optimistic case, the compiler may tell the addresses of the diffs at compile time. This can decrease the cost of creating diffs.

Finally messages with diffs sent to page owners may cause a traffic jam in the network when all systems servers try to send those messages to other system servers at the same time. These messages actually can be overlapped with computation. They do not need to arrive at their destinations immediately. Since we use FIFO channels between each pair of system servers, messages for synchronization purposes could be blocked by messages with diffs. An effective way to coordinate these messages may enhance the performance of the single-owner protocol.

Appendix A

Sample of Code

This appendix shows the code for defining `Pool` mentioned at Sec. 5.2.5. This is a full example of a user-definable high level synchronization object.

```
#ifndef _object_class
#define _object_class

class return_class {
public:
  int nrectangles;
  int stack_top;
};

class waiting_proc_class{
public:
  waiting_proc_class(int proc_id, int n)
{id = proc_id; nrectangles = n;};
  waiting_proc_class(){};
  int id;
  int nrectangles;
};

class Pool {
public:
  Pool(int s);
  /* return value:
     (1) if nrectangle < 0 then pop that many items off
 local stack.
     (2) if nrectangle > 0 then add items from parameter to
 the local stack.
     (3) if nrectangle == 0, do nothing
     (4) if stack_top == -1, then thread should exit,
 computation has finished.
  */

  acquire  Return_Class GetAction(int nrectangles);

  release void Done();

  void Init(int nrectangles);
private:
  /* data structure defined by the user */
  int size_of_stack;
  int top;

  /*
     for holding processes requests when the stack is
     empty or some thread is accessing global array
```

```
  */
  queue_class<waiting_proc_class> *mutex;
  queue_class<int> *empty;
  int stack_in_use;

  int num_jobs;
  int *proc_jobs;

  int NPROC;

  void CheckTermination(int proc_id);
  void DistribJobs(int proc_id, int nrectangles);
};
#endif
```

```
#include "queue.H"

template class queue_class<waiting_proc_class>;
template class queue_class<int>;

Pool::Pool(int s)
{
  size_of_stack = s;
  top = 0;

  NPROC = smile_getnsite();
  proc_jobs = new int [NPROC];
  memset (proc_jobs, 0, sizeof(int) * NPROC);
  num_jobs = 0;

  stack_in_use = FALSE;

  //  empty = new Queue_Class<int>(NPROC);
  mutex = new queue_class<waiting_proc_class>(NPROC);
  empty = new queue_class<int>(NPROC);
}
void
Pool::Init(int nrectangles){
  num_jobs = top = nrectangles;
}
void
Pool::GetAction(int nrectangles)
{

  int proc_id = GetCliId();

  num_jobs = num_jobs - proc_jobs [proc_id] + nrectangles;
  proc_jobs[proc_id] = nrectangles;

  if ((nrectangles == 0)&&(top == 0))
    {
      CheckTermination(proc_id);
      return;
    }

  if (stack_in_use) {
  }
  DistribJobs(proc_id, nrectangles);
}
```

```
void
Pool::DistribJobs(int proc_id, int nrectangles)
{

  int quota = num_jobs / NPROC;

  if ((quota == 0)&&(num_jobs > 0)) quota = 1;

  if (nrectangles < quota)
    {
      /* get rectanlges from global stack */
      quota -= nrectangles;
      if (quota > top) quota = top;
    }
  else
    {/*put rectanlges to global stack */
        quota = nrectangles - quota;
if ((top+quota) >= size_of_stack) quota = size_of_stack - top;

quota = -quota;
    }

  if (quota != 0){/* need to update shared stack */
    if (stack_in_use) {
mutex->EnQueue(waiting_proc_class(proc_id, nrectangles));
return;
     }else stack_in_use = TRUE;
  }

  return_class result;
  result.nrectangles = quota;
  result.stack_top = top;

  top -= quota;

  proc_jobs[proc_id] = nrectangles + quota;
  SyncReply (proc_id, &result);
}

void
Pool::Done(){

  stack_in_use = FALSE;
  // let the next thread in the queue go if there are jobs waiting for it.
```

```
  if ((top > 0)&&(empty->N_Objects())){
    int proc_id;
    proc_id = empty->DeQueue();

    DistribJobs(proc_id, 0);

    return;
  }

  int nwaiting, i;
  if (nwaiting = mutex->N_Objects()){

    for (i=0;i<nwaiting;i++){
      waiting_proc_class item;
      item = mutex->DeQueue();

      if ((item.nrectangles > 0)||(top > 0)){

DistribJobs(item.id, item.nrectangles);

if (stack_in_use) return;/* thread will change global stack */
      }else CheckTermination(item.id);
    }
  }

}
void
Pool::CheckTermination(int proc_id)
{
  int nwaiting;

  nwaiting = empty->N_Objects();

  if (nwaiting == (NPROC - 1)) {/* finished */
    int i;

    return_class r;
    int w;

    r.nrectangles = 0;
    r.stack_top = -1;

    for (i=0;i<nwaiting;i++){
```

```
      w = empty->DeQueue();

      SyncReply(w, &r);
    }
    SyncReply(proc_id, &r);
  }else empty->EnQueue(proc_id);
}
```

Bibliography

[ACD+96] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[Adv93] S. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[AG95] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. Technical report, Rice University ECE, 1995.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. the NAS parallel benchmarks. Technical Report RNR-94-007, NASA, 1994.

[BBH+96] H. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, and T. Ruhl. Orca: A portable user-level shared object system. Technical Report IR-408, Vrije University, 1996.

[BDG+91] A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM: Parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratory, 1991.

[BDK95] A. Baratloo, P. Dasgupta, and Z. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceeding of the 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995.

[BH86] J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 1986.

[BM93] O. Babaoglu and K. Marzzullo. *Distributed Systems*, chapter 4. Addison-Wesley, second edition, 1993.

[BZS93] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. Technical Report CMU-CS-93-119, Carnegie-Mellon University, 1993.

[Car93] J. B. Carter. *Efficient Distributed Shared Memory Based on Multi-Protocol Release Consistency*. PhD thesis, Rice University, 1993.

[Car95] J. B. Carter. Design of the Munin distributed shared memory system. *Journal of Parallel and Distributed Computing on Distributed Shared Memory*, 1995.

[CBZ91] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteen Symposium on Operating System Principles(SOSP)*, 1991.

[CBZ95] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[CK96] C. Chu and Z. Kedem. Techniques for improving the performance of multiple writer memory protocols in distributed shared memory systems. In *Proceedings of High Performance Computing*, 1996.

[CKK95] J.B. Carter, D. Khandekar, and L. Kamb. Distributed shared memory: Where we are and where we should be headed. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems*, 1995.

[DKR95] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel processing on networks of workstations: a fault-tolerant, high performance approach. In *Proc. 15th Intl. Conference on Distributed Computing Systems*, 1995.

[DS90] M. Dubois and C. Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transaction on Software Engineering*, June 1990.

[DSB86] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *Proceeding of 13th Annual International Symposium on Computer Architecture*, 1986.

[DSB88] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 1988.

[GBJ+94] A. Geist, A. Beguelin, J. Dongarra W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

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

[GLS94] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1994.

[Goo89] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 61, SCI Committee, 1989.

[Hoa75] C. A. R. Hoare. Monitor, an operating system structuring concept. *Communication of the ACM*, 17:549–557, October 1975.

[Kaa92] M. F. Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1992.

[KCDZ94] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *the 1994 Winter USENIX Conference*, 1994.

[KCZ92] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *the 19th Annual International Symposium on Computer Architecture*, 1992.

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

[KFJ94] P. T. Koch, R. J. Fowler, and E. Jul. Message-driven relaxed consistency in a software distributed shared memory. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation*, 1994.

[KPS90] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proc. 22nd ACM Symp. on Theory of Computing*, 1990.

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

[LDCZ95] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings of Supercomputing*, 1995.

[LDCZ97] H. Lu, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Quantifying the performance differences between PVM and TreadMarks. *Journal of Parallel and Distributed Computation*, 1997.

[LH89] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li88] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, 1988.

[Sch89] C.E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989.

[SD88] C. Scheurich and M. Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceeding of 1988 International Conference on Parallel Processing*, 1988.

[SHT+95] J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical N-body methods: Barnes-Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, June 1995.

[Ste90] W. R. Stevens. *UNIX network programming*, chapter 3. Prentice-Hall,Inc., 1990.

[Sun90] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*, chapter 2. Prentice Hall, 1992.

[WOT+95] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of 22nd Annual International Symposium on Computer Architecture*, 1995.

[ZSB94] M. Zekauskas, W. Sawdon, and B. Bershad. Software write detection for a distributed shared memory. In *Proceedings of USENIX*, 1994.