# A DECENTRALIZED ALGORITHM FOR COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORY

By

LEGAND L. BURGE III

Bachelor of Science

Langston University

Langston, Oklahoma

1992

Submitted to the Faculty of the
Graduate College of
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July, 1995

# A DECENTRALIZED ALGORITHM FOR COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORY

Thesis Approved:

Mitchell L. Neilsen
Thesis Adviser

[illegible]

H. Lu

Thomas C. Collins
Dean of the Graduate College

## ACKNOWLEDGMENTS

I sincerely thank my graduate adviser Dr. Mitchell L. Neilsen for the guidance, help and time he has given me for the completion of my thesis work. His perseverance and hard work inspired me to venture into the advanced aspects of this work. I would like to express my sincere thanks to Dr. George for his direction and leadership. Without the encouragement and help he has given me, the completion of this work would have been impossible. I also sincerely thank Dr. H. Lu for serving on my committee. Her suggestions have helped me to improve the quality of this work.

My special thanks goes to Dr. In Hai Ro, from Langston University, for the support that you have giving me throughout my studies here at Oklahoma State University. I would also like to thank Dr. Massaki Mizuno, from Kansas State University, for allowing me to utilized his simulator for my research.

My respectful thanks goes to my parents Dr. Legand L. Burge Jr. and Mrs. Gwenetta V. Burge for all the love and support they have given me in my life. And, last but certainly not least, I thank all other members of my family for the love, encouragement and confidence they have endorsed in me.

TABLE OF CONTENTS

## LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

# INTRODUCTION

A *sequential computer* executes one CPU instruction at a time. Over the years sequential computers have increased steadily in performance primarily as a result of improvements in digital hardware technology. One major concern of computer designers is that logic and memory devices are approaching ultimate physical limits on their size and speed. While size reductions and speed increases of a few orders of magnitude beyond present levels seem feasible, further improvements in the performance of sequential computers may not be achievable at acceptable cost. A more economic solution is to design systems that can process more than one CPU instruction at a time. This is known as *parallel processing.* Parallel processors are also referred to as *distributed systems.* These systems consists of an interconnected collection of autonomous computers [Sta84]. There are many ways of classifying distributed systems based on their structure or behavior.

Based on Flynn's taxonomy of computer architectures, distributed systems belong to the MIMD (*multiple instruction multiple data*) class of computer architectures [Tan92]. The MIMD class consist of two categories: those that have shared memory (*tightly coupled*), and those that do not (*loosely coupled*).

As shown in Figure 1.1, each category can be furthered divided based on the architecture of the interconnection network. In bus-based systems, there is a single network, backplane, bus, cable, or other medium that connects all machines. Switched systems connect machines by individual wires.

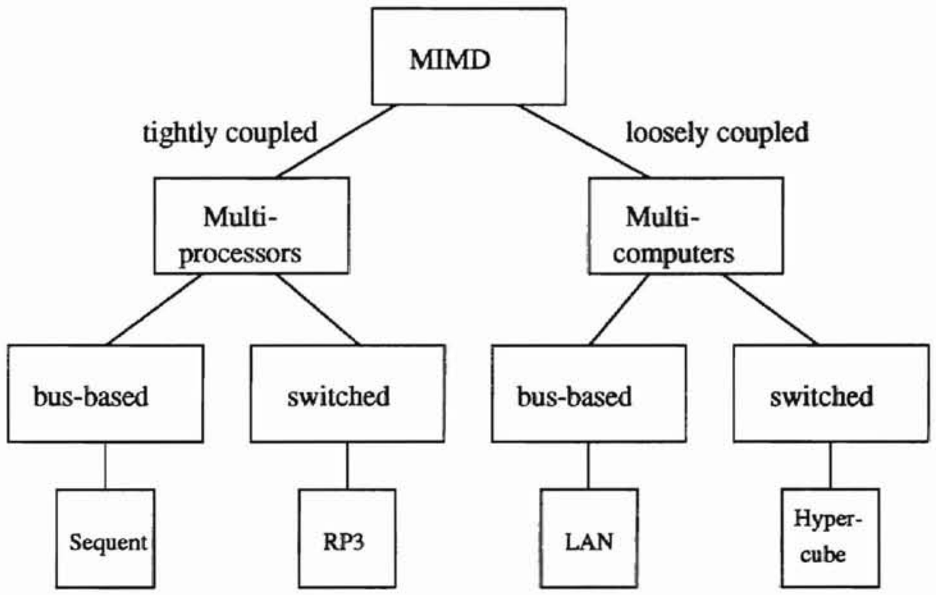

**Figure 1.1 MIMD Hierarchy**

Tightly coupled systems are also referred to as **multiprocessors**. In multiprocessor systems, at least part of the primary memory is shared as shown in Figure 1.2. A system with this shared (*global*) primary memory organization provides a convenient message depository for fast processor to processor communication.

A shared memory can, however, be a major bottleneck, particularly when the processors must share large amounts of information since normally only one processor can access a given memory module at a time [Hay88]. Tightly coupled systems tend to be used more as parallel systems (working on a single problem).

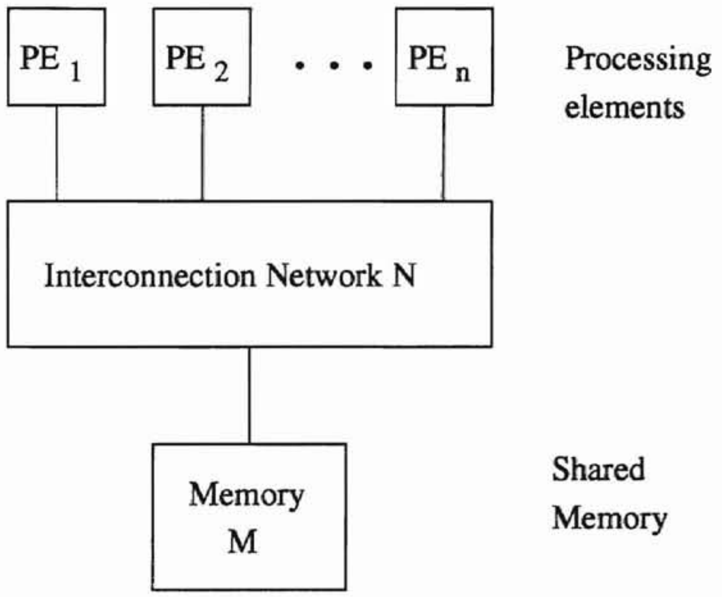

**Figure 1.2 Tightly Coupled System**

Loosely coupled systems are also referred to as **multicomputers**. In multicomputer systems, processors only have access to their own local memories and processors communicate through message passing as shown in the system of Figure 1.3.

Loosely coupled systems are easy to build with the disadvantage of more complex software. Software designed to run on distributed systems give them a high degree of cohesiveness and transparency. Loosely coupled systems tend to be used for working on many unrelated problems.

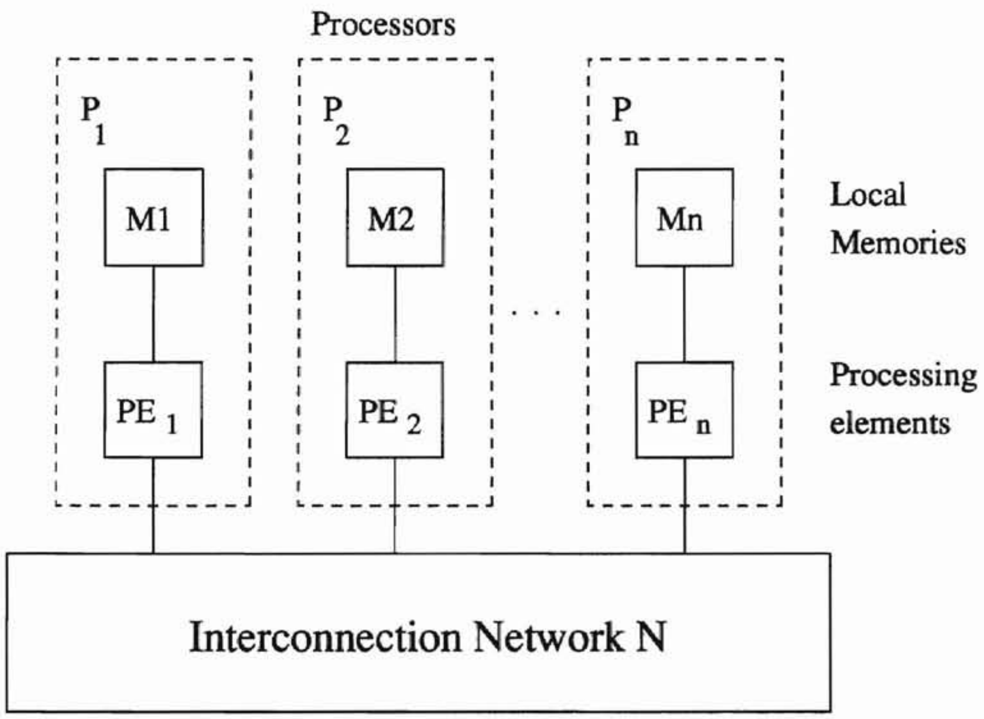

**Figure 1.3 Loosely Coupled System**

Initially, researchers strictly followed the common parallel programming paradigms: shared variables (for tightly coupled systems) and message passing (for loosely coupled systems). More recently, efforts to combine the advantages of multiprocessors (easy to program) and multicomputers (easy to build) have lead to communication paradigms that simulate shared memory on multicomputer systems [SZ90]. These paradigms allow multicomputers to communicate through *Distributed Shared Memory* (DSM). Distributed Shared Memory is an attractive abstraction because it provides processes with uniform access to local and remote information. This uniformity of access simplifies programming, eliminating the need for separate mechanisms to access local state and remote state information. Several techniques have been proposed to allow multicomputers to communicate through Distributed Shared Memory (DSM) [BT91, LH89, MSRN93, MSZ93, SZ90, FL92, AHJ91, GLL$^+$90]. Each technique provides its own level of coherence. An important class of DSM implementations is one which uses cache memories to improve efficiency. Brown, Afek, and Merritt proposed cache-consistency protocols that provide a lower level of coherence. Such protocols are useful for applications that do not require strict consistency among all sites in a distributed system [ABM89, Bro90]. Mizuno, Zhou, Singh, and Neilsen proposed more efficient algorithms which enforce the same level of coherence as Brown's protocol [MSRN93, MSZ93]. These protocols use the abstraction of a single copy of shared memory to enforce sequential consistency. This provides the advantage of a simple implementation and a clean correctness proof. However, a single copy of shared memory could become a bottleneck. Typically, if remote accesses to shared memory is costly, this would also decrease performance and object availability.

## 1.1 Thesis

In this thesis, we present a decentralized cache-consistency protocol for DSM which provides the same level of coherence as the protocols presented in [ABM89, Bro90, MSRN93, MSZ93]. Our protocol distributes the shared objects among all processors in the network providing an increase in performance and object availability. Our protocol is not dependent on the system architecture, therefore allowing the algorithm to scale to a large number of processors more efficiently than the protocols in [ABM89, Bro90, MSRN93, MSZ93]. As memory cost decreases and the cost of communication become more expensive, we show that the increase in memory performance/cost of our protocol is minimal as compared to the reduction in communication cost. We prove that our protocol satisfies a formulation of sequential consistency. Next, we provide an in-depth comparison/analysis of our protocol and the previously proposed protocols. Lastly, we show performance metrics of each protocol and explain which protocol performs better or worse in various situations.

In summary, this thesis is three-fold:

1. To present a decentralized cache-consistency protocol for DSM.
2. To prove that the protocol enforces sequential consistency.
3. To provide a comparison of our protocol with proposed protocols.

## 1.2 Organization

The thesis is divided into the following chapters:

- Chapter 2: A literature review of cache-consistency protocols for DSM is presented.
- Chapter 3: A discussion of the decentralized cache-consistency protocol is presented.
- Chapter 4: A comparison of cache-consistency protocols for DSM is presented.
- Chapter 5: A summary of the thesis and suggestions for future work are presented.
- Appendix A: Related proofs are included.
- Appendix B: Memory and communication costs of the existing protocols are presented.
- Appendix C: DSM simulation parameters are presented.

CHAPTER 2

# LITERATURE REVIEW

The extent to which all processors can be kept busy depends on the computer architecture, the tasks being performed, and the manner in which the task have been programmed. A major concern in designing and programming efficient parallel applications is in avoiding conflicts in the use of shared resources e.g. memory. In order to maintain an appropriate performance level, often multiple copies of shared data are maintained. In most distributed applications, all updates are performed on a primary copy and all reads are performed on a local copy that is cached. The value of a primary copy is replicated to remote cached copies once an update occurs. Replication introduces the problem of having inconsistent copies of the same logical data. Complications also arise because the operations on shared data may not be instantaneous. A memory consistency model defines certain restrictions on the use of shared memory. Applications that adhere to these restrictions are given guarantees about the coherence of that memory. Several notions of consistency have been proposed in the literature to implement DSM [HW90, Lam79, AHJ91, GLL+90, FL92].

## 2.1 Memory Consistency

Herlihy and Wing proposed the idea of *linearizability*, which is a correctness condition for concurrent objects that allows strict consistency providing a high level of coherence. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously. Linearizability is more appropriate for applications such as multiprocessor operating systems in which concurrency is of primary interest.

A correctness condition which provides a less restricted form of consistency than rithm incurs less latency as compared to [Bro90]. As shown in Appendix B, the cost performance of this algorithm also requires an expensive atomic broadcast/multicast.

### Mizuno, Singh, Raynal, and Neilsen Algorithm

Mizuno, Singh, Raynal, and Neilsen proposed a memory consistency protocol in [MSRN93, MSZ93] that allows the same set of sequentially consistent executions as the protocol in [Bro90]. This protocol maintains additional information in shared memory in order to reduce the amount of communication (i.e. no multicasting). The architecture organization consists of a shared memory module (*SMem*), residing at a network processor, and multiple processors. *SMem* keeps track of the most recent write operation on each object as well as the values in the local cache of each processor. This is done by maintaining state information and capturing causal relations among read/write operations at *SMem*. All updates are performed on local cached copies and also at *SMem*. All reads are performed locally if the object is present;otherwise the object is read from *SMem*. After each access to *SMem*, a process is notified of any out-of-date values through an acknowledgment. As shown in Appendix B, this memory consistency protocol uses the communication verses memory and computation trade-off to achieve efficient performance.

## 2.3 Definitions

As stated in the previous section, most DSM implementations are based on cache-consistency protocol which use different variations of the notion of sequential consistency. In this section, we review definitions of consistency on which our implementation is based. Some of the definitions and notations introduced in this section follow [MSRN93, MSZ93]. A shared memory system consists of a set of processors $P$ and a memory $M$. Each processor in $P$ may execute a sequence of read and write operations on objects in $M$. A write operation by processor $i$ on an object $x$ is denoted by $w_i(x)v$, where $v$ is the value written on $x$ by this operation. A read operation on $x$ by $i$ is denoted by $r_i(x)u$, where $u$ is the value of $x$ returned by this operation. For simplicity, we assume that all values written by read and write operations are distint.

An *execution history* of a shared memory system is a poset $\hat{U} = (U, \rightarrow_U)$, where $U$ is a set of read and write operations and $\rightarrow_U$ is an irreflexive and antisymmetric relation on $U$; that is, $\rightarrow_U$ is a partial order on $U$. In the following we give some definitions:

- We say that an execution history $\hat{U} = (U, \rightarrow_U)$ is *processor-ordered* if the operations of each processor in $U$ are totally ordered by $\rightarrow_U$.

- An execution history $\hat{S} = (S, \rightarrow_S)$ is a *sequential history* if it is processor-ordered and $\rightarrow_S$ is a total order.

- A sequential history $\hat{S} = (S, \rightarrow_S)$ is legal if for every read operation $r(x)v$ in $S$, there exists a write operation $w(x)v$ such that $w(x)v \rightarrow_S r(x)v$ and there does not exist a write operation $w(x)u$ such that $w(x)v \rightarrow_S w(x)u \rightarrow_S r(x)v$.

- A *restriction* of $\hat{V} = (V, \rightarrow_V)$ to the set $U$, where $U \subseteq V$, is an execution history $\hat{U} = (U, \rightarrow_U)$ such that for any operations $o$ and $o'$ in $U$, $o \rightarrow_U o'$ iff $o \rightarrow_V o'$.

- We define $\hat{U} \mid i$ to be the restriction of history $\hat{U}$ to the set of operations performed by $i$.

- Two execution histories $\hat{S}$ and $\hat{U}$ are *equivalent* if for every processor $i$, $\hat{S} \mid i = \hat{U} \mid i$.

- Two execution histories $\hat{S} = (S, \rightarrow_S)$ and $\hat{U} = (U, \rightarrow_U)$ are *result-equivalent* if $S = U$; that is, corresponding read operations return the same value and corresponding write operations write the same value on both $\hat{S}$ and $\hat{U}$. For example, $w_1(x)1, w_2(x)2, r_2(x)1$ and $w_1(x)1, r_2(x)1, w_2(x)2$ are result-equivalent but not equivalent.

- $\hat{U} = (U, \rightarrow_U)$ *respects* $\hat{V} = (V, \rightarrow_V)$ if $V \subseteq U$ and for any two operations $o$ and $o'$ in $V$, if $o \rightarrow_V o'$ then $o \rightarrow_U o'$

**Definition 1:** A memory $M$ is consistent if for each of its execution histories $H$, there exists a legal sequential execution history $\hat{WR} = (WR, \rightarrow_{WR})$, where $WR$ is the set of all read and write operations in $\hat{H}$, such that $\hat{H}$ and $\hat{WR}$ are equivalent.

**Definition 2:** A memory $M$ is consistent if for each of its execution histories $H$, there exists a legal sequential history $\hat{W} = (W, \rightarrow_W)$, where $W$ is the set of all write operations in $\hat{H}$, such that the following property holds for each processor $i$

(a) Let $WR_i = W \cup R_i$, where $R_i$ is the set of read operations performed by processor $i$ in $\hat{H}$. Then, there exists a legal sequential history $\hat{WR} = (WR, \rightarrow_{WR})$ such that $\hat{WR_i}$ respects $\hat{W}$ and $\hat{H} \mid i = \hat{WR_i} \mid i$.

It has been shown in [MSRN93] that Definitions 1 and 2 are equivalent. Definition 1 considers a sequential history for the entire system. This consists of the read and write operations issued by all processors. Definition 2 considers a sequential history for each processor $i$. This consist of the write operations issued by all the processors and the read operations issued only by processor $i$. We will use the definitions in Appendix A, to prove that our protocol satisfies a formulation of sequential consistency.

CHAPTER 3

# PROBLEM STATEMENT

The previously proposed protocols in [Bro90, ABM89, MSRN93, MSZ93], each require the abstraction of a single processor centralized memory to enforce the real-time ordering on writes. As this simplifies the implementation and provides a clean correctness argument, in reality this strategy would perform poorly. Particularly, as the number of processors/objects increase, and each processor accesses shared memory more frequently. In order to maintain an efficient performance, [Bro90, ABM89, MSRN93, MSZ93] assume the architecture consists of a set of processors connected by a shared bus. In this thesis, we consider a larger scale system architecture in which computers are logically fully-connected and communicate over costly point-to-point links. Due to the cost of remote accesses, a single processor centralized memory strategy would become a bottleneck; decreasing the performance and object availability.

In this chapter, we present a decentralized cache-consistency protocol for DSM which manages objects distributed among all processors in the system. This provides an increase in access performance due to the locality of reference. It also allows the algorithm to scale to a large number of processors/objects more efficiently than the previous protocols, by avoiding the bottleneck of a single processor centralized memory. Our protocol preservers the real-time ordering on write operations, and allows the same set of sequentially consistent executions as [Bro90, MSRN93, MSZ93] without requiring atomic broadcast/multicast. As memory cost decrease and the cost of communication become more expensive, we show that the increase in memory performance/cost of our protocol is minimal as compared to the reduction in communication cost. In the following sections we give an overview of the protocol, followed by a description of two implementations of the the protocol. Finally, we show the performance of the protocol in terms of memory and communication cost.

## 3.1 Overview of Protocol

We assume that the system consist of logically fully-connected autonomous computers communicating across point-to-point links. Each processor contains two threads of control, a *Processor Manager* and an *Object Manager*, which share a single address space. This address space contains state information to capture causal relations of read/write operations to an object, and to notify a processor of invalid objects. Each processor initially owns a set of objects, and no two processors own the same object. An owner of an object, owns the most consistent version of an object, as updates to an object are only allowed to be processed by the owner of that object. This allows the real-time ordering on write operations to an object to be preserved; but only with respect to the owner of the object. Therefore, all other processors only maintain local cache copies. Each processor manager communicates with the owner of an object for a read/write request, if:

1. During a read to an object not currently owned, the value in the cache is invalid.

2. A write operation is issued to an object not owned by the current process.

Otherwise, the read/write operation is performed locally.

### 3.1.1 Distributed Manager Implementations

In the next section, we describe two management schemes used to keep track of the owner of an object. A primary problem with distributed manager schemes is the initial distribution of objects. As show in Figure 4.7 , an optimal solution would be to distribute an object to a processor who accesses the object most frequent.

### Fixed Distributed Manager

The fixed distributed manager scheme distributes the central manager's (*SMem*) role to every processor in the system, thereby avoiding a single processor bottleneck situation. In this scheme, every processor keeps track of owners of a predetermined set of objects (determined by a mapping function $H$) [LH89]. The primary difficulty in such a scheme is choosing an appropriate mapping from objects to processors. If we assume there are $M$ objects in the system and $I = \{1, ..., M\}$. $H$ is defined as a hashing function such that

$$H(p) = p \ mod \ N$$

where $p \in I$ and $N$ is the number of processors. Therefore, when processor $i$ requests to access an object $p$, processor $i$ contacts the object manager $H(p)$, and the protocol proceeds as in the centralized protocol in [MSRN93, MSZ93].

### Dynamic Distributed Manager

In the dynamic distributed manager scheme, every processor keeps track of the ownership of an object in its local cache. This is maintained through the use of the vector *Probowner* [LH89]. The value *Probowner*$[o]$ contains the owner of object $o$. As processors that frequently access an object can cause the object to migrate, this value can either be the true owner or the probable owner of an object. This value is used as a hint to locate the true owner of an object.

When a processor wants to perform a remote operation on some object $o$, it sends a request to the processor $i$ indicated by the *Probowner*$[o]$ field. Upon receipt of the request, if processor $i$ is the true owner of the object the algorithm proceeds as in the centralized protocol described in [MSRN93, MSZ93]. Otherwise, processor $i$ forwards the request to the processor indicated in its *Probowner*$[o]$ field. This continues until the true owner of the object is found. The hint in the *Probowner*$[o]$ is updated after every remote operation to object $o$. In Appendix A, we show that the implementation of the dynamic distributed manager algorithm requires at most $(N - 1)$ forwarding request messages to locate an owner of an object in a system containing $N$ processors. In the optimal case, only one extra message is require to forward a request; assuming the hint of the probable owner is correct. Because, the hints are updated as a side effect of different migration policies, the average number of messages required should be much less.

### Migration Policies

Our dynamic distributed manager scheme allows objects to migrated between processors. This introduces the notion of a migration policy which could upgrade or degrade the performance of the protocol due to the locality of reference. There are two policies that can be used: *Random Policy* and *Threshold Policy.* Our thesis is only concerned with the threshold policy. We consider migration on read, write, and read/write accesses.

- *Random Policy* - The random policy is a simple migration scheme that uses no state information. An object $o$ is simply migrated to process $i$ after process $i$ request a remote operation on object $o$. The problem with this approach is that useless object migration can occur when an object is migrated to a processor that doesn't access it frequently.

- *Threshold Policy* - The problem of useless object migration under the random policy can be avoided by maintaining statistical information of an object most frequently accessed by a processor. Based on locality of reference, this strategy chooses the best processor to engage in migration of an object.

This is very costly in terms of memory. Each processor must maintain a threshold vector $T$ of size $N \times M$; where $N$ denotes the number of processors and $M$ denotes the number of objects. Moreover, $T[p, o]$ contains the expected number of accesses by processor $p$ on object $o$.

### 3.1.2 Data Structures

Each processor manages the following data structures:

Let $N$ denote the number of processors and $M$ denote the number of objects.

Fixed Distributed Manager Scheme

1. Memory area $C[M]$. $C_i[M]$ contains the values cached at processor $i$.
2. One-dimensional array $Causal[M]$, used to capture causal relations among write operations. $Causal_i[o]$ keeps the version number of the most recent write on object $o$ of processor $i$.
3. A set of valid cache objects *valid*. The set $valid_i$ is initialized to the objects owned by processor $i$.

Dynamic Distributed Manager Scheme

1. Same as the Fixed Manager Scheme.
2. One-dimensional array $Probowner[M]$. Entry $Probowner_i[o]$ contains the hint of the owner of object $o$ by processor $i$.

## 3.2 Description of Protocol

In this section, we provide the actual description of the protocol using a syntax similar to the C programming language. We denote all elements in a one dimensional array $R$

by $R[*]$. Note that all operations on an object local or remote are executed atomically. Therefore, simultaneous updates to local memory by the processor manager or the object manager are synchronized.

### Variable Definitions

In this section, we define and motivate all variables used in the our description of the protocol.

1. Let $x$ be an integer value denoting the object to access in the cache.
2. Let $v$ represent any data structure or block of data structures to be stored in shared memory.
3. Let *valid* represent a set of integer values to denote the valid objects stored in the local cache.
4. Let *Causal* be an integer vector used to capture causal relations among write operations to a shared object.
5. Let $C$ be a vector of the type $v$ to represent the shared objects maintained in the cache.
6. Let $i$ and $j$ be integer values to denote the processor id.
7. Let *Probowner* be an integer vector used to denote the owner of an object.

## Fixed Distributed Manager

In this section we provide a description of the decentralized protocol using the fixed distributed manager scheme.

OBJECT MANAGER at $processor_i$:

```
    Process [$write, j, x, v, Causal_j[*]$ ] message from $processor_j$ ::
        $C_i[x] = v$;
        increment($Causal_i[x]$);
        Invalidate($Invalid_i$);
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        send[$Causal_i[*]$] message to $processor_j$

    Process [$read, j, x, Causal_j[*]$ ] message from $processor_j$ ::
        Invalidate($Invalid_i$);
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        send[$C_i[x], Causal_i[*]$] message to $processor_j$

    Procedure Invalidate(var $Invalid$) ::
        $Invalid = \emptyset$;
        For each $y \in M$, $y \neq x$ do
            if ($Causal_i[y] < Causal_j[y]$) then
             $Causal_i[y] = 0$;
             $Invalid_i = Invalid_i \cup \{y\}$
            endif
        enddo
```

PROCESS MANAGER at $processor_i$:

```
    write ($x, v$) ::
        if ($H(x) \neq i$) then
            send[$write, i, x, v, Causal_i[*]$] message to $processor_{H(x)}$
            receive[$Causal_j[*]$] message from $processor_{H(x)}$
            Invalidate($Invalid_i$);
            $Causal_i[x] = Causal_j[x]$;
            $valid_i = valid_i - Invalid_i$;
        else
            increment ($Causal_i[x]$);
        endif
        $valid_i = valid_i \cup \{x\}$;
        $C_i[x] = v$;
```

```
**read**$(x)$ ::
    if $x \notin Valid_i$ then
        send $[read, j, x, Causal_i[*]~]$ **message to** $processor_{H(x)}$
        receive $[v, Causal_j[*]~]$ **message from** $processor_{H(x)}$
        Invalidate($Invalid_i$);
        $Causal_i[x] = Causal_j[x]$;
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        $C_i[x] = v$;
    endif
    return($C_i[x]$);
```

### Dynamic Distributed Manager

In this section we provide a description of the decentralized protocol using the dynamic distributed manager scheme.

OBJECT MANAGER at $processor_i$:

```
**Process** $[write/forwardw, j, x, v, Causal_j[*]~]$ **message from** $processor_j$ ::
    if ($Probowner_i[x] == i$) then
        **checkthreshold**$(x, j)$;
        $C_i[x] = v$;
        increment($Causal_i[x]$);
        Invalidate($Invalid_i$);
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        send$[Causal_i[*], Probowner_i[x]]$ **message to** $processor_j$
    else
        send$[forwardw, j, x, v, Causal_j[*]]$ **message to** $processor_{Probowner_i[x]}$

**Process** $[read/forwardr, j, x, Causal_j[*]~]$ **message from** $processor_j$ ::
    if ($Probowner_i[x] == i$) then
        **checkthreshold**$(x, j)$;
        Invalidate($Invalid_i$);
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        send$[C_i[x], Causal_i[*], Probowner_i[x]]$ **message to** $processor_j$
    else
        send$[forwardr, j, x, Causal_j[*]]$ **message to** $processor_{Probowner_i[x]}$

**Procedure** Invalidate(var $Invalid$) ::
    $Invalid = \emptyset$;
    For each $y \in M$, $y \neq x$ do
        if ($Causal_i[y] < Causal_j[y]$) then
         $Causal_i[y] = 0$;
         $Invalid_i = Invalid_i \cup \{y\}$;
        endif
    enddo
```

```
**Procedure** checkthreshold($x, j$) ::
    increment($T_i[j, x]$);
    if $T_i[j, x] > t$ then
        $Probowner_i[x] = j$;
    endif

**Procedure** resetthreshold($x$) ::
    if ($Probowner_i[x] == i$) then
        for each $j \in N$ do
         $T_i[j, x] = 0$;
        enddo
    endif
```

PROCESS MANAGER at $processor_i$:

```
**write** ($x, v$) ::
    if ($Probowner_i[x] \neq i$) then
        send[$write, i, x, v, Causal_i[*]$] **message to** $processor_{Probowner_i[x]}$
        receive($Causal_j[*], owner$) **message from** $processor_{Probowner_i[x]}$
        $Probowner_i[x] = owner$;
        Invalidate($Invalid_i$);
        **resetthreshold($x$);**
        $Causal_i[x] = Causal_j[x]$;
        $valid_i = valid_i - Invalid_i$;
    else
        increment   $Causal_i[x]$;
    endif
    $valid_i = valid_i \cup \{x\}$;
    $C_i[x] = v$;

**read**($x$) ::
    if $X \notin Valid_i$ then
        send [$read, j, x, Causal_i[*]$ ] **message to** $processor_{Probowner_i[x]}$
        receive [$v, Causal_j[*], owner$ ] **message from** $processor_{Probowner_i[x]}$
        $Probowner_i[x] = owner$;
        Invalidate($Invalid_i$);
        **resetthreshold($x$);**
        $Causal_i[x] = Causal_j[x]$;
        $valid_i = (valid_i - Invalid_i) \cup \{x\}$;
        $C_i[x] = v$;
    endif
    return($C_i[x]$);
```

### Cost Performance

Our decentralized protocol requires one round of message exchange for a write operation if the current process is not the owner of the object; else the value is written to the local cache. A read operation requires one round of message exchange if the value in the local cache is not valid and the current process is not the owner of the object; else the value is read from the local cache. We provide a protocol similar to [MSRN93, MSZ93], that does not require an atomic broadcast capability, and utilizes less message rounds due to the locality of reference (refer to Appendix B).

In particular, if we consider performing all read operations, our protocol provides about the same level of performance as the protocols in [MSRN93, MSZ93]. Although on the average, our protocol requires slightly fewer messages than the single shared memory protocols in [MSRN93, MSZ93]. This is because remote reads to objects owned by a process can be performed locally, were as, it must always be performed remotely using the protocols in [MSRN93, MSZ93].

# CHAPTER 4

# PERFORMANCE ANALYSIS AND RESULTS

In this chapter, we simulate our protocol and the protocols presented in [Bro90, MSRN93, MSZ93]. We analyze the behavior of each protocol under various conditions, and show which protocol behaves better or worse using different metrics. Our simulation consists of a process scheduler, which schedules discrete events involving multiple processes. Processes created can communicate by using **send()** and **receive()** fuctions. The simulation provides several process synchronization techniques such as: send/receive, signal/wait, and release/acquire. We run each simulation using a 486DX4 100-MHz computer running the Linux Operating System. We assume the simulation parameters given in Appendix C, and the maximum *duration* of any protocol to be 10000000 (i.e. in simulation ticks).

## 4.1 DSM Simulation

### 4.1.1 Performance Metrics

For each protocol we assume the simulation parameters given in Appendix C. We model the performance of each protocol and provide various metrics such as: local access efficiency, average time for an operation, average wait time for an operation, average number of forward messages per forward request, and the comparison of performance between two algorithms.

Given the total access time $t_a$, and the percentage of local access operations $P_{local}$, we define an access time ratio $r = t_{a_{smem}}/t_{a_{local}}$; where $t_{a_{local}}$ is the Local Read/Write time and $t_{a_{smem}}$ is the Remote Read/Write time. We calculate the Remote Read/Write time as the time to send the request to shared memory, process the request, and receive the result/complete (refer to Appendix C for actual times).

Because Brown's Algorithm requires different access times for remote reading and writing, the access time at shared memory $t_{a_{smem}}$ is defined as follows:

$$t_{a_{smem}} = RemoteReadTime * p + WriteInvalidationTime * (1 - p)$$

where $p$ is the probability of performing a read operation. For all other algorithms, the remote read/write access times are equal (refer to Appendix C). We define the average access time $t_{a_{ave}}$ as:

$$t_{a_{ave}} = P_{local} * t_{a_{local}} + (1 - P_{local}) * t_{a_{smem}} \qquad \text{Eq. 1}$$

We define the local access efficiency $e = t_{a_{local}}/t_{a_{ave}}$, to be the ratio of local access time to the average access time [Hay88]. This determines the factor by which $t_{a_{ave}}$ differs from its minimum possible value $t_{a_{local}}$. From Eq. 1 and $r = t_{a_{smem}}/t_{a_{local}}$, we obtain

$$e = 1/(r + (1 - r) * P_{local}) \qquad \text{Eq. 2}$$

The wait time per operation is defined as the time spent waiting for shared memory to perform the request. Due to contention of processors for shared memory, this could vary among the different algorithms. We relate the performance of two algorithms, say $X$ and $Y$, by showing how much of a percentage faster $X$ is than $Y$. This is denoted as follows

$$P_{faster} = ((ExecTime_Y - ExecTime_X)/ExecTime_X) * 100$$

## 4.2 Analysis

In the next sections, we analyze the behavior each protocol in various conditions, and show which protocol behaves better or worse and by what metrics. Except for as designated in the following figures, we assume:

- the total number of processors is 100.
- the total number of objects is 100.
- the total number of operations is 1000 per processor.
- the probability of performing a read/write operation is equally likely (i.e. 0.5)
- objects are selected from a uniform distribution.

### 4.2.1 Centralized Protocols

In this section we show the performance of the algorithms proposed in [Bro90, MSRN93, MSZ93].

As defined in Eq. 2, $e$ is calculated as a fraction of $P_{local}$. Figure 4.1 shows that it is important to achieve high values of $P_{local}$ (between 0.9 and 1.0), in order to make $e \approx 1$ (i.e. $t_{a_{ave}} \approx t_{a_{local}}$). Because all writes must be performed at shared memory, the local access efficiency directly depends on the probability of reading an object; which can be performed local or remote. If the probability of reading is very high, the local access efficiency increases; while the access time decreases. Based on Figure 4.2, these algorithms will probably perform better—in terms of object accessibility—if the application using these algorithms consisted of more reads than writes.

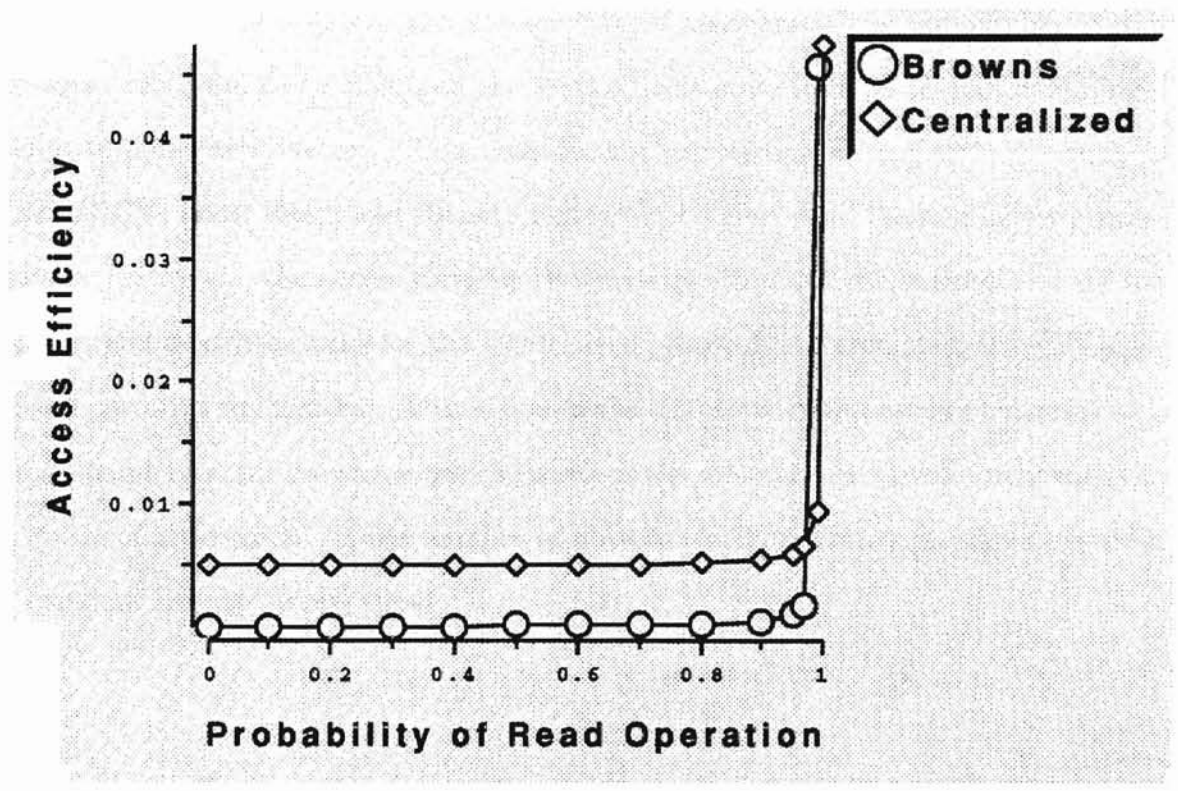

Figure 4.1 Access Efficiency vs the Probability of Reading

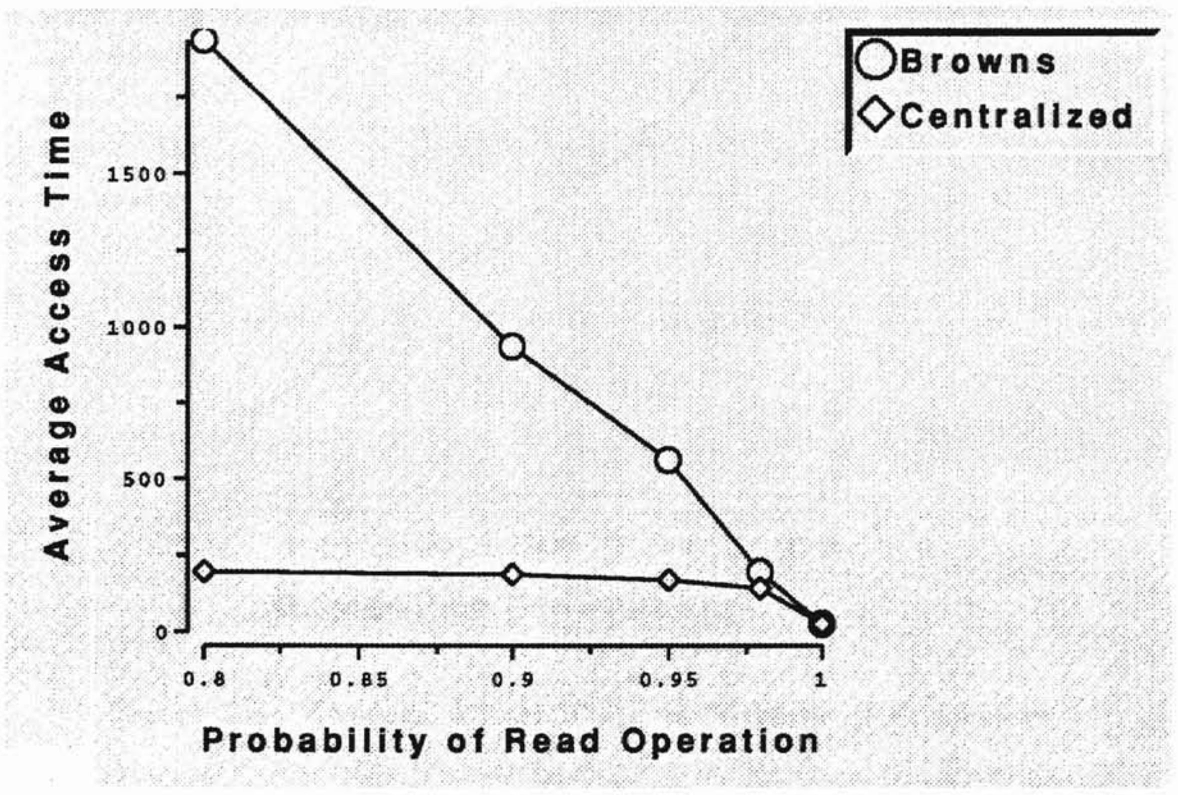

Figure 4.2 Average Access Time vs the Probability of Reading

Figures 4.3 and 4.4 show the average time of an operation as the number of processors scale from 50 to 1000 and the total objects scale from 50 to 400; assuming a point-to-point architecture. If we consider the parameters in Appendix C—such as LATENCY, Local Read, and Remote Read/Write Time—and the number of processors in Figure 4.3, the access time for Browns algorithm can range from 1 to 105105 clocks, and 1 to 202 clocks for the Centralized protocol. As predicted, Brown's algorithm performs much worst. This is due to the expensive multicast implemented as a set of point-to-point messages, which increases the write access time linearly with the number of processors. As the number of objects increase, both algorithms maintain a constant average access time.

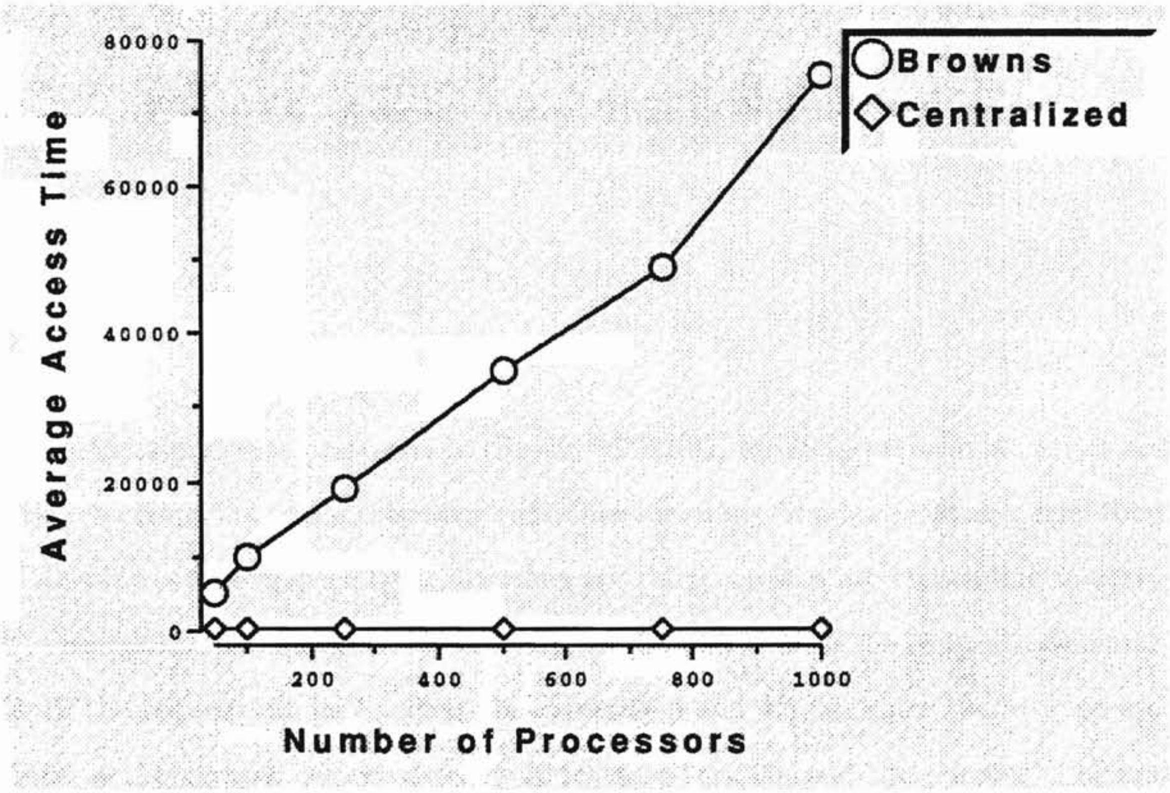

**Figure 4.3 Average Access Time vs Number of Processors**

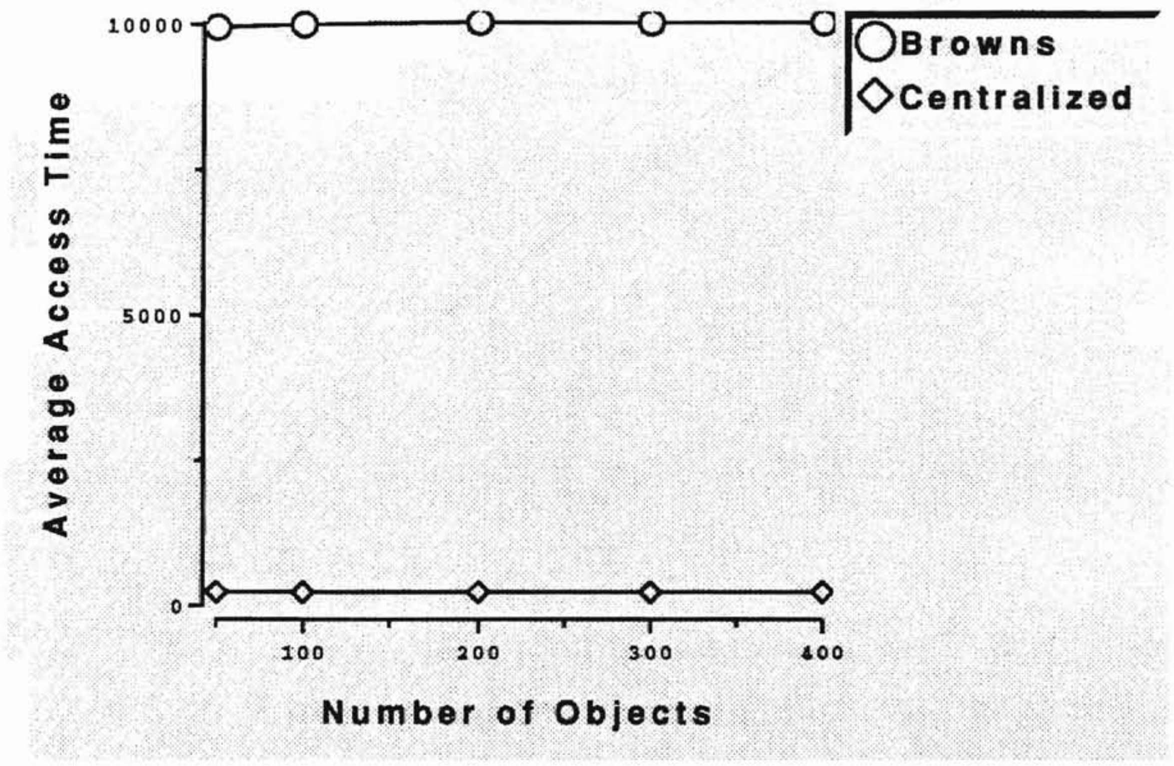

**Figure 4.4 Average Access Time vs Number of Objects**

Both the algorithms presented in [Bro90, MSRN93, MSZ93], perform all writes and reads to/from the single processor centralized memory in a single atomic operation. This creates a bottleneck typically when several processors are contending to access the centralized shared memory. This results in a wait time at the centralized memory until the request can be handled. In Figures 4.5 and 4.6, we show the average wait time as the number of processors scale from 50 to 1000, and the number of objects scale form 50 to 400. We will show that our Decentralized protocol scales the number of processors much better by distributing objects uniformly to all processors; which in turn decreases processor contention, as well as, access time.

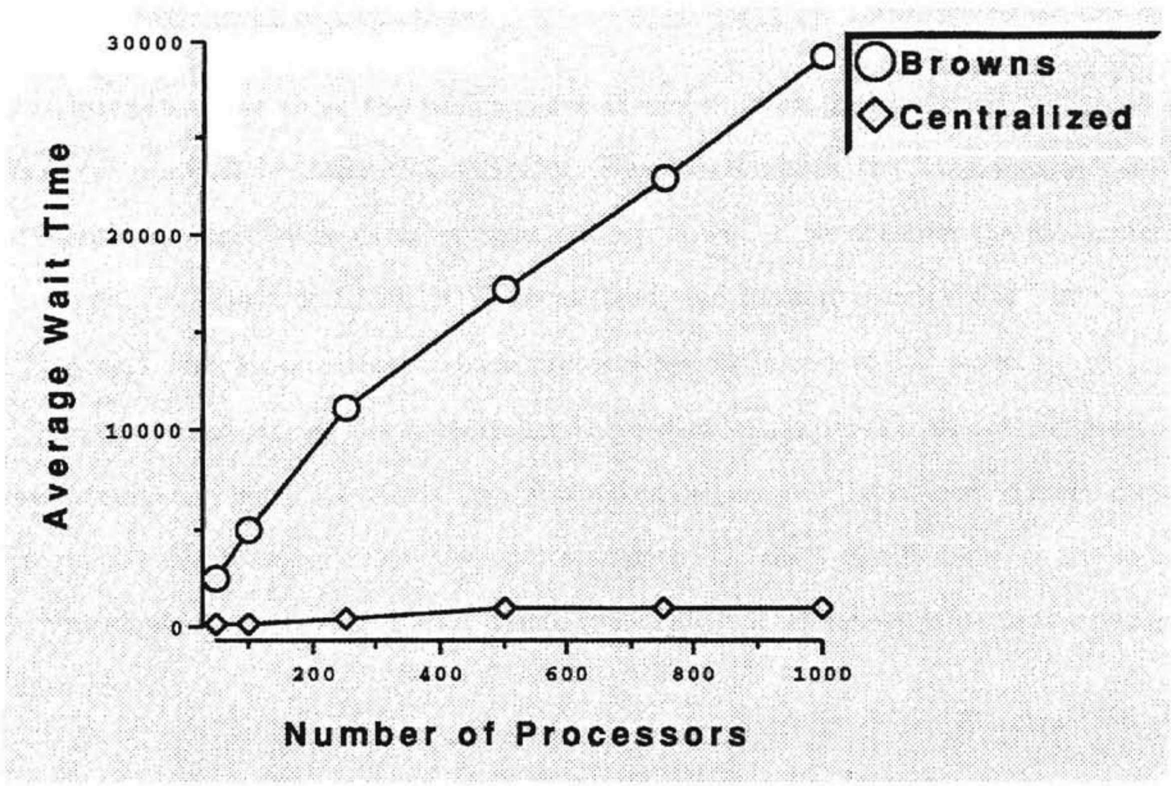

Figure 4.5 Average Wait Time vs Number of Processors

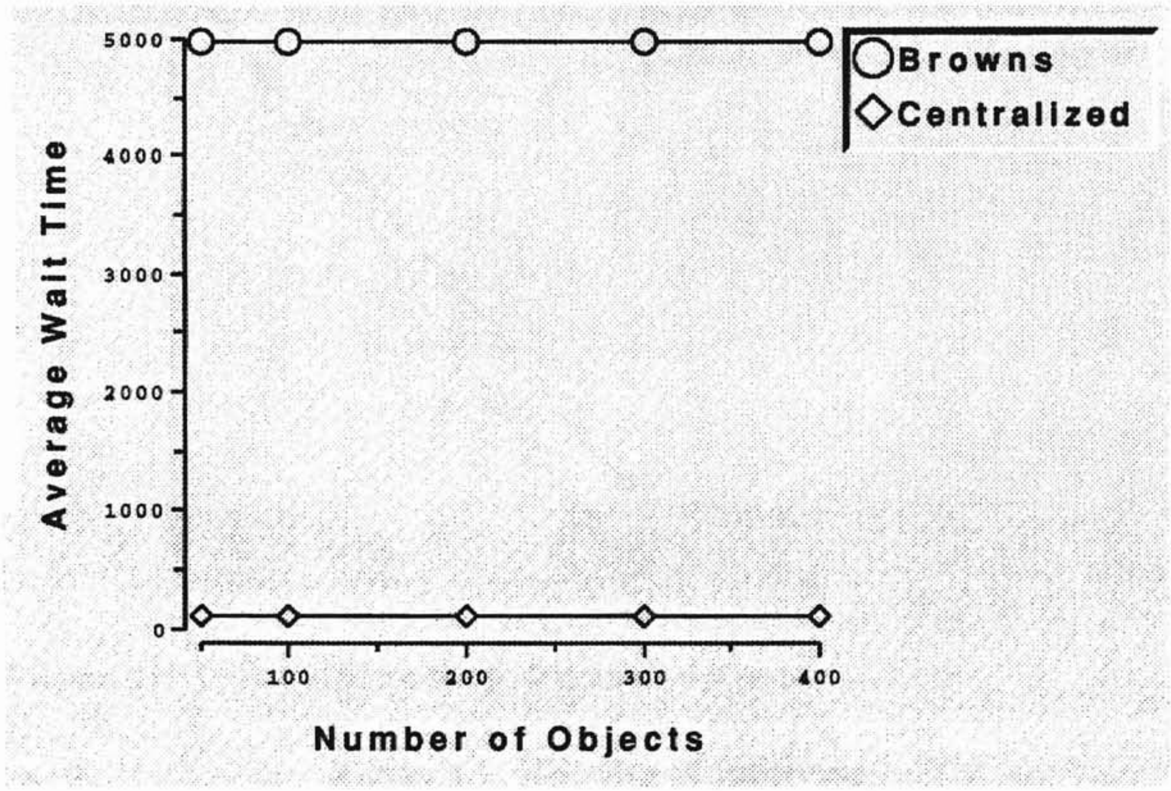

Figure 4.6 Average Wait Time vs Number of Objects

### 4.2.2 Decentralized Protocol

In this section we show the performance of our protocol as compared to the centralized protocol in [MSRN93, MSZ93]. We examine both the fixed manager and dynamic manager schemes to implement our protocol. If we consider the parameters in Appendix C,such as LATENCY, Local Read, and Remote Read/Write Time, then the access time for our decentralized protocol ranges from 1 to 202 ticks.

In certain situations the decentralized algorithm could provide an optimal level of performance. Figure 4.7 shows that if a processor accesses the objects it owns more frequently, the average access time per operation decreases significantly, as the local access efficiency increases. This is due to the locality of references to processor owned data objects.

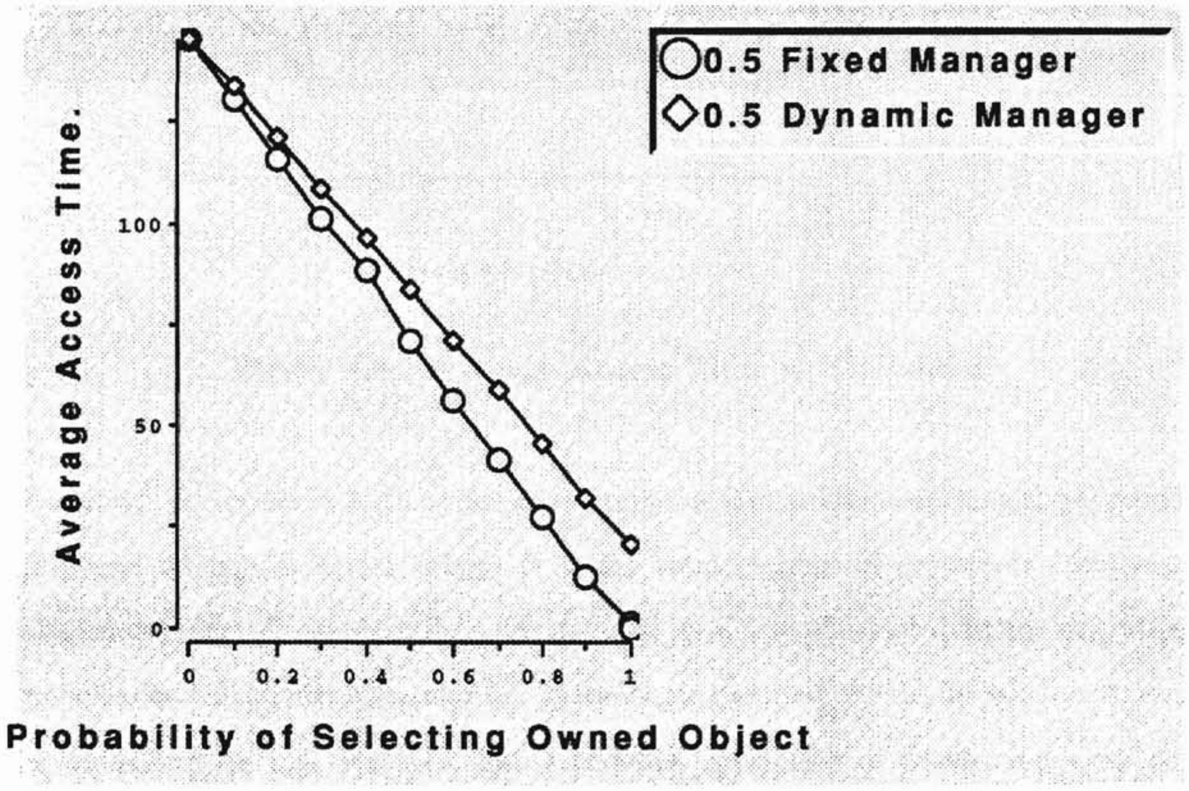

**Figure 4.7 Locality of Reference**

Although migration in the dynamic manager scheme can be performed on read, write, and read/write accesses, Figure 4.8 shows that read/write migration yields a lower average access time for low values of the threshold $t$. As the threshold increases, the average access time for all migration schemes approaches to 83.0.

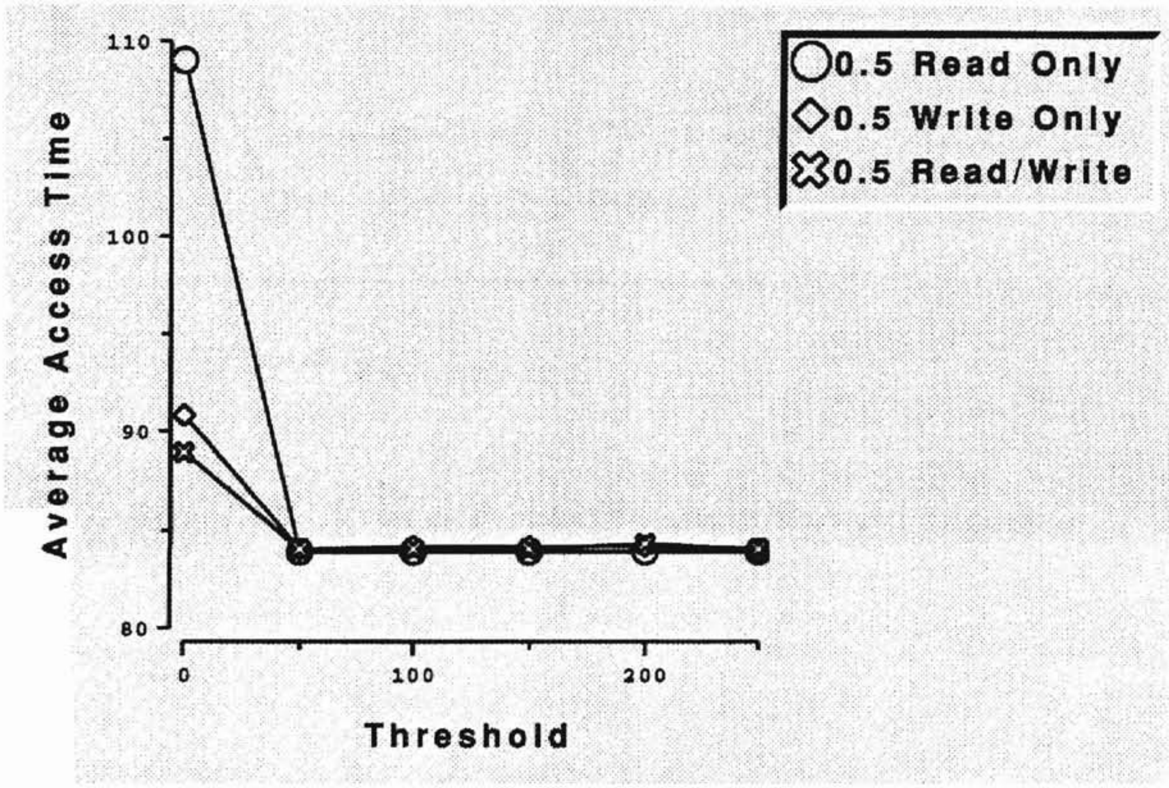

**Figure 4.8 Average Access Time vs Threshold**

As proved in Appendix A, the worst case number of forward request using the dynamic manager scheme is $N - 1$; where $N$ is the total number of processors. Because, the hints in the *Probowner* field are updated as a side effect of different migration policies, overall the average number of messages required should be much less; and is dependent on the threshold level $t$ and the probability of performing a remote operation $(1 - P_{local})$. Figure 4.9 shows the average forward messages per forward request.

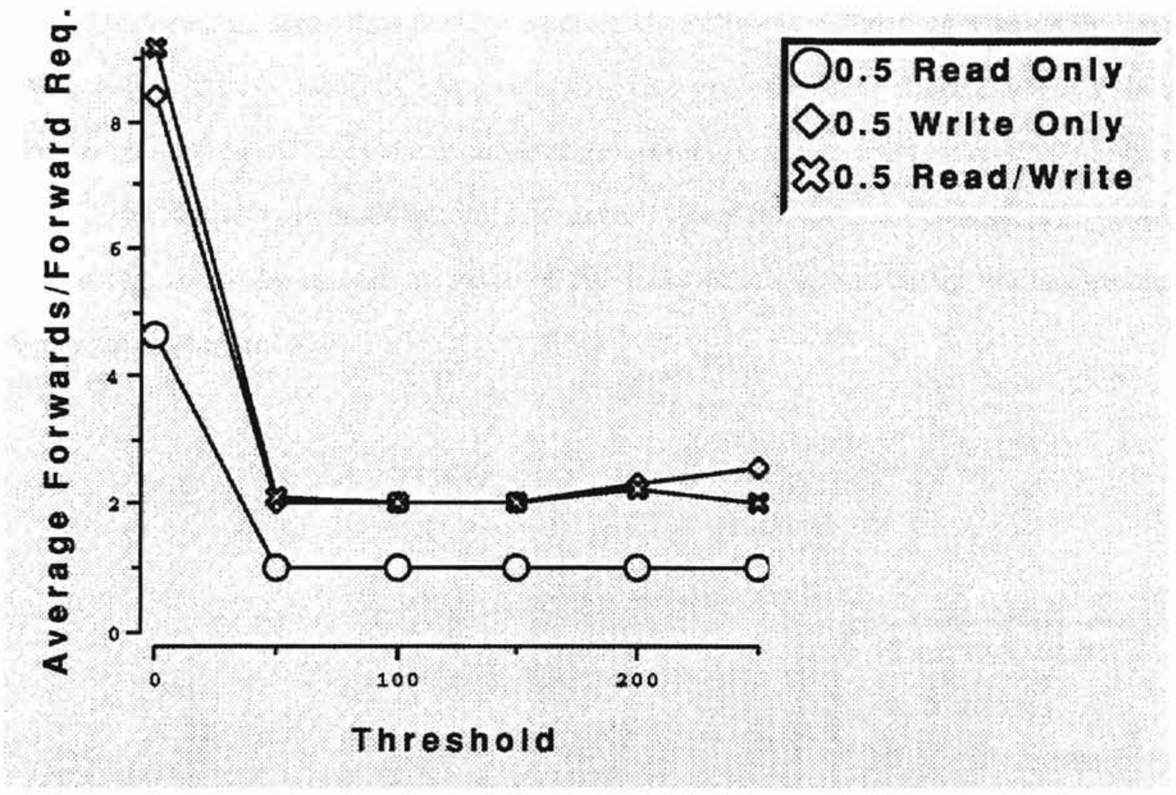

**Figure 4.9 Average Forward Messages/Forward Request**

For the remaining analysis, we assume a 50% chance of selecting an owned object using both the fixed and dynamic manager schemes. We also assume read/write migration for the dynamic manager scheme. As shown in Figures 4.10 and 4.11, the decentralized algorithm performs significantly better. In particular, the local access efficiency $e$ approachs 1 slightly faster than the centralized algorithm (Figure 4.10). This is due to the locality of reference; as reads and writes can both be performed locally if the object is owned by the current processor.

Therefore, our algorithm performs better than the centralized algorithms in cases were reads/writes are equally likely to occur. Our protocol scales more efficiently than the centralized protocol as the number of processors increases from 50 to 1000 (Figure 4.11). Particularly, in maintaining low access times for low numbers of processors; increasing up to the maximum value of 202 ticks which is constantly maintained by the centralized protocol.

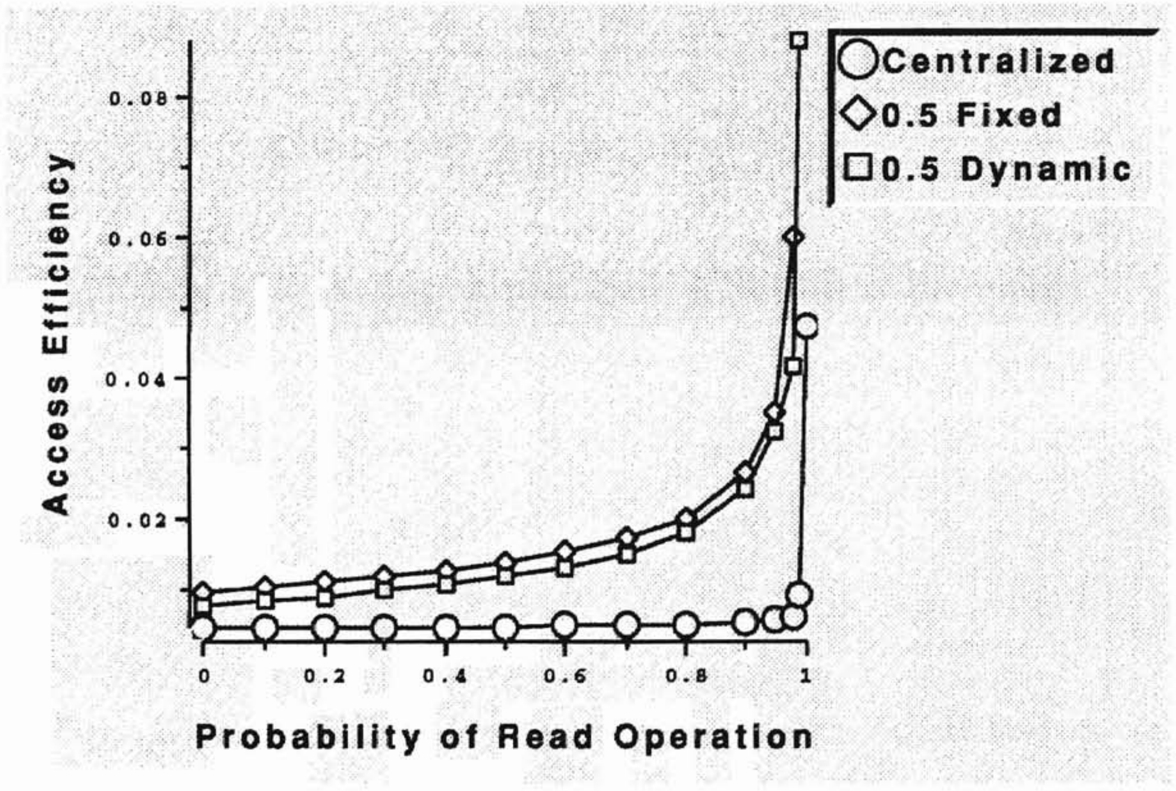

**Figure 4.10 Access Efficiency vs the Probability of Reading**

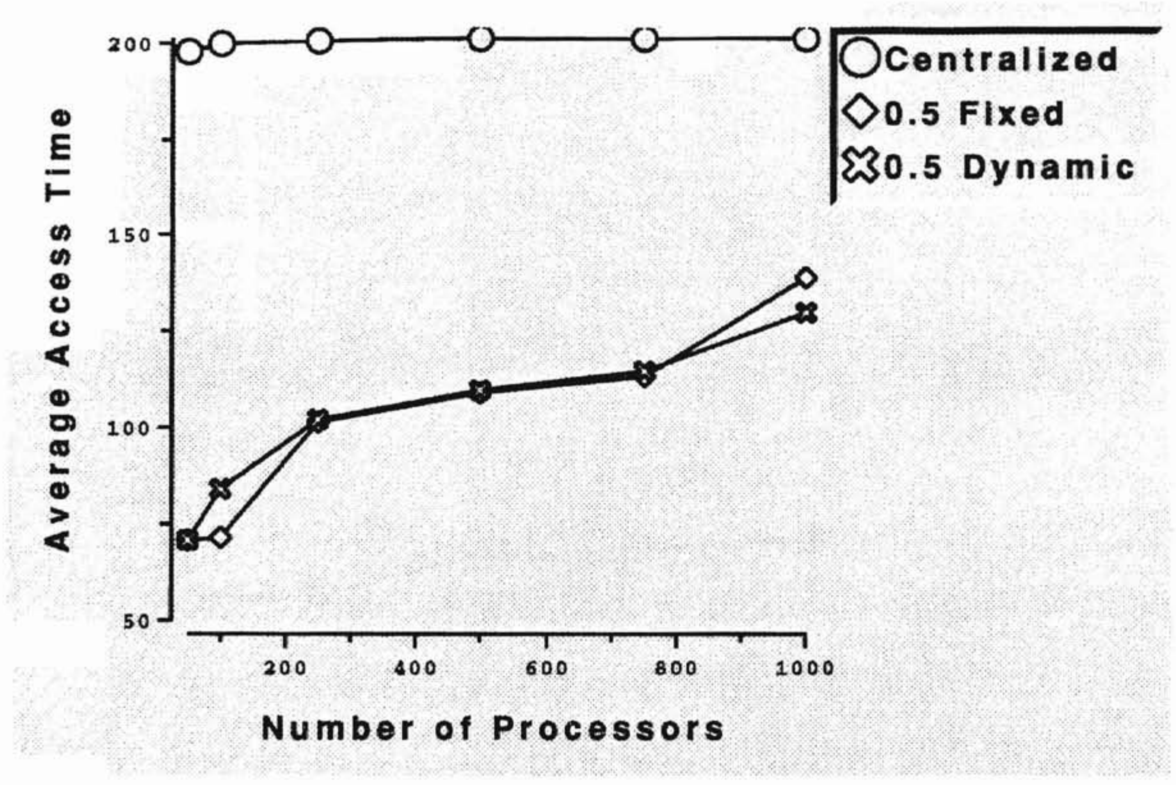

**Figure 4.11** Average Access Time vs Number of Processors

Figure 4.12 shows that for a low number of processors, the decentralized protocol maintains a low wait time; which increases up to a maximum of 202 ticks as the number of processors increase. The centralized protocol maintains a significant increase in waiting time; due to the contention among processors to the centralized shared memory. This in turn increases the average access time; which is at a maximum of 202 clock ticks (refer to Figure 4.11).

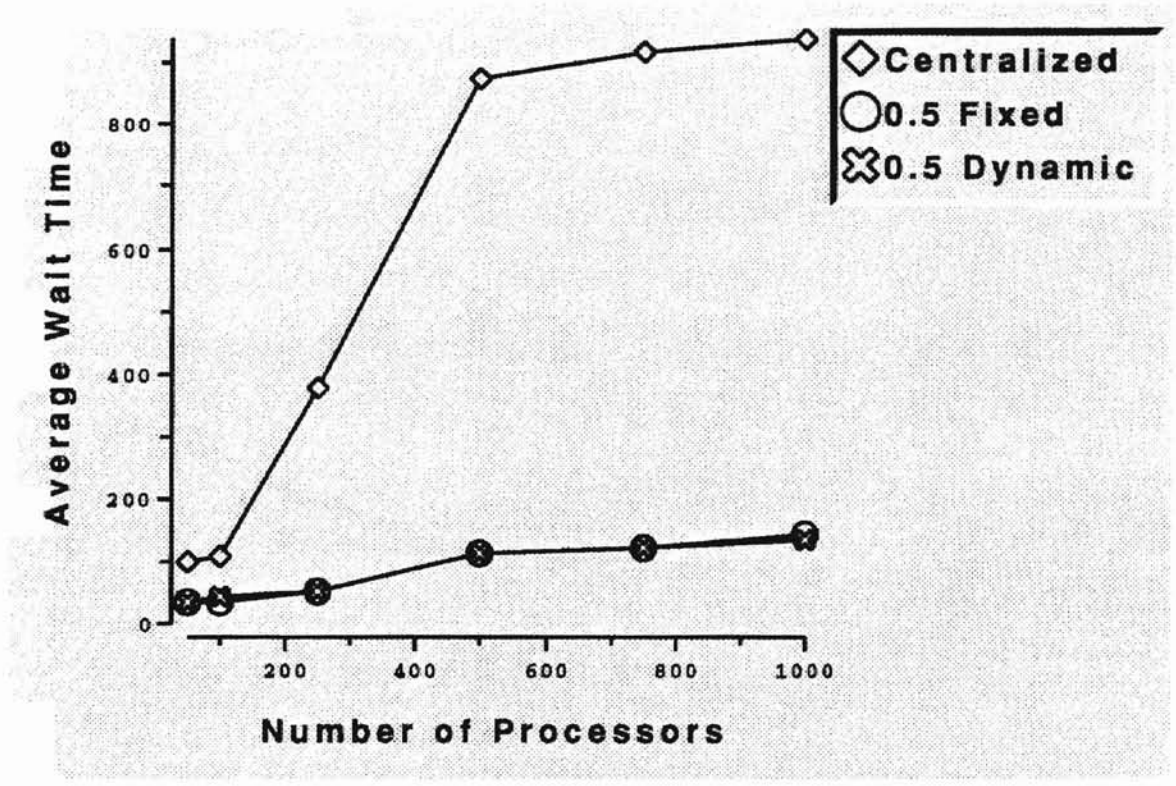

**Figure 4.12 Average Wait Time vs Number of Processors**

For our simulation, the decentralized protocol distributes objects to processors uniformly. Figures 4.13 and 4.14, show that as the number of objects scale from 50 to 400, the average access/waiting time per operation decreases for the decentralized protocol and is constant for the centralized protocol. In the decentralized protocol, if objects are not distributed among processors evenly, or if the number of objects is less than the number of processors, then the possibility of processor contention to the shared objects increases; which in turn increases the access/wait time per operation. Therefore, it is probably best to distributed objects to processors evenly, and to processors which access the object more frequently.

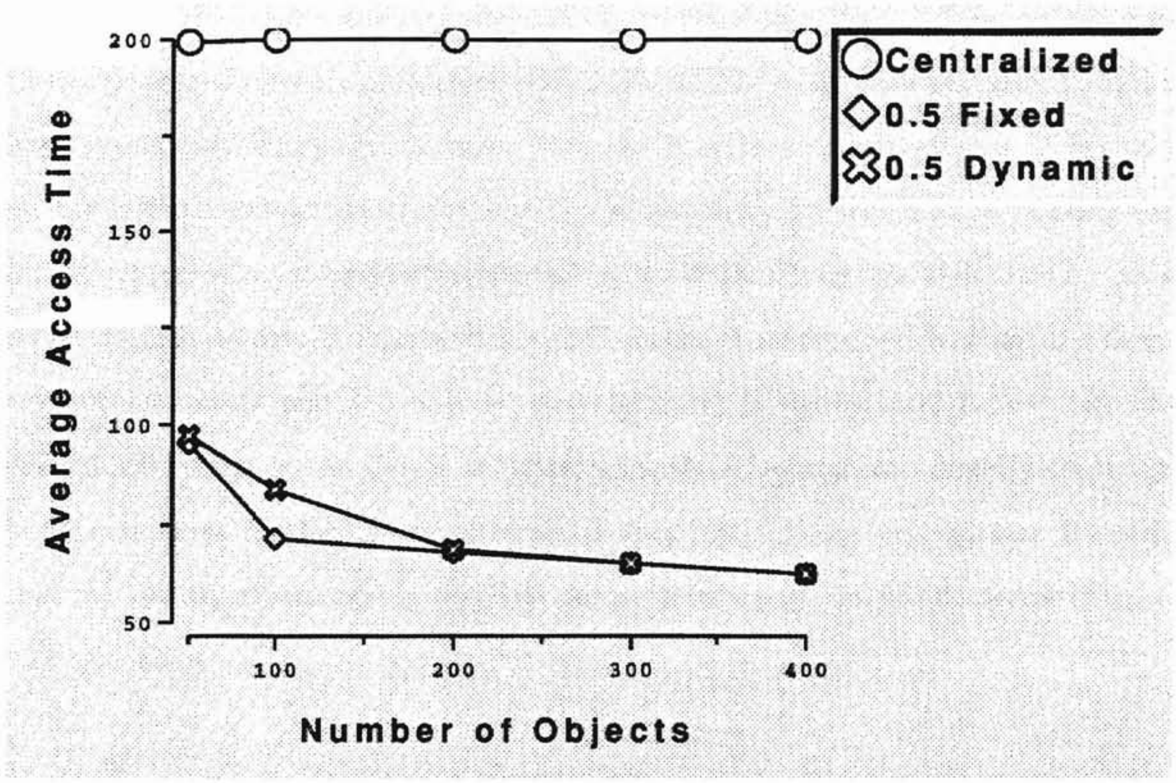

Figure 4.13 Average Access Time vs Number of Operations

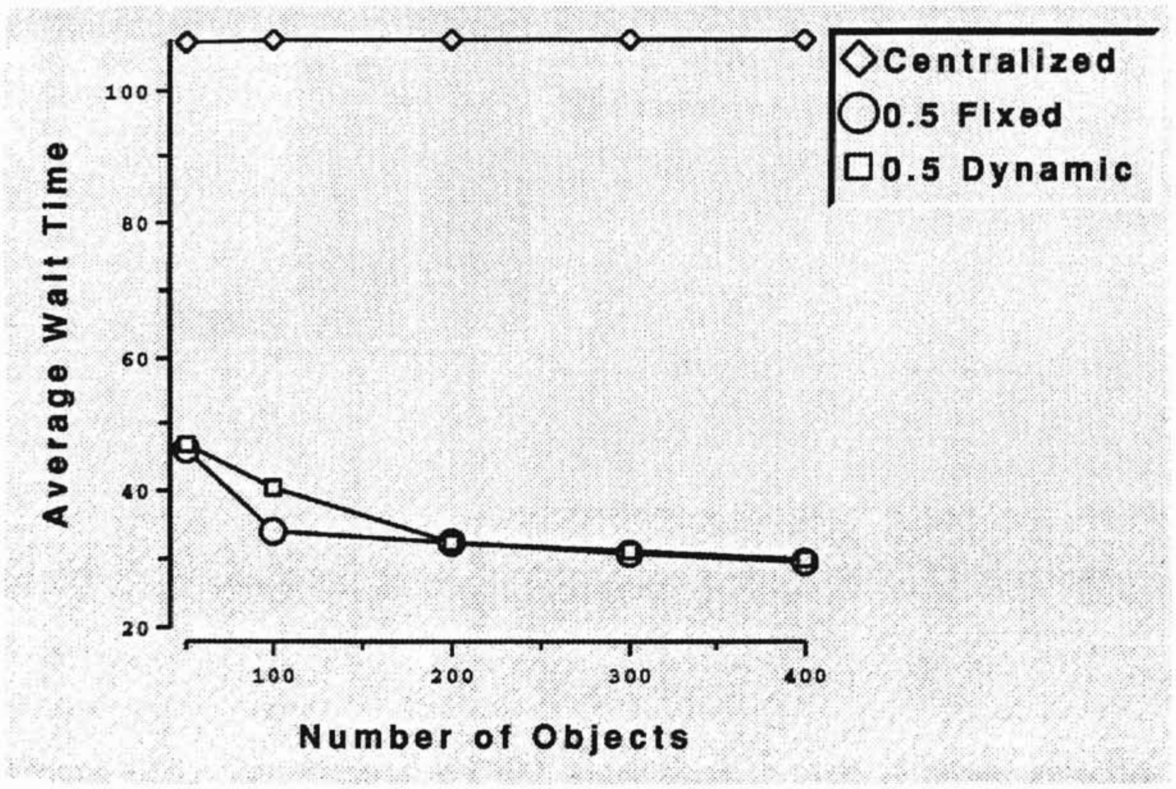

Figure 4.14 Average Wait Time vs Number of Objects

Finally, we compare the performance—in terms of execution—of our protocol to the single shared memory protocol in [MSRN93, MSZ93]. As shown in Figure 4.15, the fixed manager scheme performs better than the dynamic manager scheme. This could be due to the overhead associated with forwarding messages which could increase the locality of reference, as well as, the average access time. Given that 50% of all accesses are performed on owned objects with a 50% chance of reading and writing, and there are 100 processors and 100 objects, our algorithm is significantly faster than the centralized protocols. As shown in Figure 4.15, these assumptions directly effect the performance; as the locality of reference is very high. If the assumptions are such that the locality of reference is very low, our protocol's performance decreases (Figure 4.16 and 4.17).

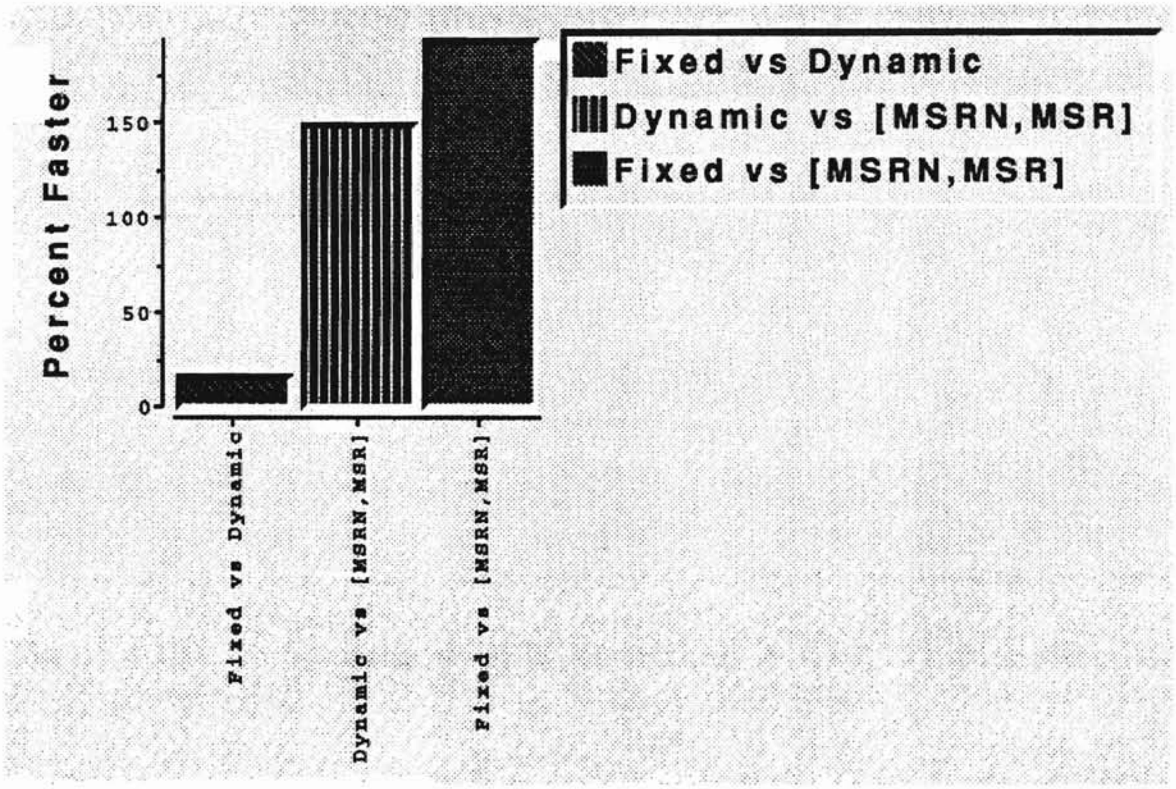

**Figure 4.15 Comparison of DSM Algorithms .5 read, .5 owner selection**

The assumptions made about the simulation throughout the thesis may provide results which are misleading. In order to justify our results, we provide a performance comparison using an 80% probability for a read, and a 0/20% probability of selecting an owned object. As shown in Figure 4.16 and 4.17, the decentralized algorithms still perform better due to the locality of reference; as read and write operations can both be performed locally.

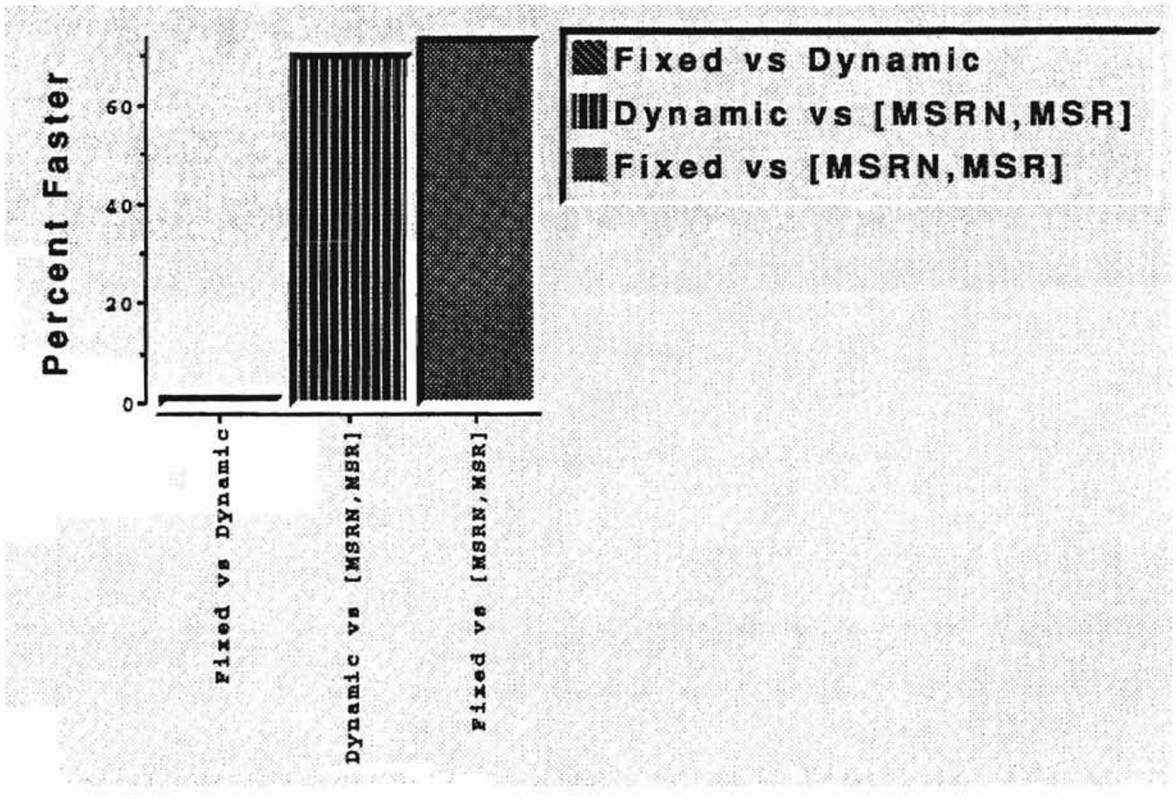

**Figure 4.16 Comparison of DSM Algorithms .8 read, .2 owner selection**

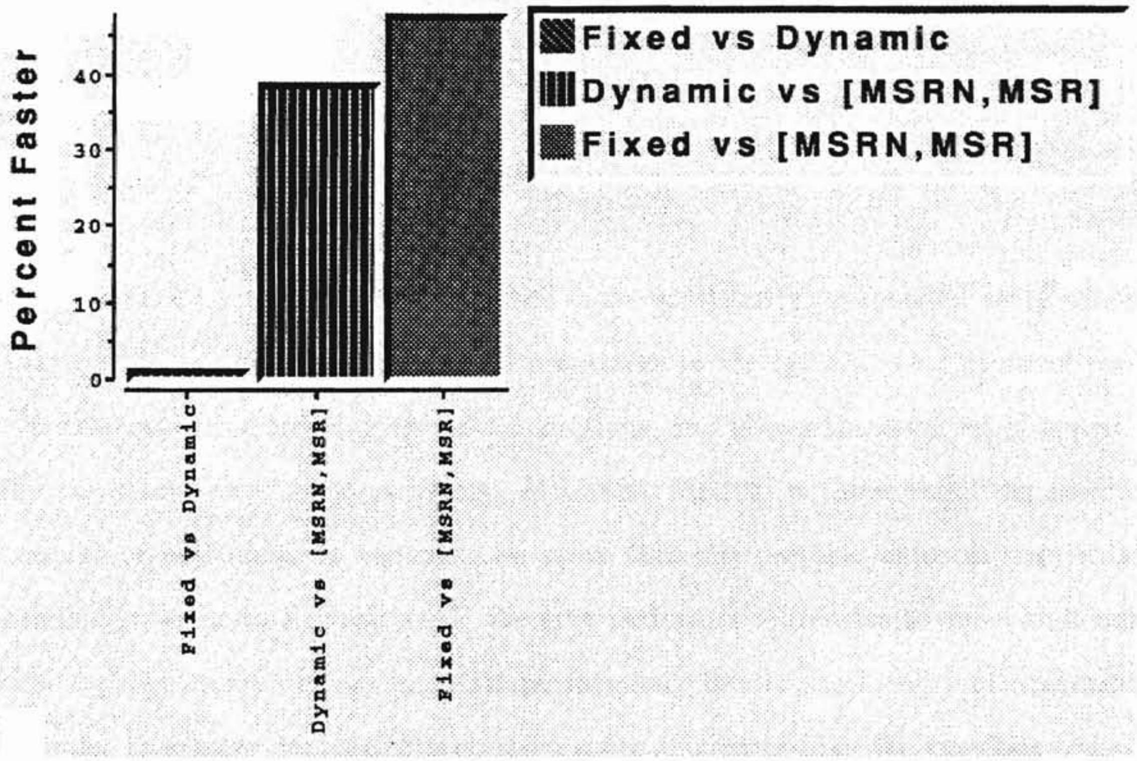

**Figure 4.17 Comparison of DSM Algorithms .8 read, 0.0 owner selection**

CHAPTER 5

# CONCLUSION

## 5.1 Summary

In this thesis, we presented a decentralized cache-consistency protocol for DSM which manages objects distributed among all processors in the system. Our protocol preservers the real-time ordering on write operations, and allows the same set of sequentially consistent executions as [Bro90, MSRN93, MSZ93] without requiring atomic broadcast or multicast. In contrast, we prove that our protocol enforces sequential consistency (Refer to Appendix A). We give performance metrics to show that our protocol provides an increase in access performance due to the locality of reference; and scales to a large number of processors more efficiently than the previous protocols, by avoiding the bottleneck of a single processor centralized memory. Although our protocol requires additional state information, the tradeoff of memory cost to communication cost provides reduction in overall communication performance.

## 5.2 Future Work

In this thesis, we are concerned with the performance of our protocol. Future work in protocol performance would be to reduce memory cost using a method simalar to [MSZ93]. Other future work would be to look at fault-tolerance issues that could effect the performance of the protocol such as transient failures. More work needs to be done simulating different distributions to distribute, access, and migrate objects.

# BIBLIOGRAPHY

[ABM89] Y. Afek, G. Brown, and M. Merritt. A lazy cache algorithm. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 209–222. ACM, 1989.

[AHJ91] M. Ahamad, R. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 274–281. IEEE, 1991.

[Bro90] G. Brown. Asynchronous multicaches. *Distributed Computing*, 4:31–36, 1990.

[BT91] H. E. Bal and A. S. Tanenbaum. Distributed programming with shared data. *Computer Languages*, 16:129–146, 1991.

[FL92] M. Feeley and H. Levy. Distributed shared memory with versioned objects. Technical Report TR-92-03-01, Department of Computer and Engineering , University of Washington, 1992.

[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 *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 15–26. Computer Architecture News, 1990.

[Hay88] J. P. Hayes. *Computer Architecture and Organization.* McGraw-Hill, Inc, 1988.

[HW90] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

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

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

[MSRN93] M. Mizuno, G. Singh, M. Raynal, and M. Neilsen. Communication efficient distributed shared memories. Technical Report TR-CS-93-3, Department of Computer and Information Sciences, Kansas State University, 1993.

[MSZ93] M. Mizuno, G. Singh, and James Z. Zhou. A sequentially consistent distributed shared memory. Technical Report TR-CS-93-4, Department of Computer and Information Sciences, Kansas State University, 1993.

[Sta84] J. A. Stankovic. A perspective on distributed computer systems. *IEEE Transactions on Computers*, 33:28–41, 1984.

[SZ90] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, 23:54–64, 1990.

[Tan92] A. Tanenbaum. *Modern Operating Systems.* Englewood Cliffs, N. J. : Prentice Hall, 1992.

APPENDIX A

# RELATED PROOFS

## A.1 Proof of Program Correctness

In this section we prove that our protocol preserves the real-time ordering on write operations and allows the same set of sequentially consistent executions as [Bro90, MSRN93, MSZ93]. The following proof is based on [MSRN93].

**Theorem:** The implementation is consistent; that is, it satisfies Definition 2.

**Assumptions:**

(a) No two processors initially own an object simultaneously.

(b) All operations on an object local or remote are executed atomically.

(c) Assume , for all processors, that all objects $o$ owned by a processor $i$ make up the distributed shared memory. If we let $DSM = h_1 \cup h_2 \cup ... \cup h_N$; where $N$ is the number of processors. Since $DSM$ is the global shared memory, all writes are performed only on objects maintained in $DSM$. More formally, let $P = (p_1, p_2, ..., p_N)$ be a set of processors, $O = (o_1, o_2, ..., o_M)$ be a set of objects maintained in the system, $H = (h_1, h_2, ..., h_N)$ be a set of objects owned by each processor such that $\forall [p_i] \in P, \exists [h_i \cap h_j = \emptyset]$. Since all updates to an object $o \in h_i$ are performed only by processor $p_i$, this maintains the strict ordering among writes for each $h_i \in DSM$.

**Proof:** Let $\hat{H}_o$ be an execution history of the protocol for object $o$. In order to show the implementation is consistent, by Definition 2, we have to show that:

(i) We can construct a sequential history $\hat{W}_o = (W_o, \rightarrow_{W_o})$, where $W_o$ is the set of all write operations in $\hat{H}_o$, (This preserves the real-time ordering on writes for each object separately)

(ii) For each processor $j$ we can construct a legal sequential history $\hat{W_oR_j} = (W_oR_j, \rightarrow_{W_oR_j})$, where $W_oR_j = W_o \cup R_j$ and $R_j$ is the set of read operations performed by processor $j$ on object $o$ in $\hat{H}_o$, such that $W_oR_j$ respects $W_o$ and $\hat{H}_o \mid i = \hat{W_oR_j} \mid i$ iff process $i$ owns $o$.

1. Now let $\hat{W}_o = (W_o, \rightarrow_{W_o})$ be a history such that if $o$ and $o'$ are operations in $W_o$ then $o \rightarrow_{W_o} o'$, if $o$ is processed before $o'$ by the owner of the object denoted $OWNER_o$. Because the $OWNER_o$ processes the write operations sequentially, $\hat{W}_o = (W_o, \rightarrow_{W_o})$ is a sequential history. We will show (ii) by constructing a legal sequential history $\hat{W_oR_j} = (W_oR_j, \rightarrow_{W_oR_j})$ as follows:

   (a) For any two operations $o$ and $o'$ in $W_oR_j$ which access $OWNER_o$, $o \rightarrow_{W_oR_j} o'$ if $o$ is processed before $o'$.

   (b) For any two operations $o_j^1$ and $o_j^2$ performed by processor $j$, $o_j^1 \rightarrow_{W_oR_j} o_j^2$, if $o_j^1$ is processed before $o_j^2$.

   (c) Let $r_{j_1}, r_{j_2}, \cdots, r_{j_N}$ be a sequence of consecutive local read operations by process $j$ ( thus, $r_{j_1} \rightarrow_{W_oR_j} r_{j_2} \rightarrow_{W_oR_j} \cdots \rightarrow_{W_oR_j} r_{j_N}$ due to the ordering enforced by (b)). Let $o_z$ be an operation by any processor $z$ which accesses $OWNER_o$ and immediately follows $o_j$ at $OWNER_o$ (thus, $o_j \rightarrow_{W_oR_j} o_z$ due to the ordering enforced by (a)). Then, $r_{j_N} \rightarrow_{W_oR_j} o_j$.

2. From (a) and the fact that all operations in $W_o$ access the $OWNER_o$, we have that $\hat{W_oR_j} = (W_oR_j, \rightarrow_{W_oR_j})$ respects $W_o$. From (b), $\hat{H}_o \mid j = \hat{W_oR_j} \mid j$. Finally, we will show that $\hat{W_oR_j}$ is legal.

   **Proof:** <u>Assume</u> that $W_oR_j$ is not legal for any processor $j$. Then there must exist a read operation $r(x)v$ such that $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ and there does not exist $w(x)s$ such that $w(x)u \rightarrow_{W_{OWNER_x}R_j} w(x)s \rightarrow_{W_{OWNER_x}R_j} r(x)v$. There are three cases to consider:

**Case 1:** Operation $r(x)$ performed by processor $i$, accesses $OWNER_x = i$. Then the last value written by any processor $j$ is the most recent write. Therefore, history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

**Case 2:** Operation $r(x)$ accesses $OWNER_x$, and processor $i$ is not the owner. Clearly, from the protocol, $w(x)u$ writes $u$ to $C_{OWNER_x}[x]$, and r(x) is performed after $w(x)u$ to $OWNER_x$. Thus, $r(x)$ does not return $v$, and the history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

**Case 3:** Operation $r(x)$ is a local read and processor $i$ is not the owner of $x$. Let $o_j$ be the last operation before $r(x)$ by processor $j$ which accesses some shared object . There are three cases to consider:

(a) $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} o_j \rightarrow_{W_{OWNER_x}R_j} r(x)v$: In this case, assume the owner of the object is processor $k$. There are two cases to consider:

(1.) $w(x)u$ is issued by processor $j$: Then $w(x)u$ sets $C_k[x] = u$, $Causal_k[x]$ is incremented, and $C_j[x] = u$, $Causal_j[x] = Causal_k[x]$, and $valid_j[x] = 1$. Since there does not exist $w(x)s$ ordered by $\rightarrow_{W_kR_j}$ in between $w(x)u$ and $r(x)$, the values of $valid_j[x]$ and $C_j[x]$ stay unchanged at least until $r(x)$ is performed. Thus, $r(x)$ locally reads value $u$ from $C_j[x]$, and history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

(2.) $w(x)u$ is not issued by processor $j$: Execution of $w(x)u$ sets $C_k[x] = u$ and $Causal_k[x]$ is incremented. After $o_j$ accesses the $OWNER_x = k$, $valid_j[x] = 1$ at processor $j$. Since $r(x)$ is a local read, $valid_j[x]$ must be 1 when $r(x)$ is performed by processor $j$. This means $valid_j[x]$ has been changed to 1 before $o_i$ is completed. From the protocol, $valid_j[x]$ can be changed to 1 only if a read or write operation on $x$ by processor $j$ is performed at $OWNER_x = k$. By the assumption, there does not exist $w(x)s$ ordered in between $w(x)u$ and $r(x)$ by $\rightarrow_{W_{OWNER_x}R_j}$. Therefore, there must be a read operation by processor $j$ which reads $C_k[x]$ at $OWNER_x = k$ between $w(x)u$ and $o_j$, including $o_i$. This read operation also sets $valid_j[x] = 1$ and $C_j[x] = u$. Thus, $r(x)$ returns $u$, and history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

(b) $o_j$ is $w(x)u$: Then, the operation $w(x)u$ sets $valid_j[x] = 1$ and $C_j[x] = u$. Since there is no operation by processor $j$ which accesses an object at $OWNER_x = k$ between $w(x)u$ and $r(x)$,$r(x)$ returns $u$. Thus, history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

(c) $o_j \rightarrow_{W_{OWNER_x}R_j} w(x)u$: In this case, rule (1c) above orders $r(x)$ in between $o_j$ and $w(x)u$. Hence, a history $w(x)v \rightarrow_{W_{OWNER_x}R_j} w(x)u \rightarrow_{W_{OWNER_x}R_j} r(x)v$ never occurs.

## A.2 Proof of Bound on Forward Messages

The two critical questions about this algorithm are whether forwarding request eventually arrive at the true owner and how many forwarding request are needed in the worst case. In order to prove these questions, consider all *Probowners* of an object $o$ as a directed graph $G_o = (V, E_o)$ where $V$ is the set of processors numbered 1,...,$N$, and $\mid E_o \mid= N$ and an edge $(i, j) \in E_o$, iff the *Probowner* for an object $o$ on processor $i$ is $j$. The following proof follows [LH89].

**Theorem:** A request for an object assumed to be owned by the hint in *Probowner*, will reach the true owner in at most $N - 1$ forwarding request messages.

**Lemma:** Because read/write request are executed atomically and migration is done only during a remote operation, this ensures that migration occurs sequentially. Assuming migration takes place in the worst case (i.e. on every access), every *Probowner* graph $G_o = (V, E_o)$ has the following properties:

1. there is exactly one node $i$ such that $(i, i) \in E_o$;
2. Graph $G'_o = (V, E_o - (i, i))$ is *acyclic*; and
3. for any node x, there is exactly one path from $x$ to $i$.

**Proof:** By induction on the number of migrations of object $o$, all *Probowners* of the processors in $V$ are initialized to a default processor, and all three properties are satisfied. After one migration of object $o$ , say from $i$ to $j$, the node $(i, i)$ (i.e. the current owner of object $o$) is deleted from $E_o$, and the node $(i, j)$ is inserted into the *Probowner* graph $G_o$. This ensures there is only one path from $i$ to $j$; satisfying property 3. As node $(i, j)$ was the root, the subgraphs are still pointing to $i$ and remain unchanged and are acyclic; satisfying property

2. The node $(j, i)$ is deleted from $E_o$, because $j$ now becomes the owner (root of $i$); therefore a new node is inserted into the graph $G_o$ denoted $E_o = (j, j)$. This satisfies property 1. After $k$ migrations of an object $o$, the *Probowner* graph $G_o$ satisfies the three properties.

**Proof:** By Lemma 1, there is only one path to the true owner and there is no cycle in the *Probowner* graph. So, the worst case occurs when the *Probowner* graph is a linear chain

$$E_o = \{(v_1, v_2), (v_2, v_3), ..., (v_{N-1}, v_N), (v_N, v_N)\}$$

in which case the number of forwarding request is $N - 1$ when processor $v_1$ request an operation from processor $v_N$.

APPENDIX B

## COST/PERFORMANCE

**Table B.1.**

**Memory Cost Performance**

Assumptions: **K = # of object, N = # of processors**

| **Protocol** | **Process** | **Memory** | **Size** |
|---|---|---|---|
| [ABM89] | Processor | IN Queue | Unbounded |
| | | OUT Queue | Unbounded |
| | | Cache[ ] | $K$ |
| | Shared Memory | Cache[ ] | $K$ |
| [Bro90] | Processor | Queue | Unbounded |
| | | Cache[ ] | $K$ |
| | Shared Memory | Cache[ ] | $K$ |
| [MSRN93] | Processor | Valid[ ] | $K$ |
| | | Cache[ ] | $K$ |
| | SMem | M[ ] | $K$ |
| | | Cache_Ver[ ][ ] | $N \times K$ |
| | | Causal[ ] | $K$ |
| [MSZ93] | Processor | Valid[ ] | $K$ |
| | | Cache[ ] | $K$ |
| | SMem | hlw[ ][ ] | $N \times K$ binary vector |
| | | Causal[ ] | $K$ |
| **Fixed Manager Decentralized** | Processor | C[ ] | $K$ |
| | | valid[ ] | $K$ |
| | | Causal[ ] | $K$ |
| **Distributed Manager Decentralized** | Processor | C[ ] | $K$ |
| | | Probowner[ ] | $K$ |
| | | valid[ ] | $K$ |
| | | Causal[ ] | $K$ |

**Table B.2.**

**Communication Cost Performance**

| **Message** | [Bro90, ABM89] | [MSRN93] [MSZ93] | **Decentralized** |
|---|---|---|---|
| Reads | 0 if valid<br>1 if ¬ valid | 1 if ¬ valid<br>0 if valid | 0 if owner<br>1 if ¬ owner |
| Writes | $N$ | 1 | 0 if owner<br>1 if ¬ owner |
| Forwards | N/A | N/A | Best case: 1<br>Worst case: $N - 1$ |

APPENDIX C

# SIMULATION PARAMETERS

**Table C.1.**

**Simulation Parameters - Browns Protocol**

Assumptions: No Special Hardware, N = # of processors

| **Process** | **Operation** | **Clock Cycly Time** |
|---|---|---|
| Process Manager | Read/Write Local | 1 |
| | Remote Read | 10 |
| | Write SMem + Invalidations | 10 + (10 * (N-1)) |
| | Equeue Invalidations | 1 |
| | Communication Latency | 100 |
| | Duration between Operations | 5 |
| Shared Memory | Process Request | 2 |

**Table C.2.**

**Simulation Parameters - Mizuno, Raynal, Singh, and Neilsen Protocol**

| **Process** | **Operation** | **Clock Cycly Time** |
|---|---|---|
| Process Manager | Read/Write Local | 1 |
| | Remote Read/Write | 10 |
| | Communication Latency | 100 |
| | Duration between Operations | 5 |
| SMem | Process Request | 2 |

Table C.3.

**Simulation Parameters - Decentralized Protocol**

| **Thread** | **Operation** | **Clock Cycly Time** |
|---:|---:|---:|
| Process Manager | Local Read/Write | 1 |
| | Remote Read/Write | 10 |
| | Communication Latency | 100 |
| | Duration between Operations | 5 |
| Object Manager | Process Request | 2 |
| | Forward Request | 2 |

2

# VITA

Legand L. Burge III

Candidate for the Degree of

Master of Science

Thesis: A DECENTRALIZED ALGORITHM FOR COMMUNICATION EFFICIENT DISTRIBUTED SHARED MEMORY

Major Field: Computer Science

Biographical Data:

Personal Data: Born in Stillwater, Oklahoma on February 5, 1972, the son of Dr. L. L. Burge Jr. and Gwenetta V. Burge

Education: Graduated from John Marshall High School, Oklahoma City, Oklahoma, 1989; received Bachelor of Science in Computer Science from Langston University, Langston, Oklahoma in 1992. Completed the requirements for the Master of Science degree in Computer Science at Oklahoma State University in July 1995.

Experience: Computer Analyst, National Security Agency, Ft. George G. Meade, Maryland, 1991 to present.