73

MULTICASTING SUPPORT FOR DISTRIBUTED SHARED MEMORY SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

AYDIN OKUTANOĞLU

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ
119274

IN PARTIAL FULLFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

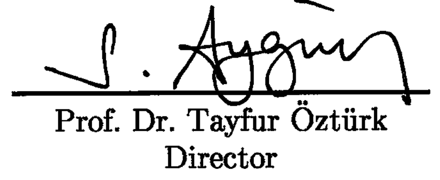
119274

IN

THE DEPARTMENT OF COMPUTER ENGINEERING

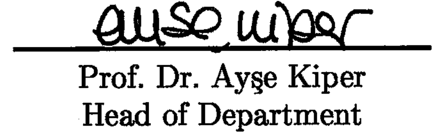
SEPTEMBER 2002

Approval of the Graduate School of Natural and Applied Sciences.



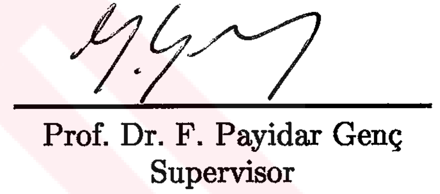
Prof. Dr. Tayfur Öztürk
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.



Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.



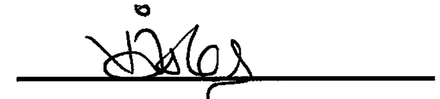
Prof. Dr. F. Payidar Genç
Supervisor

Examining Committee Members

Prof. Dr. F. Payidar Genç

Prof. Dr. Müslim Bozyiğit

Assoc. Prof. Dr. Volkan Atalay

Assoc. Prof. Dr. Veysi İşler 

Dr. Attila Özgit

ABSTRACT

MULTICASTING SUPPORT FOR

DISTRIBUTED SHARED MEMORY SYSTEMS

Okutanoğlu, Aydın

M.Sc., Department of Computer Engineering

Supervisor: Prof. Dr. F. Payidar Genç

September 2002, 64 pages

Distributed Shared Memory (DSM) Systems offers an easy-to-use and scalable solution for a large class of scientific and numeric applications. Unfortunately, the performance of current DSM systems are not as good as message passing systems because of extra processing and communication for consistency. In this study it is tried to increase the efficieny of existing DSM systems using new distributed algorithms. Multicasting is used as the communication method of these distributed algorithms.

Keywords: distributed systems, shared memory, computer networks

ÖZ

DAĞITIK PAYLAŞIMLI HAFIZA SİSTEMLERİ İÇİN

ÇOKLU YAYIM DESTEĞİ

Okutanoğlu, Aydın

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. F. Payidar Genç

Eylül 2002, 64 sayfa

Dağıtık Paylaşımlı Hafiza sistemleri (DSM) bir çok çeşit sayısal ve bilimsel uygulama için kolay kullanımlı, ölçeklenebilir çözümler sunmaktadır. Fakat, şuan ki DSM sistemleri mesajlaşma sistemleri kadar performans sağlayamamaktadır. Bunun sebebi asıl olarak dağıtık durumdaki hafiza sayfalarının tutarlılığını sağlamak amacıyla yapılan ekstra işlemler ve mesajlaşmadır. Bu çalışmada, var olan paylaşımlı hafiza sistemlerinin performansı yeni dağıtık algoritmalar kullanılarak yükseltilmeye çalışılmıştır.

Anahtar Kelimeler: dağıtık sistemler, paylaşımlı hafiza, bilgisayar ağları

ACKNOWLEDGMENTS

I would like to thank my supervisor Prof. Dr. F. Payidar Genç for his guidance during the development of this thesis.

TABLE OF CONTENTS

ABSTRACT iii

ÖZ iv

ACKNOWLEDGMENTS v

TABLE OF CONTENTS vi

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

1 INTRODUCTION 1

2 BACKGROUND INFORMATION 3

2.1 Page-based versus object-based 3

2.2 Update-based versus invalidation-based 5

2.3 Sequential Consistency 6

2.4 Release Consistency 8

2.5 Lazy Release Consistency 10

2.6 Entry Consistency 13

2.7 Scope Consistency 14

2.8 Diff Mechanism 16

2.9 Multiple Writers Protocol 18

2.10 Distributed Synchronization 20

2.10.1 Locks 21

2.10.2 Barriers 21

2.11 Multicasting 22

2.11.1 Multicast Addresses 23

2.11.2 Sending Multicast Messages 24
2.11.2.1 TTL 25
2.11.3 Receiving Multicast Messages 26
2.11.3.1 Joining a Multicast Group 26
2.11.3.2 Leaving a Multicast Group 26
3 MULTICASTING SUPPORT FOR DSM SYSTEM 27
3.1 Run-time System 28
3.1.1 Shared Object Creation 29
3.1.2 Distributed Lock Creation 33
3.2 Distributed Synchronization 34
3.2.1 Lock Mechanism 35
3.2.2 Barrier Mechanism 42
3.2.3 Diff Mechanism 44
3.3 Consistency Scheme 45
4 EXAMPLE APPLICATIONS 52
4.0.1 Travelling Salesman Problem (TSP) 52
4.0.2 Parallel Quicksort 56
5 CONCLUSION AND FUTURE WORK 59
REFERENCES 61

LIST OF TABLES

4.1 Running times for TSP problem 55

LIST OF FIGURES

2.1 Sequential Consistency 8
2.2 Release Consistency 9
2.3 Lazy Release Consistency 11
2.4 Creating a Twin 17
2.5 Creating Diff 17
2.6 Multiple Writers 18
2.7 False sharing with arrays 19
2.8 Multiple concurrent writers 20
2.9 IP Address Classes 23

3.1 Run-time System 29
3.2 Shared Object Creation Algorithm 30
3.3 Shared Pages in a Host 32
3.4 A DSM Page Structure 33
3.5 Lock Structure 34
3.6 Lock algorithms 36
3.7 Lock Mechanism Example - 1 37
3.8 Lock Mechanism Example - 2 38
3.9 Lock Mechanism Example - 3 39
3.10 Client and DSM Server Interaction in Lock mechanism 41
3.11 Barrier Algorithm 43
3.12 Diff Structure 45
3.13 Consistency Algorithm Example 46
3.14 Consistency Algorithm Example 48
3.15 Consistency Algorithm Example 49
3.16 Consistency Algorithm Example 50

4.1 TSP Search Spaces 53
4.2 Running Times for Parallel Quicksort 57

CHAPTER 1

INTRODUCTION

This study has the purpose of developing effective and efficient group communication algorithms using multicasting protocol to increase the efficiency of the distributed shared memory systems and distributed synchronization systems.

There are two fundamental models for the parallel programming. The shared memory model and the message passing model. In shared memory model an update to the memory becomes visible to all the nodes in the system. In contrast, in the message passing model the only way for nodes to communicate is through explicit message passing over the interconnection network [16].

A distributed shared memory (DSM) system provides a shared memory programming on distributed memory machines, which is an easy way of programming. Hardware DSM supports this abstraction at the architecture level, software DSM systems support this abstraction within a runtime system. Software DSM systems consists of the same hardware as that found in the distributed

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

memory machines, with the addition of a software layer that provides the abstraction on a single shared memory.

Although many DSM systems have been proposed and implemented, they do not have good performance as message passing systems for much of the applications. The primary reason for this is overhead of communication for keeping the shared memory consistent. Ideally, the amount of communication for an application executing on a DSM system should be comparable to the amount of communication for the same application executing directly on the underlying message passing system.

The reminder of this thesis is organized as follows: Chapter 2 introduces the basic concepts in distributed shared memory systems and multicasting technology. Chapter 3 gives the detailed explanation of the study done in the thesis and chapter 4 gives the experimental results obtained using the implemented distributed shared memory system.

CHAPTER 2

BACKGROUND INFORMATION

2.1 Page-based versus object-based

In a distributed shared memory system, shared segments of memory must be kept consistent according to some ordering definition. While keeping the memory segments consistent, DSM system should consider the shared memory part by part. This means that, the shared memory is considered in the way that it consisting of some memory parts. If DSM system does not handle the shared memory in that way, when a process want to access to some part of the shared memory it must get all shared memory.

The level of granularity and how this will be achieved is an important question in DSM systems. Two schemes, namely page-based and object-based, are used.

In page-based consistency [3, 2], DSM system tries to keep consistent the pages of shared memory as a whole. It does not have any information about the

shared variables on the shared pages. DSM system just tries to keep consistent only shared pages as a whole, not the shared variables. It does not matter whether the shared objects on the page are relevant or not. Here relevance of the objects means that, objects are frequently accessed in the same critical sections, i.e. same acquire and release block, which is the block of operations between any acquire and release operation. This irrelevance of information on the same page can lead to some drawbacks, for example false sharing (False sharing is explained in the future sections).

With page-based approach [11, 7], it is easy to control memory access rights which is necessary in consistency algorithm implementation. For example, if a shared memory location should not be accessed in a process, the page it is in can be easily made unreadable. Also with page-based scheme more natural virtual memory abstraction is achieved because real memory access is also controlled with page structures.

The other approach is object-based consistency. In that, objects are kept consistent independent of each other. This consistency requires that every shared object in the parallel program should be associated with a synchronization variable. This is necessary because at the release point for example, DSM system should reflect the changes only shared ojects which are related with this lock. There is no much work in this algorithm, because the association of the lock variables and object gives enough information for handling consistency operations.

Since the association information can not be known by the run-time system,

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

user must explicitly specify it in the shared memory program code. Another drawback is in controlling the accesses to the objects. Current architectures support only page-level control, for this reason each object must kept in a separate page and this will waste memory.

2.2 Update-based versus invalidation-based

In a DSM consistency system, when a process makes a change to a shared memory portion, this change must be reflected to all other processes. DSM system can choose either updating or invalidating the remote copies of page at that time. Updating process is simply sending modified portions of memory to all other processes that have the cached copy, so that the processes update their shared objects using this information. In invalidation process, processor sends a message to the other processes containing an invalidation indicator. This message, changes state of the shared memory object to an invalid state. When a process tries to access an invalid memory, it must get the changes from the original process.

In the update based method, after a modification changed portions of shared memory is sent to all relevant processes immediately. But in invalidation based method after a modification, only an invalidation message is send, not the update information. After sending the invalidation message, that page is became invalid. This means that when a process tries to access this page, a page fault is rised by the system. In this situation, process should get the update information from the other process which makes the modification. Using this update information

it should refresh its invalid page, and after that, the page is became valid, and process can access to that page.

Update based method is write-blocked, that is, after a write operation, it blocks the program to send the update information to other processes. This method saves time for reading, because no extra message is needed on a read operation. On the other hand, invalidation based method is read blocked, i.e. before a read operation to be performed changes to that memory portion must be got from other process. However, in this method, extra time to send update information on each write (or write block) is reduced. The detailed information for update-based and invalidation based aproaches can be found in references [3, 2].

2.3 Sequential Consistency

The first and naive consistency model for distributed shared memory is Sequential Consistency[1]. Sequential Consistency (SC) have the same consistency properties with a time shared uniprocessor machine, i.e. after every read or write operations, global state of memory must be made consistent. This requires that after any access to the memory all changes must be reflected to all the copies of that memory portion. Of course this update operation causes too many unnecessary communications and computations. A number of relaxed consistency models were developed to overcome the inefficiency of SC. Some of them are, weak consistency[4], processor consistency[6], release consistency[2], lazy release consistency[3] and entry consistency[7]. The other studies on this

topic can be found in references [21, 11, 17, 19, 20, 23].

Some definitions[4] related with the consistency models are the followings:

- A **read** by processor p_i is **performed with respect to processor** p_j at a point in time when the issuing of a **write** to the same address by p_j can not affect the value returned to p_i.

- A **write** by processor p_i is **performed with respect to processor** p_j at a point in time when a **read** from the same address by p_j returns the value defined by the **write**.

- An access is **performed** when it is performed with respect to all processors.

- A read access is **globally performed** when it is performed and the write that is the source of the returned value has also performed.

A DSM system that implements sequential consistency reflects the following constraint[4]

- Before a read or write is allowed to perform with respect to any remote processor, all previous reads must be globally performed and all previous writes must be performed.

In the Figure 2.1, processor P_1 acquires the lock l then writes x and y, and releases the lock. After that, processor P_2 acquires the lock, reads x and y, writes z and releases the lock.

As can be seen from the Figure 2.1, in sequential consistency all access to the shared variables immediately reflected to the other processors. In above

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

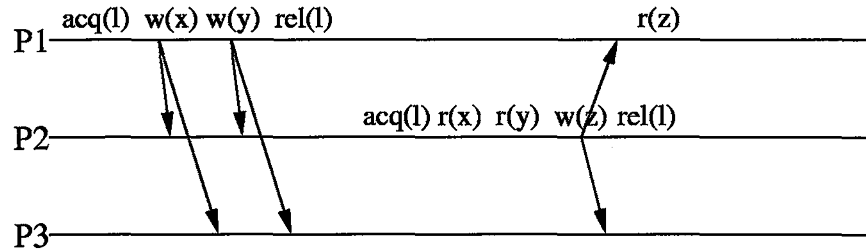


Figure 2.1: Sequential Consistency

situation P_3 does not access the shared variables but its cached copies of shared memory are also updated. There are too many extra messages to satisfy the constraint of making writes performed.

2.4 Release Consistency

Release Consistency[2, 18], is a relaxed consistency model, which utilize the fact that programmers of shared memory generally use synchronization variables to properly access the shared portions of memory. When a processor acquires a lock, it can safely access shared variables, while other processors that want to access shared variables must wait for lock to become free. This fact brings the relaxation that it is necessary to keep consistent shared memory only at synchronization points. Making use this reality, release consistency brings three constraints on the memory model [2]:

1. Before any read and write allowed to perform with respect to any other processor, all previous acquire accesses must be performed,

2. Before a release access is allowed to performed with respect to any other processor, all previous read and write accesses must be performed,

3. Synchronization accesses must be sequentially consistent with respect to another.

(1) implies that when a processor wants to access a shared variable it must acquire a lock, so that all consistency related operations (i.e. updates or invalidates of all related caches) are done. In (2), when a processor finishes its job with a shared variable and wants to release a previously acquired lock, it must update all cached copies of that shared portion in all other hosts, i.e make the shared memory consistent. (3) emphases the requirement of linearization of accesses to the synchronization variables. Because accesses to them are random, they must be keep sequentially consistent.

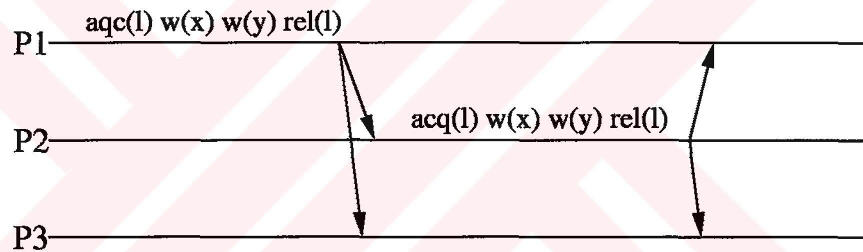


Figure 2.2: Release Consistency

Advantages of release consistency over sequential consistency is obvious, in release consistency only at release time the other processors are updated and the extra time of an update operation on each write on sequential consistency is reduced.

In release consistency (also referred as eager release consistency) after a release operation, cached copies of all other processors updated. In the situation on the Figure 2.2 even processor P_3 which does not access the page currently, still gets the update messages.

An optimization for eager release consistency is the update timeout mechanism. In this method, an update list for each shared page are maintained and if a processor does not access a shared page for a timeout limit, it is deleted from update list of that page, By doing so, much of the unnecessary update messages are avoided.

2.5 Lazy Release Consistency

Although release consistency relaxes the restrictions of sequential consistency, it still requires accesses to be performed globally before a local release can complete. In lazy release consistency[3, 12], synchronization transfers takes place without performing any ordinary shared access globally, instead shared accesses only have to be performed at other processes as they synchronize with the performing process.

Conditions for lazy release consistency (LRC) are:

- Before a read and write access is allowed to perform with respect to another process, all previous acquire accesses must be performed with respect to that other process,
- Before a release access is allowed to perform with respect to any other process, all previous read and write accesses must be performed with respect to that other process,
- synchronization accesses are sequentially consistent with respect to one another.

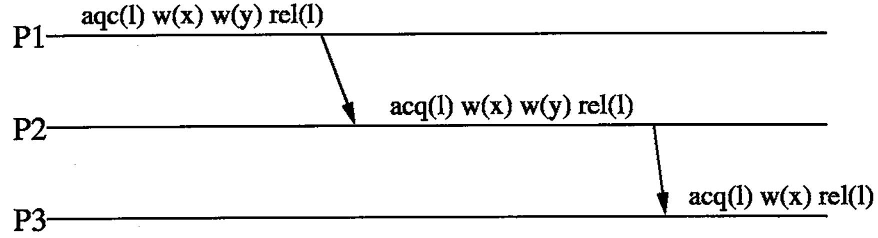


Figure 2.3: Lazy Release Consistency

In the Figure 2.3, processors P_1, P_2 and P_3 accordingly acquires and then releases the lock *l*. The update information for shared variables reflected only from last releaser to current acquirer. Using this method considerable amount of network traffic are reduced.

In order to support the memory model conditions given above, a *happened-before-1*[5] partial ordering over all shared accesses are used,

Definition: Shared memory accesses are partially ordered using happened-before-1, denoted by $\xrightarrow{hb1}$, defined as follows:

- if a_1 and a_2 are accesses on the same process, and a_1 occurs before a_2 in program order, then $a_1 \xrightarrow{hb1} a_2$.
- if a_1 is a release on process P_1, and a_2 is an acquire on the same memory location on process P_2, and a_2 returns the value written by a_1, then $a_1 \xrightarrow{hb1} a_2$.
- if $a_1 \xrightarrow{hb1} a_2$ and $a_2 \xrightarrow{hb1} a_3$ then $a_1 \xrightarrow{hb1} a_3$.

By happened-before-1 relation, shared accesses are ordered with both program order and synchronization order. (i.e. an acquire is ordered after the last previous release of the same lock).

Lazy release consistency requires that before a process may continue past an acquire, all shared accesses that precede the acquire according to $\xrightarrow{hb1}$must be performed at the acquiring process, where "performing" an access at process P means either updating or invalidating P's copy of indicated data item.

Both eager and lazy release consistency protocols guarantee to support the same programming model as sequential consistency protocol, if programs are *data-race-free*[5]. Definition of data-race-free is:

Definition: A *data race* in an execution is a pair of conflicting operations, at least one of which is write, that is not ordered by happened-before-1 relation for the execution. An execution is *data-race-free* if and only if it does not have any data race. A program is *data-race-free* if and only if all its sequentially consistent executions are data-race-free

If a program properly labelled, i.e. appropriate synchronization accesses placed in the program, it will become data-race-free. This condition on shared memory program is not arduous, because the most of the parallel programs are data-race-free already.

Maintaining and using the ordering of happened-before-1 satisfies the conditions of lazy release consistency. But applying this ordering to all shared accesses brings heavy processor and network load. Instead, in LRC programs, intervals are ordered according to $\xrightarrow{hb1}$. A new interval begins each time process executes a synchronization access. Before an interval begins all previous updates (or invalidates) of previous intervals must be done on this process according to

happened-before-1.

2.6 Entry Consistency

Entry Consistency[7] (EC) is another relaxed consistency system, that further relaxes the restrictions of release consistency. Entry consistency takes advantage of the relationship between specific synchronization variables and the shared data accessed within those critical sections. In an entry consistent system, a processor's view of shared memory becomes consistent only when it enters a critical section and only shared memory that is guaranteed to become consistent is that which can be accessed within the critical section.

Synchronization variables are said to be guard to a group of shared data (D_s) which can be accessed within the critical section. The condition for the entry consistency is,

- before an *acquire* access of s is allowed to perform with respect to processor p_i, all updates to D_s must be performed with respect to p_i.

Entry consistency allows to replicate the data for reading only by replicating synchronization variables as well as shared data. This mode is called *non-exclusive mode.* This situation gives rise to two further conditions for entry consistency,

- Before an exclusive mode access to a synchronization variable s by processor p_i is allowed to perform with respect to p_i, no other processor may hold s in non-exclusive mode.

- When an exclusive mode access to s has been performed, any other processor's next non-exclusive mode access to s may not be performed until it is performed with respect to the owner of s.

Entry consistency protocol reduces the required communication by getting the lock related information from the last releaser of the lock. However this method requires that synchronization variables are explicitly binded with a logical group of shared objects in the program, and this requirement is the major drawback of the entry consistency.

2.7 Scope Consistency

Scope Consistency[8] is a bridge between Release Consistency and Entry Consistency. Scope Consistency (ScC) offers most of the performance advantages of EC, without requiring explicit association of data to synchronization variables. It defines *consistency scopes*, which indicate the associations between data and synchronization variables dynamically.

A consistency scope is a restricted view of shared memory with respect to which memory accesses are performed. This means, modifications to data performed within a scope are only guaranteed to be visible within that scope. A consistency scope begins by acquiring a specific lock (open scope operation) and ends by releasing that lock (close scope operation). The interval during which a consistency scope is open is called a *session*. Modifications during a consistency scope session is guaranteed to be reflected the next opening process of that scope, not any other modifications are reflected.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

To define the consistency model an additional definition was made to distinguish a reference being *performed with respect to a consistency scope* from a reference being *performed with respect to a process*:

- A write that occurs in a consistency scope is *performed with respect to that scope* when the current session of that scope closes.

The consistency rules for Scope Consistency are:

1. Before a new session of a consistency scope is allowed to open at process P, any write previously performed with respect to that consistency scope must be performed with respect to P.

2. A memory access is allowed to perform with respect to a process P only after all consistency scope sessions previously entered by P (in program order) have been successfully opened.

ScC gets some properties from both Entry Consistency and Release Consistency. Like RC, it does not require explicit binding of data to synchronization. However, ScC assumes an implicit binding of data to consistency scopes. This leads a decrease in granularity of the shared parts and thus, a decrease in false sharing (see Section 2.9 for false sharing) in comparing with RC.

Like EC, ScC somehow binds the data to synchronization. However, ScC does this implicitly, with the use of synchronization scopes. In EC, binding is done by the programmer, and he can arrange the granularity of coherence, and can decrease the false sharing and increase the performance of shared memory.

The main penalty of Scope Consistency is determining the changed fields of shared memory in the run-time system for that consistency scope especially in multi-threaded shared memory system. It can be done by either suspending other threads while one thread enters a consistency scope or labelling the writes in the consistency session.

2.8 Diff Mechanism

While keeping the shared memory consistent the updates of processes must be reflected to other processes shared memory. Of course transferring all pages through network is not an efficient way of doing this. Instead, it is sufficient to transfer the modified portions of a page.

In diff mechanism, before updating a shared page, a copy, or twin page is created and changes are done on the original copy. When the update operations are finished, the DSM system compares the original page and its twin and creates a *diff*, representing the changes.

In relaxed memory consistency systems in which consistency is based on the synchronization access, the twin is created at the time of acquire, because this is the time when a program starts to change a shared page. But it is unknown that which shared pages will be modified, and generating the twins of all shared pages is not necessary unless all pages will be modified.

For this reason shared pages are originally in read-only mode. If a process tries to write a page, a write fault will happen, and this fault is handled by the DSM system. At the time of a write fault, DSM system creates the twin and

stores it in the system space, and changes the state of original page to read-write mode. This operation can be seen on Figure 2.4.

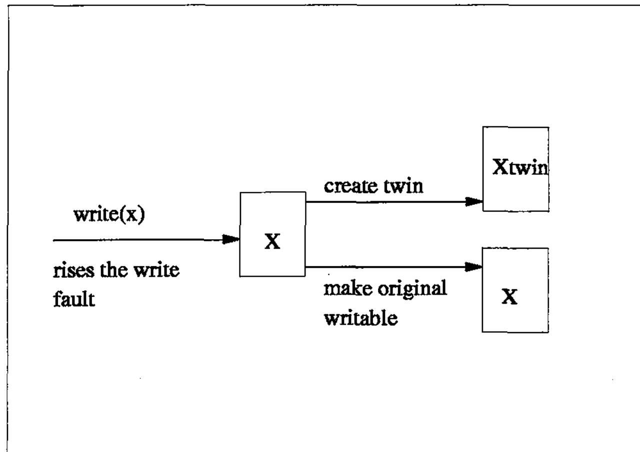


Figure 2.4: Creating a Twin

At the time of lock release, the DSM system compares the twin and the original copy, creates a *diff*, and makes original copy write-protected for the next acquire operation. Figure 2.5 shows this operation.

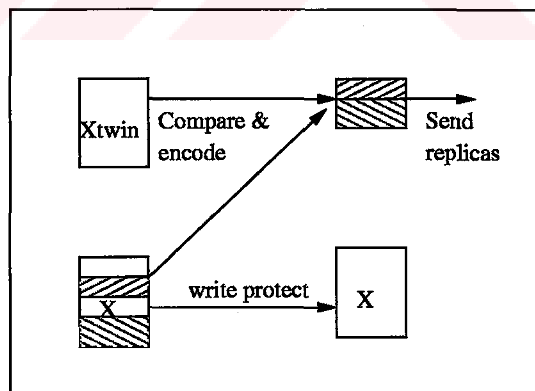


Figure 2.5: Creating Diff

When a process gets a diff from another process, it can easily update its local page using diff.

2.9 Multiple Writers Protocol

In page-based consistency protocols consistency units are pages. Most of the architectures use either 4KB or 8KB page sizes. This size is too big for consistency systems. Consider the situation in Figure 2.6 where the shared variables A and B are both placed on the same page.

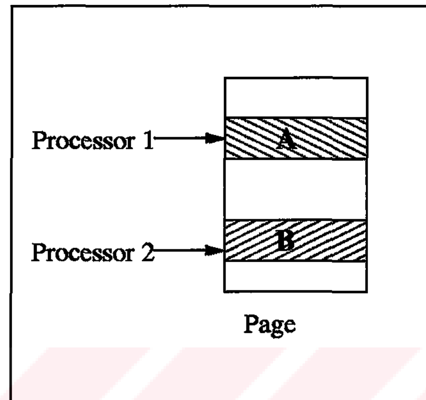


Figure 2.6: Multiple Writers

Processor 1 wants to access variable A and processor 2 wants to access variable B two or more times simultaneously. Let us assume that the processor 1 gains the access (by acquiring a lock), updates A, and finishes its job with A temporally and releases the lock. In the same time processor 2 tries to access B by acquiring another lock (This is possible because, A and B are logically unrelated to each other and programmer may choose to synchronize them using different lock variables). But it realizes that the page containing B (also A) is modified. So, it must get the update information from processor 1. However, the new contend of A is not required by processor 2 at that time because it deals only with B. It gets the update information of A, and begins its process and updates B. This time processor 1 faces the same problem when it wants to

access A again. As a result, this page ping-pongs between the processor 1 and 2. This phenomenon is called *false sharing*[22].

False sharing can occur in two ways. The first one occurs when two scalar variables are placed on the same page as in the Figure 2.6. This type of false sharing can be avoided by allocating a seperate page for for each shared variable, but this method wastes considerable amount of memory.

The second situation that false sharing occurs is when, a shared array is placed on a page and two or more processors access the distinct portions of the array as in the Figure 2.7. This type of false sharing can not be avoided.

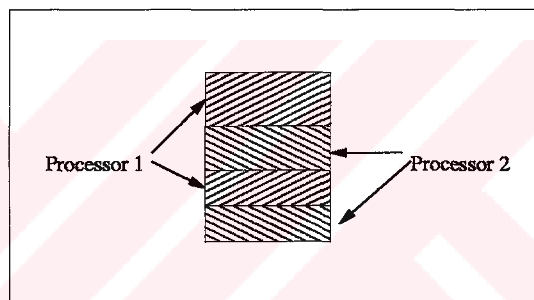


Figure 2.7: False sharing with arrays

False sharing is a particularly serious problem for DSM systems for two reasons: (i) the consistency units (pages) are large, so false sharing is very common. (ii) the latencies associated with detecting modifications and communicating are large, so unnecessary faults and messages are particularly expensive. So, it is necessary to achieve a fine-grained access granularity, rather than operating on a strict per page granularity.

A solution to this problem is based on the assumption that all programs are data-race-free. In multiple concurrent writers method, all processors are allowed

to write on the same page simultaneously. Since the programs are data-race-free, any of them modifies the same place on a page. At a synchronization point, each processor sends its modifications (diffs) to the requesting processor.

For example, in Figure 2.8 processors P_1, P_2 and P_3 share a page containing two scalar variables A and B. Processor P_1 and P_3 modifies the different parts of shared page, A and B accordingly. P_2 can construct a fresh copy of that page by getting and applying the diffs from P_1 and P_3.

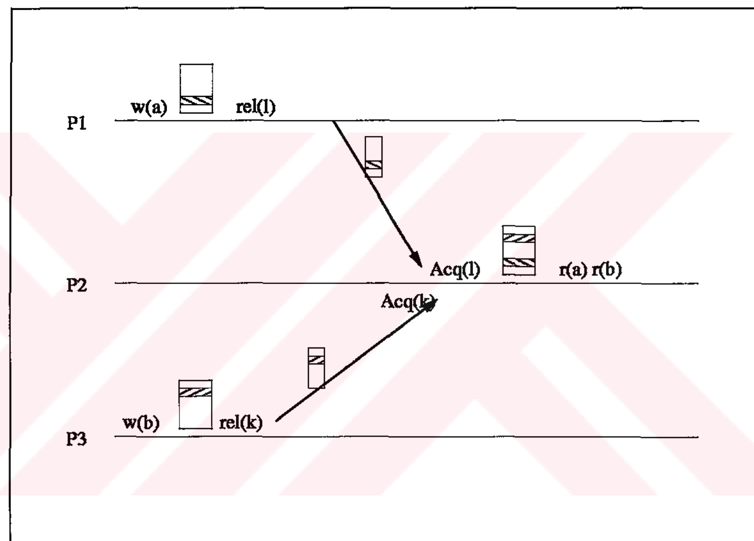


Figure 2.8: Multiple concurrent writers

2.10 Distributed Synchronization

There are several distributed synchronization management techniques. Some of them are explained in [10, 13, 14, 15].

2.10.1 Locks

There mainly two types of distributed lock management method. In the centralized one, there is a lock manager and each processor keeps two indicators, local and hold, indicating that lock is currently local and acquired respectively. If the lock is a local lock, acquiring the lock is just setting the flag hold. If it is not local, processor sends a message to lock manager, to request the lock. If the lock is currently free, the manager sends the grant message, else it forwards the message to the current owner of the lock, so that the current owner deals with the request. In this method, at most two message are enough to get the lock.

In distributed lock management [2], a distributed queue for each lock is maintained. When a processor wants to acquire a lock and it is not local, it forwards the request to the probable owner of the lock. If the probable owner is the actual owner, it deals with the request, else it forwards the message to its own prediction for probable owner. This method may require more than two message to get the lock, but it is a fully distributed algorithm for lock mechanism.

2.10.2 Barriers

Barriers are generally implemented with central barrier manager. Requesters send a message to the manager, when manager gets specified number of barrier message it answers back to barrier waiters to continue.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

2.11 Multicasting

Multicasting [9] is a one-to-many network transportation mechanism. Its main use is transporting the broadcast content to multiple clients. For example, for IP based radio transformation, it is not feasible to send content in using different TCP or UDP connection to all listeners because volume is high and this communication brings high processing and communication overhead. The solution that multicasting brings is sending only one message to a predefined address and all related clients gets the message using this address.

Multicasting is a receiver-based concept. Receivers join a particular multicast session group and traffic is delivered to all members of that group by the network infrastructure. The sender does not need to maintain a list of receivers. Only one copy of a multicast message will pass over any link in the network, and copies of the message will be made only where paths diverge at a router. Thus IP Multicast yields many performance improvements and conserves bandwidth.

IP Multicast is an extension to the standard IP network-level protocol. RFC 1112, Host Extensions for IP Multicasting [9], describes IP Multicasting as: "the transmission of an IP datagram to a 'host group', a set of zero or more hosts identified by a single IP destination address. A multicast datagram is delivered to all members of its destination host group with the same 'best-efforts' reliability as regular unicast IP datagrams. The membership of a host group is dynamic; that is, hosts may join and leave groups at any time. There is no restriction on the location or number of members in a host group. A host may be a member of more than one group at a time." In addition, at the application level, a single

group address may have multiple data streams on different port numbers, on different sockets, in one or more applications. Multiple applications may share a single group address on a host.

2.11.1 Multicast Addresses

The range of IP addresses is divided into "classes" based on the high order bits of a 32 bits IP address. These classes can be seen in Figure 2.9.

Bits: 0					32	Address Ranges:	
0					Class A Address	0.0.0.0	– 127.255.255.255
1	0				Class B Address	128.0.0.0	– 191.255.255.255
1	1	0			Class C Address	192.0.0.0	– 223.255.255.255
1	1	1	0		Multicast Address	224.0.0.0	– 239.255.255.255
1	1	1	1	0	Reserved	240.0.0.0	– 247.255.255.255

Figure 2.9: IP Address Classes

The one which concerns the multicasting is "Class D Address". Every IP datagram whose destination address starts with "1110" is an IP Multicast datagram. The remaining 28 bits identify the multicast group the datagram is sent to. Every multicast address identifies a multicast group in which there are some number of nodes that listens messages that sent to this group.

There are some special groups, which are should not be used in particular applications due to the special purpose they are destined to:

- 224.0.0.1 is the *all-hosts* group. If you ping that group, all multicast capable hosts on the network should answer, as every multicast capable host

must join that group at start-up on all it's multicast capable interfaces.

- 224.0.0.2 is the *all-routers* group. All multicast routers must join that group on all it's multicast capable interfaces.

- 224.0.0.4 is the all DVMRP (Distance-Vector Multicast Routing Protocol) router, 224.0.0.5 the all OSPF (Open Shortest Path First) routers, 224.0.0.13 the all PIM (Protocol Independent Multicast) routers, etc.

The range from 224.0.0.3 to 224.0.0.255 is assigned for administrative purposes and datagrams destined to them are never forwarded by multicast routers. Similarly, the range from 239.0.0.0 to 239.255.255.255 has been reserved for *administrative scoping* (which is explained in the following section). So normal programs should use the D Class IP numbers other than this IP ranges for their multicasting purposes.

2.11.2 Sending Multicast Messages

The multicast messages are sent as datagram packages which is handled at transport layer with UDP. The other transport layer protocol, which is TCP, is not suitable because this protocol is defined for point-to-point connections.

In principle, an application just needs to open an UDP socket and fill with a class D multicast address with the destination address where it wants to send data to. However, there are some operations that a sending process must be able to control.

2.11.2.1 TTL

The TTL (Time to Live) field in the IP header has a double signification in multicast. As always, it controls the live time of the datagram to avoid being looped forever due to routing errors. Routers decrement the TTL of every datagram as it traverses from one network to another network and when its value reaches 0 the packet is dropped.

TTL in IPv4 multicasting has also the meaning of *threshold.* Its use becomes evident, because of the general use of multicasting. For example, multicasting is frequently used for transforming high volume data from a server to many clients, such as sending video over the network for many clients. For this reason the range of networks that gets this multicast datagrams should not exceed the necessary value because this high volume data overloads the normal network traffic.

With the use of TTL value applications can arrange the destination range of their multicast messages according to the application needs. For some needs, especially when dealing with overlapping regions or trying to establish geographic, topological and bandwidth limits simultaneously, arranging the TTL value is not the solution. To solve these type of problems, *administratively scoped* IPv4 multicast regions were established, which are in the range from 239.0.0.0 to 239.255.255.255.

2.11.3 Receiving Multicast Messages

As said before, multicasting is a receiver-oriented concept. This means that the client who wants to receive multicast messages should indicate this.

2.11.3.1 Joining a Multicast Group

When using multicasting, a process should inform the kernel which multicast group the multicast groups it is interested in. Depending on underlying hardware, multicast datagrams are filtered by the hardware or the IP layer. Only those with a destination group previously registered via a join are accepted.

2.11.3.2 Leaving a Multicast Group

When a process is no longer interested in a multicast group, it informs the kernel that it wants to leave that group. It does not mean that kernel will no longer accept multicast datagrams destined to that multicast group, because some other processes may still be interested in that group.

CHAPTER 3

MULTICASTING SUPPORT FOR DSM SYSTEM

The DSM system presented in this thesis is aimed to increase the efficiency of shared memory parallel programs using the multicasting communications protocol for distributed algorithms. Also multithreading is used to obtain the efficiency of overlapping communications with computation.

The presented DSM system is a distributed run time system. It consists of a number of nodes connected with a network. These nodes communicate with the other nodes both using one-to-one and one-to-many approach. One-to-one communication is simpler because it is done just opening a connection and sending the information through this connection. But in one-to-many communication, it is hard to keep track of individual connections when many one-to-one connection is used for each node, also this process is too time consuming.

Multicasting gives high flexibility to the distributed algorithms. With mul-

ticasting is it too easy and simple to communicate within a group of network nodes. It is particularly advantageous for the situation that when a node wants to send a message to another node and do not know the address or identity of it, it simply sends a multicast message and concerned node gets the message and the others simply ignore the message. Also when an information is to send a group of node, multicasting is again the best choice.

3.1 Run-time System

The run-time system consists of a number of servers (called DSM servers) which are responsible from managing, creating and sharing distributed locks and shared pages. Clients of DSM servers are the actual programs which use shared memory. General picture of the run-time system is shown in Figure 3.1

On each node in the network which use DSM programs, there must be a DSM server. There are no special servers (e.g. masters or slaves) to do some centralized jobs. However, during the initialization process of DSM servers, one of them is assigned as initiator and all other servers take the initial configuration from this server.

Each DSM server and client has an unique identifier which is assigned in the initialization process. DSM servers take their id's from initiator server. Clients ask for an id to its DSM server which is on the same host when it starts to run. Also each variable and distributed lock has an identifier, but this identifier is given by the programmer of the DSM program (i.e. client program). When a client starts to run, it gives the information (i.e. identifier) for each shared

object and distributed lock to its DSM server.

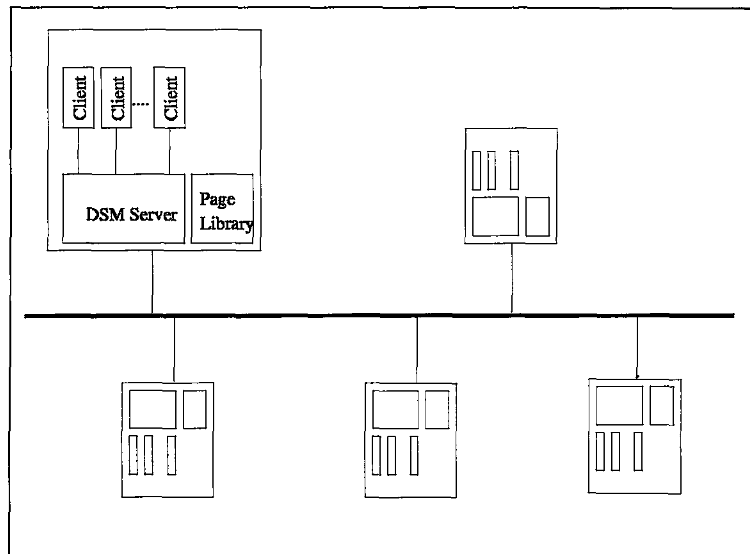


Figure 3.1: Run-time System

3.1.1 Shared Object Creation

All shared objects (scalars, arrays) have a unique identifier, which is defined by shared memory program. When a client program declares that it wants to use a shared object with its identifier and size, it first checks its available shared pages (if any) so that there is enough free space to hold that object. If there is, it attaches this object to this empty space. If there is no space available, it sends a message to the DSM server for creation of a new shared page. This process can be seen in Figure 3.2.

An important point in shared object creation mechanism is, every client program must create its shared objects in the same order. For example, consider a shared memory parallel program, with three shared objects A, B and C. In shared object creation phase all processes should create them in the same order;

that is, first create A, then B and then C (or some other order). The reason for this, is the placement of the shared objects in the shared pages as explained in the previous paragraph. A shared object does not have globally unique identifier, only shared pages have this. So, the placement of the shared object on shared pages should be same for all clients. Creation of the shared object in the same order in all clients of a parallel program guarantees this.

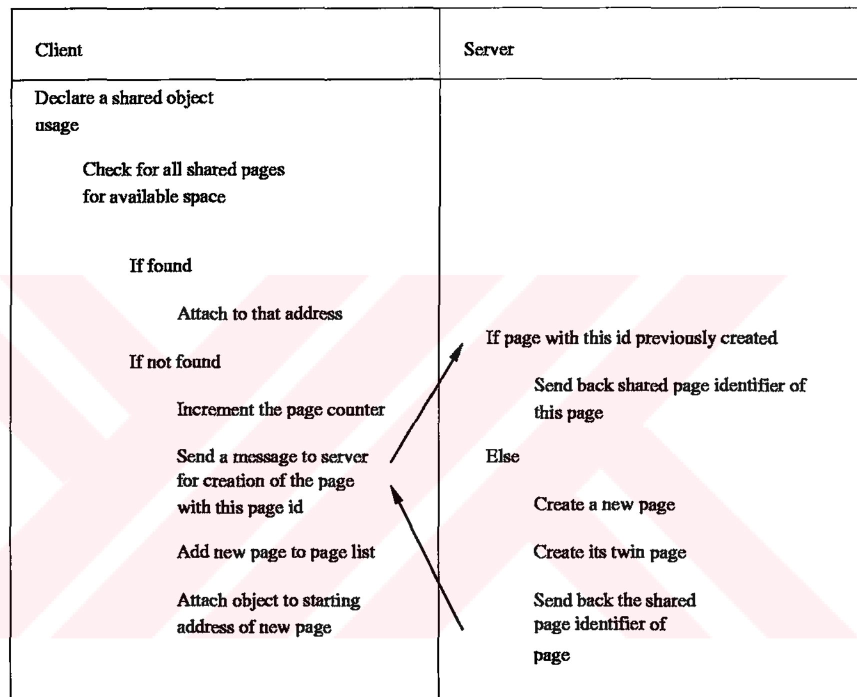


Figure 3.2: Shared Object Creation Algorithm

The globally unique identifier of the shared pages are determined in the shared object creation phase. When a new page requirement appears while creating a shared object, an identifier for the shared page is determined by incrementing a counter in the client program. Because every client uses the same shared object creation order, every client calculates the same identifier for the page that contains the same shared objects.

All clients on a host uses the same page with the help of System V Interprocess Communication (IPC) calls like shmat, shmget. These functions are used for shared memory operations on the same host. Using shmget DSM server gets the page and pass the IPC identifier and the address to the client and client uses the shmat to attach the page to its virtual memory map.

When a new shared page creation request comes to the DSM server, it checks its previously created shared pages, to see the page with this identifier is already created. If created, server simply sends the IPC shared page identifier of the shared page. If this page did not created previously, it creates a page, a twin for that page and sends back the IPC identifier of this newly created shared page to the client program.

When the DSM server creates a shared page, it passes back to client program an identifier of shared page, which is called IPC identifier of page. With this identifier, client program can map this previously created page to its virtual memory map.

In Figure 3.3 on Host A, the DSM server and all client programs use the same physical memory page with the help of IPC shared memories. Of course the accessing addresses of this page can be different, because they map this page to their virtual address spaces. When one of them updates the memory, every other process on the same host automatically sees this change.

When a page is created, a structure shown in Figure 3.4 is created for that page both in DSM server and client program. The *id* is the globally unique identifier of the shared page. The *shmid* is the IPC shared memory identifier of

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

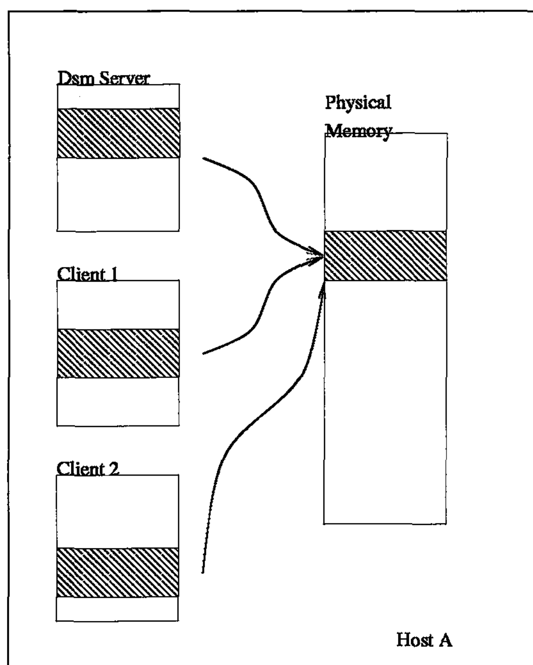


Figure 3.3: Shared Pages in a Host

the shared page. This identifier is used to map the same shared page in all of the client programs and the DSM server in a host. The *address* is the address of the shared page, in the address space. In the structure of DSM page this field holds the address of the shared page in virtual address space of DSM server process. In client programs it holds the address in the virtual address space after mapping the page. The *twin* is the address of page twin. This field is only meaningful in the DSM server because only the server holds a twin of the page. The *start* and *size* fields holds start address and size of the shared page respectively. The *free* field holds the start of the free space (i.e. not dedicated for a shared object). This field is used in client programs for arranging the places of the new shared objects. The *dirty* field indicates whether this page is dirty or not. This information is used in diff creation algorithm.

At creation time, the shared pages are mapped as read-only in client pro-

```
typedef struct {
    int   id           // Identifier of the shared page
    int   shmid        // IPC Identifier
    void * address     // Address of the original page
    void * twin        // Address of the twin page
    long  start        // Start address in address space
    long  size         // Size of the page
    int   free         // Start of unallocated memory in page
    int   dirty        // Indicates whether this page is dirty
} dsmPage_t;
```

Figure 3.4: A DSM Page Structure

grams and *dirty* fields of shared pages in DSM server set as false indicating there is no change in the page content. The reason for this is to find the dirty pages while creating the diffs easily. When a client program tries to update a shared page, a segmentation fault arises. The client program sends a message including the identifier of shared page, to indicate the page update attempt. DSM server makes the shared page dirty, by setting the *dirty* field true. Then it sends the response to the client program to indicate it can continue with its job. The client program makes the shared page writable and continues updating the page.

3.1.2 Distributed Lock Creation

During initialization process of a client, if the client wants to use distributed lock, it must send a message including the identifier of the lock for each lock. Server keeps two fields for each lock. One is the *local* field. If this field id true, whether lock is owned by a local client or lock is on the local DSM server and no one wants to acquire the lock. If the *local* field is false, the lock is on a remote node. The other field for the lock, *hostId*, holds the owner information which is meaningful if lock is local. It holds the information of which client acquired the

lock currently. Also for each lock, DSM server keeps a queue on which multiple requesters of the lock are kept in a first-in-first-out manner.

```
typedef stuct {
    int     id;           // identifier of lock
    char    local;        // indicates whether this lock is local
    int     queue[SIZE]   // queue for the lock
    int     hostId        // current owner of the lock
    int     last          // last owner of the lock
} Lock_t;
```

Figure 3.5: Lock Structure

Every lock in the system is represented by a structure in Figure 3.5. The *id* field holds the identifier of the lock. This identifier is determined by the programmer of the parallel program and sent with the lock create message to the DSM server. The *local* field indicates whether this lock is local or not. If this field is true for a lock, it means that the given lock is either on a local client program or released from a local client and not wanted by any server after this release. The *queue* array holds the candidate acquirer hosts, both local clients and other DSM servers. The *hostId* field holds the current owner of the lock. If no local clients owns the lock, this field contains -1. The TRUE value of *local* and -1 value of *hostId* means that this lock is **free**. If the *local* field is FALSE the *hostId* field does not mean anything because lock is not local and that DSM server cannot know which program has the owner of the lock.

3.2 Distributed Synchronization

Distributed synchronization in the thesis consists of distributed locks and distributed barriers.

3.2.1 Lock Mechanism

When a client wants to acquire a lock, it sends a message to its DSM server indicating the acquire request and including the lock identifier. DSM server checks whether the lock is local. If the lock is local and no one owns it, DSM server simply sends a acknowledgment message indicating completion of acquire. If lock is local and not free, i.e. another local client owns the lock, DSM server enqueues the requesting client to the local queue of the lock.

If the lock is not local, DSM server of client (lets say DSM server A) sends a multicast message indicating the acquire request and enqueues the client to the local queue of the lock. The other DSM servers on which lock is not local simply ignores the message. When the DSM server that owns the lock (lets say DSM server B) gets the external acquire request for the lock, it checks whether lock is free. If it is free, it sends the acknowledgement, else it enqueues the identification number of requesting DSM server A to the queue of lock. In that case, originating client on server A still waits for lock acknowledgement message. The DSM server of the client (server A) begins to deal with other requests, i.e. it does not wait for the answer from DSM server B. After some time, the lock holder client on DSM server B release the lock. At that time, DSM server B sends the acknowledgement message to DSM server A. Server A looks for which local client waits for this lock by dequeuing this first client from the lock queue. Then it sends the lock acquire acknowledgement message to that client.

When a client wants to release a lock, it sends a message to its DSM server.

If queue for the lock is empty, there is nothing to do. If the next acquirer at the head of the queue is a local client, DSM server just sends the acknowledgment message to that client (it was waiting for that message after sending acquire request)

If the next acquirer is another DSM server on remote host, it sends a message for acknowledgment for that lock and also includes the rest of the lock queue (up to next client) and appends its identifier to the tail of the list. By doing it this, way it guarantees that lock comes back to it after new acquirer DSM server (and possibly some other DSM servers) finishes its jobs with the lock.

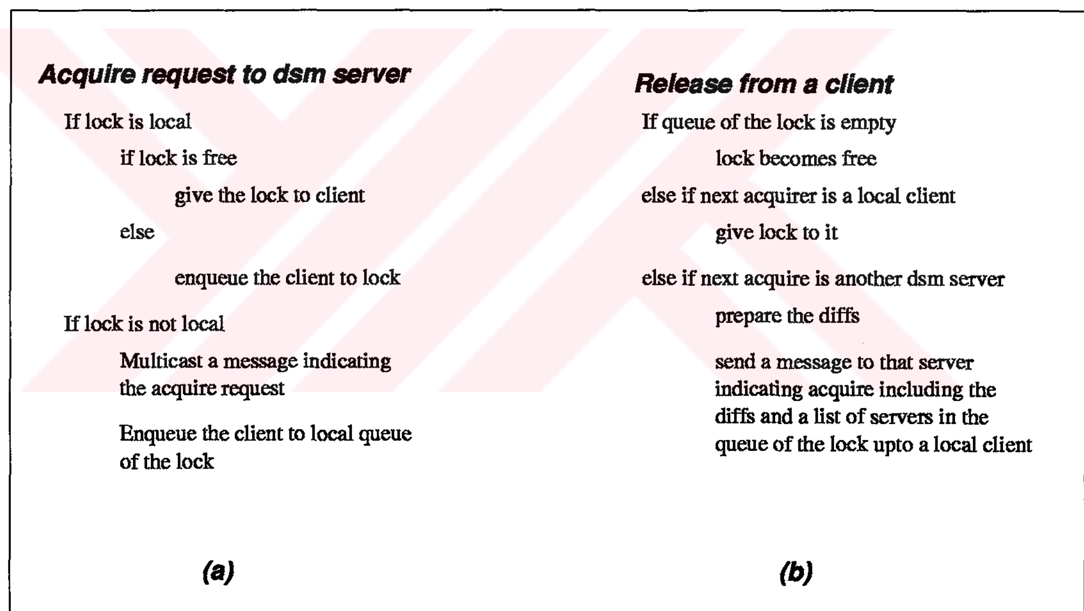


Figure 3.6: Lock algorithms

The algorithms for the lock mechanism are shown in the Figure 3.6.

Let us explain the lock mechanism with an example in Figure 3.7, Figure 3.8 and Figure 3.9. In this example, there are four DSM servers S1, S2, S3 and S4 on four different hosts. The clients C1, C2 and C3 are on DSM server S4, clients

C4 and C6 on server S3, client C5 is on server S1. There is one lock L1, which is acquired and released by the clients.

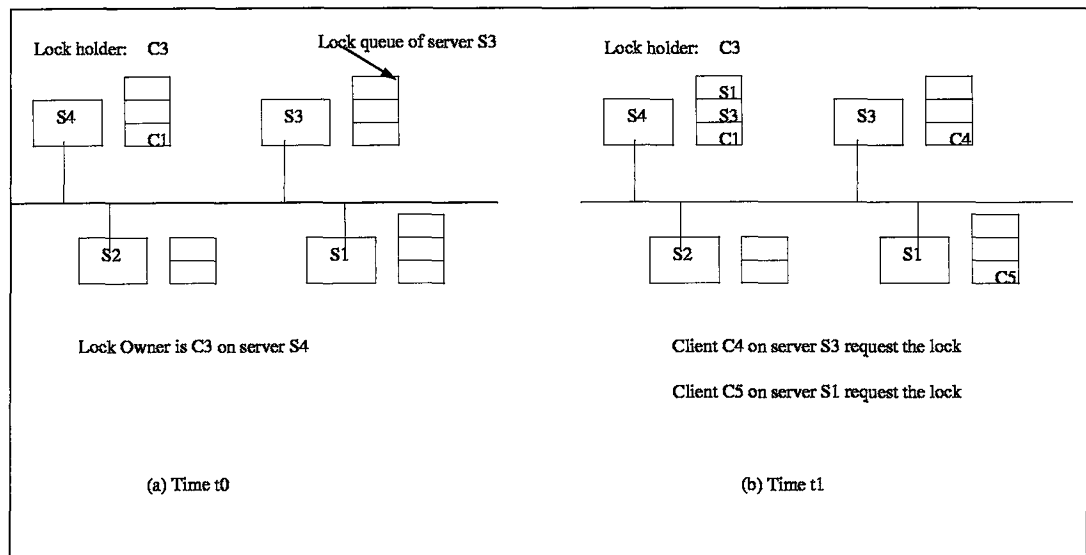


Figure 3.7: Lock Mechanism Example - 1

At time t0 (shown in Figure 3.7 (a)), lock holder is client C3 on DSM server S4. The other clients do not want to acquire the lock at that time. At time t1, the client C4 on server S3 requests to acquire the lock. The server S3 sees the lock L1 is not local to it, and multicast a message for lock request. The servers other than S4 ignore the message because lock is not local to them. S4 enqueues the server S3 to its local lock queue of L1. Meanwhile, client C5 on server S1 requests the lock. Similarly S1 realizes that L1 is not local and multicasts a message indicating lock acquire request. DSM server S4 enqueues S1 to the queue of lock L1.

At time t2, client C3 on server S4 releases the lock. DSM server S4 checks the queue of the lock L1 and dequeues the client C1 from queue and sends the acquire acknowledgement message. At this time, client C1 is the owner of lock.

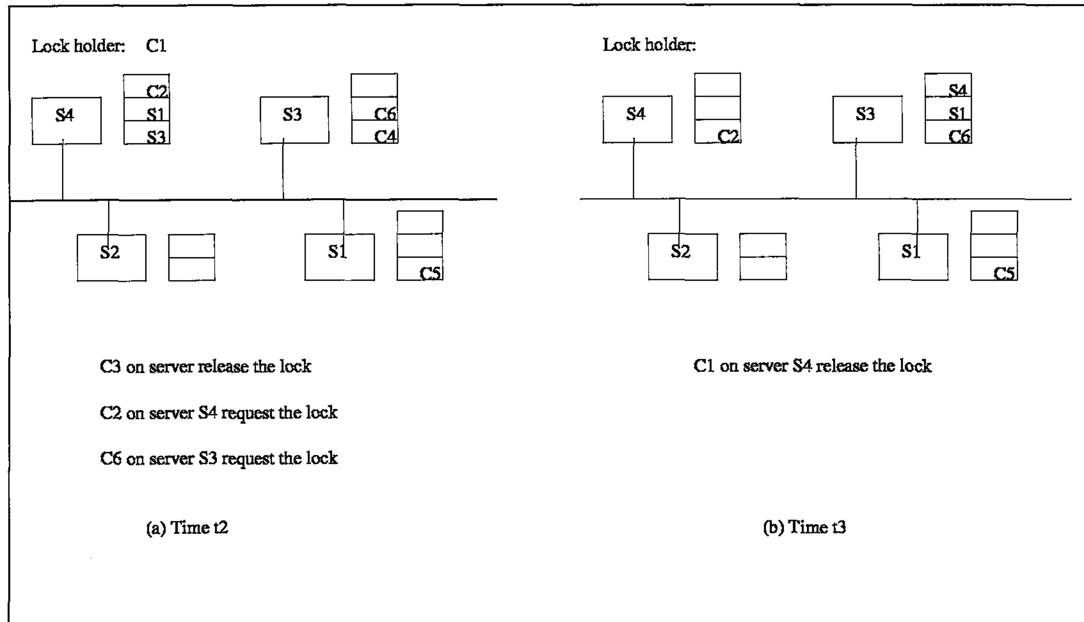


Figure 3.8: Lock Mechanism Example - 2

Meanwhile, client C2 on server S4 requests the lock. Server S4 verifies that lock is local and enqueues C2 to the queue of lock. Also client C6 on server S3 requests the lock. DSM server S3 realizes that lock is not local and it previously sent multicast message for that lock, so it does not send the message again. It just enqueues the client C6 to the local lock queue. Because of previous multicast message, at some time in the future, the lock acknowledgement message will come to it and it can give the lock after the other client which requested the local lock.

At time t3, client C1 on server S4 releases the lock L1. The server S4 looks to queue of lock for next acquirer and it sees that DSM server S3 is at the head of queue. S4 calculates the diffs, dequeues the S3 and next DSM servers up to next client in the queue (C2 in this case) and generates a message including the diffs for updating dirty pages and a server list containing S1 and S4 (itself).

Then it sends this message to S3 using multicasting method. The DSM server S3 enqueues all the servers in server list to its local queue of lock. The aim of this list is directing the next server for this lock. In the example, S4 sends a list containing S1 and S4. This means that when S3 finishes its job with the lock, it will send it to S1 directly, not to go over S4. S4 here means that it wants back the lock because it has another client (C2) that wants to acquire the lock. Of course, S4 here may give the lock to C2 before sending it to S3, but this will not be a fair sharing, because S3 requested the lock before C2. After enqueueing the elements of server list, S3 gives the lock to C4.

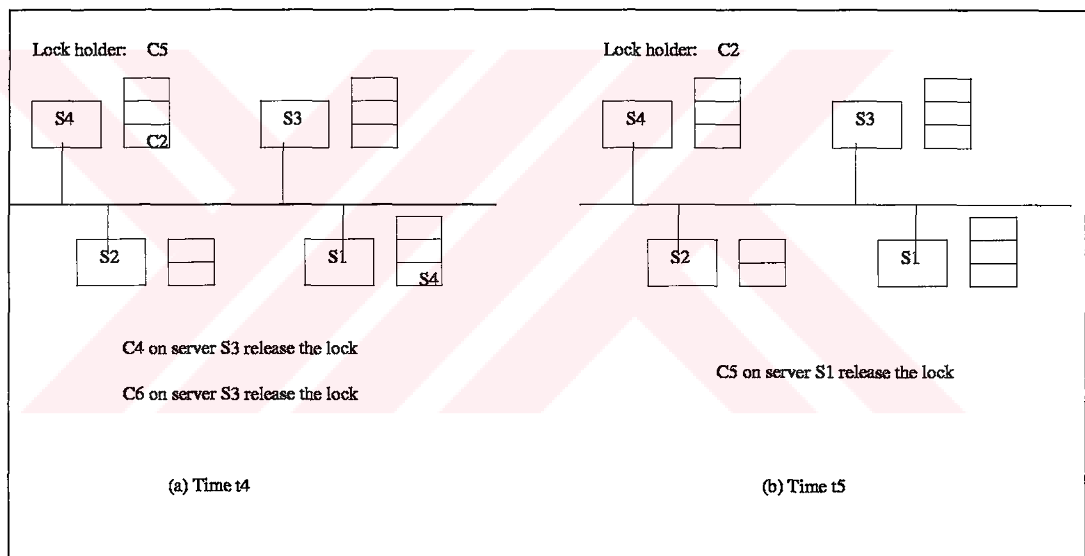


Figure 3.9: Lock Mechanism Example - 3

At time t4, client C4 on server S3 releases the lock and DSM server S3 gives the lock to C6. After that client C6 release the lock. At this time, server S3 looks to the queue and sees that next acquirer of the lock is server S1. S3 sends a message to S1 including diffs and server list containing S4. This time server list does not contain the sending server (S3) because S3 does not have any local

client requesting the lock, so S3 does not want back the lock. S1 sends the lock acquire acknowledgement message to C5 and the current owner becomes C5.

At time t5, client C5 on server S1 releases the lock. S1 gets the next acquirer S4, which is a server, from the lock queue. It sends a message to S4 containing diffs. This time no server list is sent because it has no client wanting the lock and there is no other server in the queue of the lock. When S4 gets this message, it sends the lock acquire acknowledgement message to client C2, then C2 becomes the current owner of the lock.

By implementing this algorithm it is guaranteed that all requesting clients gain fair access to the locks. Fair here does not mean that lock acquisition of the clients exactly matches the lock requesting order. In the same host, for local clients acquisition and request orders matche but in a different host it may not. This is because a lock comes to a DSM server (S1) all clients up to another DSM server (S2) gain access and then server sends the lock to that DSM server. Furthermore, exact request order of all requesting clients can not be known because they are distributed over the different hosts.

The above example explains especially the lock mechanism between the DSM servers. The algorithm of the interaction between a client program and a DSM server is shown in Figure 3.10.

At first, a client program tries to acquire a lock L. It sends a message to its DSM server, the server runs the algorithm mentioned above and at some point in time, it gives the lock to the client program. At that point client program changes the state of all its shared pages to read-only. After some time, the client

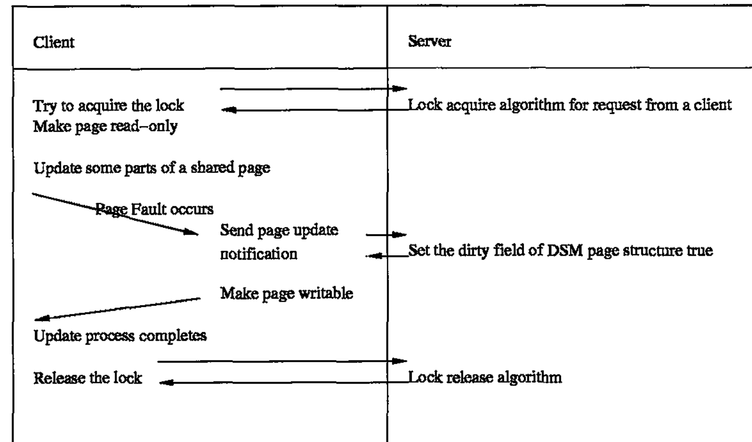


Figure 3.10: Client and DSM Server Interaction in Lock mechanism

tries to update a shared object on one of the shared pages. This try causes a segmentation fault because all the shared pages are read-only in that time. Fault handler in the client program (this handler is a part of DSM library linked to the client program at compile time) finds the identifier of shared page that caused the segmentation fault, that is, the shared page that is to be updated, sends a message including the page identifier to the DSM server. DSM server sets the dirty field of shared page structure corresponding to this shared page and sends a response, which means that client program can continue its execution. When fault handler (of client program) gets this message from server, it makes the shared page writable and the client makes its update to the shared object. After some time, client program releases the lock, and sends a message to the DSM server.

In our lock mechanism, programmer of the parallel program implicitly does the associations between locks and shared memory objects. This means that, the knowledge of accessing a shared object A requires the acquirement of lock

L, is only known by the programmer. For this reason, DSM system cannot make any assumption at lock release time about which shared objects can be updated during the client acquirement time. However DSM server should find the updated fields in the shared memory pages in the system. Comparing each shared page with their twins could cause drawbacks of extra computation for unnecessary updated field search on unchanged shared pages. In some parallel programs, the number of shared memory pages can be too big, and this unnecessary check can be a big drawback. To overcome this problem, DSM server keeps information about which pages are updated, while client program continues. This information is gathered by the fault handler part of the algorithm in Figure 3.10. At the lock acquirement time, the client program makes all shared pages read-only and when it wants to update a shared object, segmentation fault rises. The handler for this fault sends the update information for the page that is to be updated, and makes that page writable. The successive updates to that page do not cause any segmentation fault because the page is writable. Also DSM server is not interested in this successive update information because it only needs to know which pages are updated. Of course, the updates of other pages cause the segmentation fault, and this update information is again sent to the DSM server.

3.2.2 Barrier Mechanism

The barrier mechanism implemented works completely in a distributed manner. Every client that reaches a barrier point sends a message to its DSM server and

waits for the response message from it. DSM server sends back the response when all the other clients in a parallel program reach the barrier point. The algorithm works as in Figure 3.11.

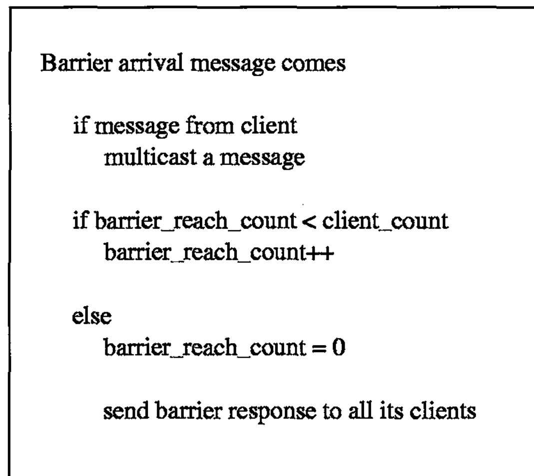


Figure 3.11: Barrier Algorithm

The above algorithm runs when a barrier arrival message comes to a DSM server. First of all, if the message is from a local client of a DSM server, the server multicasts a message to indicate the barrier arrival of the client to the other DSM servers in the system so that they can check the total arrival of barrier. Then, DSM server checks whether total count of clients that arrive this barrier equals to the client count of parallel program to identify that all clients arrives to the barrier.

If the message is not from a local client, the server just increments the barrier reach count of the program and does nothing else. However, if all clients reach the barrier, it sends a message to all clients that connected to itself indicating the barrier arrival of all clients, and then clients continue to work with their jobs. The other clients that run on different machines, i.e. connected to different DSM

servers, gets this message from its servers. The message that is multicasted to other DSM servers gives them enough information about this condition.

3.2.3 Diff Mechanism

When a shared page is created by a DSM server, a twin of this page is also created. The twin of the page holds the last reflected snapshot of the shared page. While the shared memory parallel program goes on, processes of program (DSM clients) updates the shared pages, and at some time this updates must be reflected to other DSM systems.

Diffs are created for reflecting the changes on shared memory pages to the other DSM nodes on the system. They are created by using a word-by-word comparison of dirty pages and twins.

When an acquire request arrives from another DSM server, and if the lock is free, DSM server generates the diffs for all dirty shared pages and send a message including both lock acquire acknowledgement. By this way, there is no need to send extra update message to the other DSM server.

Diff structure is shown in Figure 3.12. As it is seen from this figure, a diff can contain update information for one or more shared pages. For each page, there are one or more page part content which are the updated parts in dirty pages. This page parts are identified by a start address, size and the content of update. The start address is the starting point of the updated part with respect to the beginning of the page. The size is the length of the updated part and the payload is the content of it.

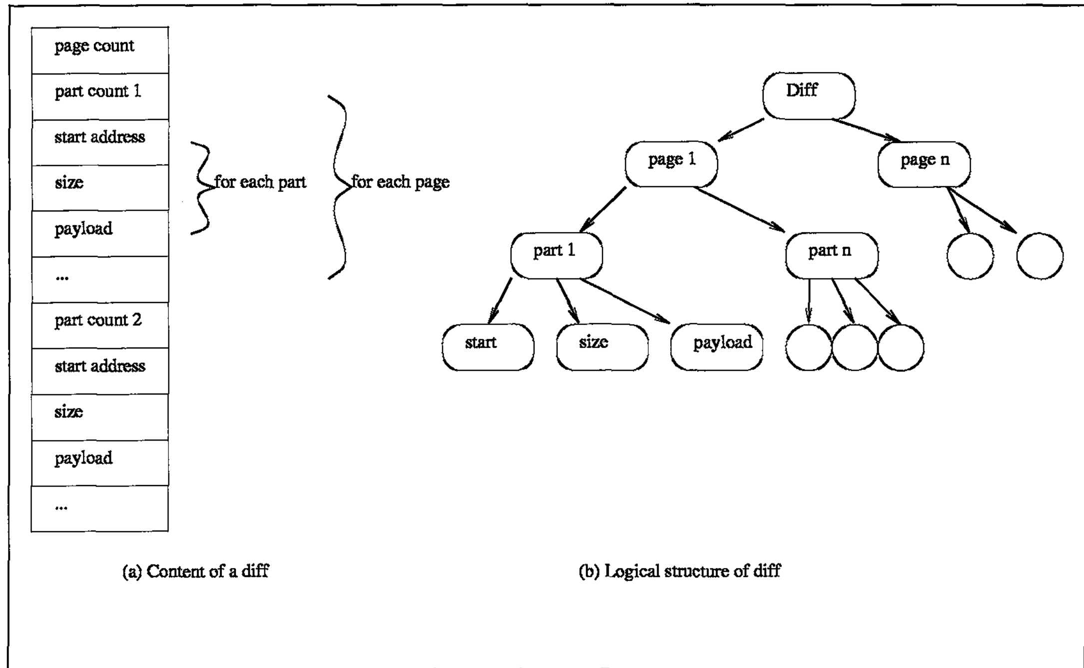


Figure 3.12: Diff Structure

When applying this diff at the lock requester, the twin of the shared page is also updated, so that at the next release point, the changes of client in this host can be differentiated. At the next release point, the diffs should reflect the changes after the acquire of this specific lock, for this reason DSM server refreshes both its original and twin page.

3.3 Consistency Scheme

Consistency scheme is mainly derived from Lazy Release Consistency (LRC). Like LRC, the consistency schema uses lock acquire and release operations as base for consistency transactions. Also the consistency scheme implemented lazily delays the diff creation and update for dirty copies of shared pages until the next lock acquire process. However, with use of multicasting as messaging

method, every DSM runtime system has a lot of knowledge about transactions of other DSM systems.

With use of multicasting and multithreading, while updating shared pages of acquiring process, all other processes shared pages are made up to date. By doing in this way, no other complex diff histories and extra messages are necessary. Multicasting facilitates the consistency method that use no extra messages other than synchronization messages. Of course, consistency messages carry more payload (diffs of dirty shared messages) than the ones in LRC, however, this method is less time consuming because sending one big message through the network takes a shorter time compared to sending two short messages.

The task of creating and applying diffs for shared pages is a function of DSM servers. The multithreading mechanism increases the overall performance because shared memory parallel programs are not interrupted for this process.

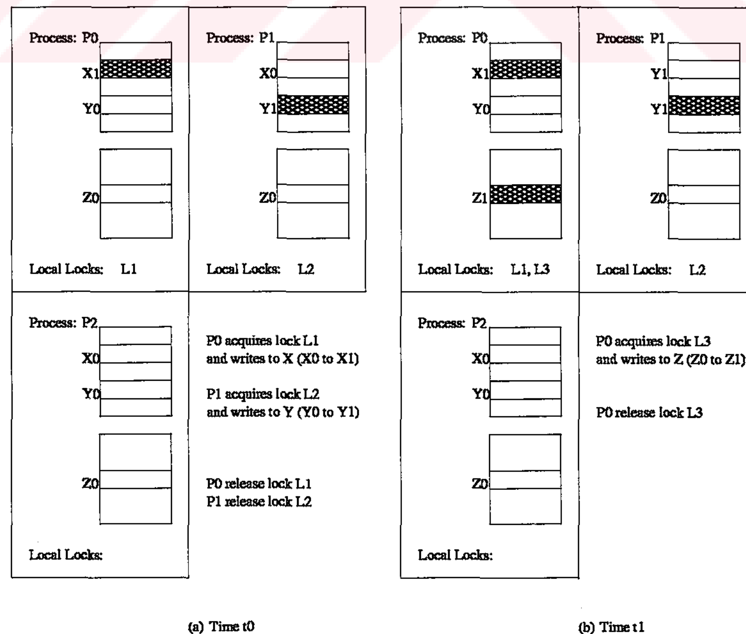


Figure 3.13: Consistency Algorithm Example

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

To explain the consistency algorithm in detail consider the example in Figure 3.13, Figure 3.14, Figure 3.15, and Figure 3.16. In Figure 3.13, there are three processes, which are using DSM system, P0, P1, and P2. These processes share three shared objects (variables) X, Y, and Z, which are distributed on to two different shared page by the DSM run-time system. Also, there are three lock variables L1, L2, and L3 in the system, which are created by the parallel program.

At time t0 in the example (in Figure 3.13), process P0 acquires the lock L1, and writes the shared variable X. In the depicted notation X0 become X1, which denotes the update of shared object, and X is shown as shaded because process P0 has the dirty copy of shared object X. In the initial case, any process does not own locks, and then lock acquiring process is carried out with no extra processing. In the mean time, process P1 acquires the lock L2 and writes to shared variable Y, and Y0 becomes Y1. At the end of these updates processes P0 and P1 releases the locks L1 and L2 respectively.

At time t1, process P0 acquires the lock L3. This step again does not require any processing because P0 is the first acquirer of the lock. P0 then writes to shared variable Z (Z0 becomes Z1) and release the lock L3. At the end of t1, process P0 has the dirty copy of X and Y which are X1 and Y1 respectively, and process P1 has dirty copy of Y which is Y1, as in Figure 3.13 (b).

At time t2, process P2 tries to acquire lock L1. When P2 requests lock L1, its DSM server realizes that this lock is not local and requests it using a multicast message. As the last owner of the lock, process P0 gets the lock request message

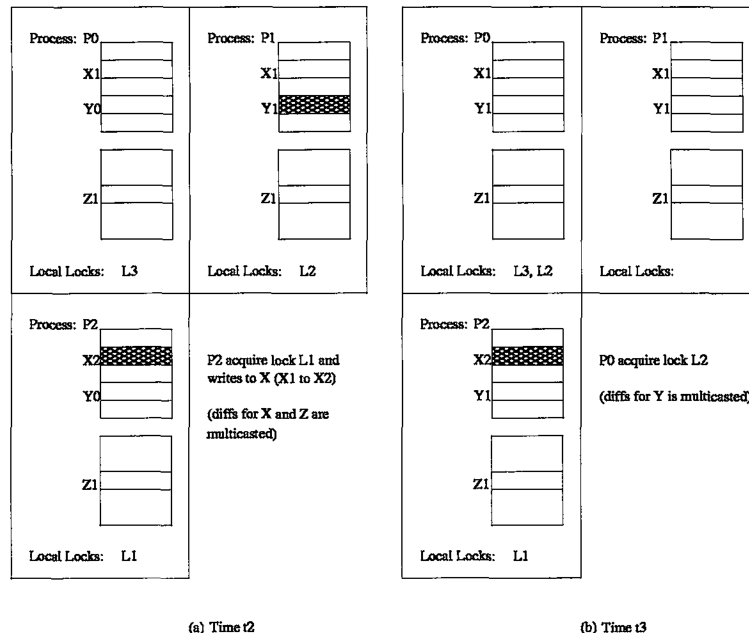


Figure 3.14: Consistency Algorithm Example

and creates the diffs of dirty pages it owns (the page that contains X and Y in this situation). After diff creation process, P0 sends a multicast message to process P2 for acquire acknowledgement. This message also contains the diffs for shared pages for validating the remote shared pages in the other processes. All processes receive this acquire acknowledgement message because the sender uses multicasting. All processes other than target receiver of the message ignores the lock acquire related part, and apply the diffs in the message. The message in our example contains the diff for X1 and Y1, which are dirty objects in process P0. DSM servers of P0, P1 and P2 apply this diff to the shared page to make it up to date. After diff application process, DSM server of P2 sends a message to P2 to indicate the completion of lock acquire process. P2 then updates X and X1 becomes X2. At the end of t2, P1 has the dirty copy of X (X2).

At time t3, P0 tries to acquire L2. DSM server realizes that L2 is not local,

and multicasts a message indicating lock acquire request. As the last owner of lock L2, DSM server of P1 gets the message, creates the diff for Y1 and sends a multicast message for lock acquire acknowledgement to P0. DSM servers of P0 and P1 apply the diff for Y in this message.

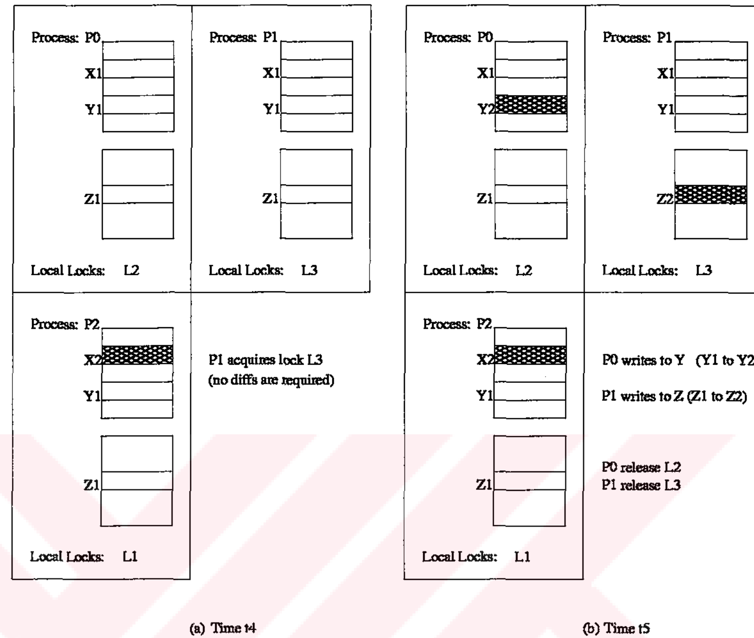


Figure 3.15: Consistency Algorithm Example

At time t4, process P1 tries to acquire lock L3. As L3 is not local, DSM server of P1 sends a multicast message for acquisition of lock L3. P0 responses back with a multicast message for acknowledgement. This time acknowledgement message do not contain any diff, because P0 does not have any dirty copy of shared objects.

At time t5, P0 updates Y (Y1 becomes Y2) then releases the lock L2 and updates Z (Z1 becomes Z2) then releases the lock L3. At the end, P0 has the dirty copy of Y (Y2), P1 has the dirty copy Z (Z2) and P2 has dirty copy of X (X2) as shown in Figure 3.15.

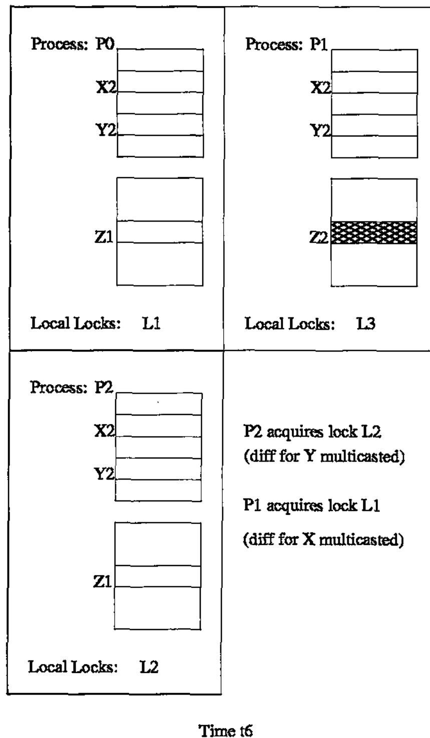


Figure 3.16: Consistency Algorithm Example

At time t6, P2 tries to acquire lock L2. In the lock acquisition acknowledgement message from P0 there is the diff for Y2 and all processes update their copies of Y. P1 tries to acquire lock L1. P1 tries to acquire lock L1. In the lock acquisition acknowledgement message from P2 there is the diff from X2 and P0 and P1 update their copies of X.

When acquirer client wants to access to one of the shared pages, DSM server checks if it finishes applying the diff of that page (if there is any). If it finishes, it immediately gives the access to that page to the client, else it blocks the client so that it can finish the applying of the diffs of that page. After that it gives access to the client.

By intermixing the application of diffs and client program processing some extra utilization can be gained, for example after getting the positive acknowl-

edgment for the lock acquisition client continues to run and if for some time it does not have processing the shared pages, DSM server can finish the application of the diffs so that client has not extra blocking for the application of diffs.

CHAPTER 4

EXAMPLE APPLICATIONS

In this chapter two example shared memory parallel applications, which are used to test and evaluate the DSM system implemented with this thesis, are described. All the examples in the following sections are tested on the an envirement with the following properties. Because the DSM system implemented runs on Solaris UNIX systems, all parallel programs again runs on Solaris UNIX. The hardwares are Sun Sparc Workstations. The source code of the example programs and DSM system implementation are included in the CDROM which is attached to the report.

4.0.1 Travelling Salesman Problem (TSP)

This algorithm takes an input set consisting of N cities with weighted path lengths connecting each city. The problem is to find the path with minimum total length that passes through each city exactly once, returning to original

city at the end. A complete route through all cities is called a *tour* [27].

The program maintains the length at the current minimum tour found so far in a global variable protected by a lock. This allows the search space to be pruned by abandoning any partially evaluated tours that exceeds this global minimum. This method is called as, depth-first branch-and-bound algorithm [24, 25, 26].

An important part of the algorithm is the task of dividing the search space to the number of clients in the parallel program. This goal is achieved by dedicating one of the parallel processes as a distributor and this distributor divides the search space for the other worker processes. The distributor initially starts to create the search space until there is enough number of search roots in the space.

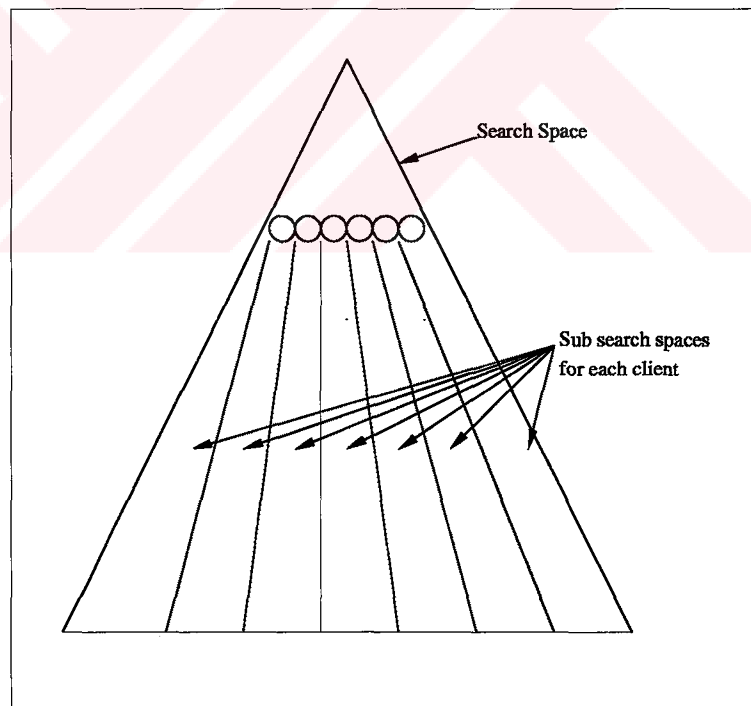


Figure 4.1: TSP Search Spaces

As it is seen from Figure 4.1 the distributor process begins to process the

search space. Every step in the graph traverse creates a sub search space on the graph. At one point, there will be enough number of sub search spaces and the distributor starts the worker processes for finalizing the search in that sub search space to find a complete tour.

Every client that finds a tour, checks the total length of the tour with the global minimum length. If it is less than the global, it replaces the global tour with its own, else throws it.

In the DSM implementation, the program maintains a set of shared objects which hold the subpaths generated by the distributor. There is also a shared object that holds the globally best tsp tour that is generated until the current time. When a client starts processing, it acquires a lock for accessing the global subpath list. It gets a subpath from the list, decrements the total number of unprocessed subpath count and then releases the subpath list lock. It places the subpath it gets from the shared memory system to its stack and starts to traverse the graph to find a tsp tour. When it finds a tour, it tries to check that this tour is a better tour than the globally best tour. To do so, it acquires a lock that guards the globally best tsp tour. It compares the cost of globally best tour and the newly found tour. If the new tour is a better tour, than it updates the shared object so that all other clients sees this as the best tour. After all the subpaths on the shared objects are processed, the clients exist and distributor node outputs the globally best tour, which was on a shared object, and exits.

In the message passing implementation, the program again maintains a set of shared objects that holds the subpaths generated by the distributor. The

distributor first generates the subpaths, and then starts to listen a predefined socket for clients to connect. When client nodes start processing, they connect to the distributor and try to get a subpath. If there is a subpath, the distributor sends this subpath to the requester and the client starts to process this node. If the client finds a better tour, it sends this tour and its cost to the distributor. When client finishes all subpaths, programs are terminated.

Table 4.1: Running times for TSP problem

TSP	P	2		4		8	
N		MP	SM	MP	SM	MP	SM
8		84	95	99	115	120	132
10		140	131	162	168	173	179
11		509	517	511	521	518	525
12		3865	3825	3546	3487	3256	3361
13		41176	41231	37974	38014	38344	38455

In Table 4.1 the running times of shared memory (SM) and message passing (MP) implementations of TSP problem are shown. Unit of time here is clock tick. Each of the running times in the table are the average of five executions. The reason for that is each execution can result in different (but simular) running times. Example executions run on 2, 4 and 8 processors (P) and 8, 10, 11, 12 and 13 nodes (N) graphs. As it can be seen from the table, the time differences between the shared memory and message passing implementations are small. The fluctuations between these two implementations comes from the underlying network and processor load changes.

4.0.2 Parallel Quicksort

The serial quicksort algorithm takes an input array of numbers, and selects a pivot randomly. The list is divided into two portions, one of which contain values smaller than pivot, the other portion contains the rest of the array. The algorithm recursively sorts the divided portions.

Because of the divide-and-conquer nature of the quicksort algorithm, parallelism can be easily added to it. There are many parallel quicksort algorithms in the literature [28, 29, 30, 31].

The parallel sorting algorithm implemented works in the following way. Initially, a list to be sorted is distributed among the processors. Then a master processor randomly selects P-1 (P is the number of processor) pivots from the array. After that, processors divide their own part in to the P number of divisions using the P-1 pivots in parallel. After this division process, processors send the divisions to the master. Master merges these divisions and generates P number of divisions that are non-sorted. Master sends these new divisions to the P processors. Processors sort their own divisions using the quicksort algorithm. Then processor sends these sorted divisions to the master and the master process puts these divisions into a new array according to the identifier of sender processor. The new array is the sorted array.

Both message passing and shared memory implementation of the algorithm works in the same way. The difference between them is how data is passed between the processors. Also some temporary storage in the message passing

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ

implementation is not required by the shared memory implementation.

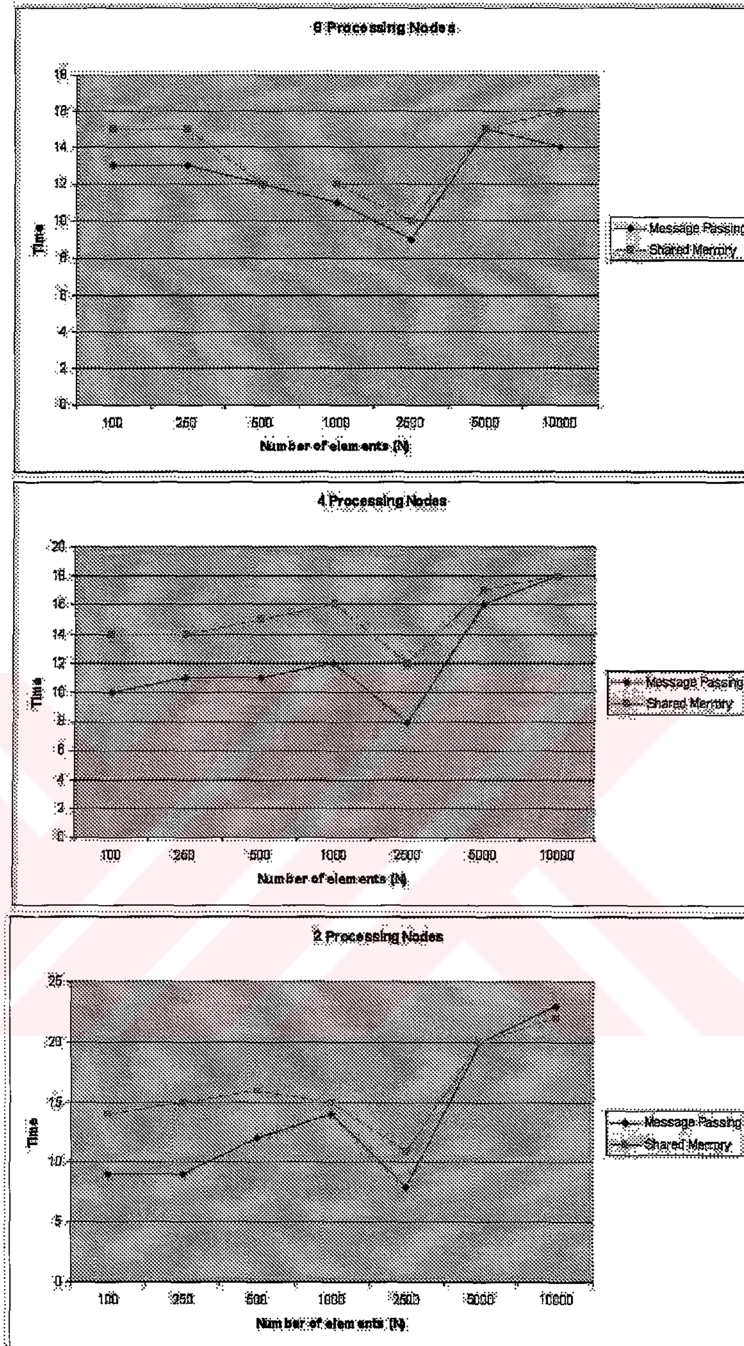


Figure 4.2: Running Times for Parallel Quicksort

The implementation of the shared memory program is really easy, because the programmer only makes sorting and memory copy operations. Also the coping of the data to the appropriate places in the shared address space decreases some of

the extra processing done in the message passing implementation. For example, the line of code in the shared memory implementation is 495, however in the message passing implementation the line of code count is 799, nearly double of the shared memory one.

In the Figure 4.2, there are three charts which show the running times of message passing and shared memory implementation of the parallel quicksort algorithm. The unit of the times here is clock tick. In this running time analysis, we propose to figure out the relationships between the running times of shared memory implementation and message passing implementation of the parallel quicksort algorithm.

As it can be seen from the charts, the running times of the shared memory implementation are very close to the message passing implementation ones. The reason for a sudden decrease when N = 2500 is the input distribution. The performance of implemented parallel quicksort algorithm is very sensitive to the distribution of the elements and also the distribution of the randomly selected pivot values.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this study, a distributed shared memory system that can be used to develop distributed parallel programs is designed and implemented. Multicasting communication method is used to develop effective group communication algorithms to increase the efficiency of the proposed system.

The approach presented in the thesis for the distributed shared memory systems is to increase the efficiency of DSM systems by introducing efficient distributed algorithms with effective group communication. For example, when one DSM system does not know which DSM node has a lock, it does not send successive messages to other systems, it just sends a multicast message, and the relevant server gives the appropriate response. As it is seen, there is only one request and one response. Again because of using multicasting, the dsm system implemented does not require complex history mechanism for previous updates. This is the main advantage when compared with Lazy Release Consis-

tency mechanism.

Also by overlapping communication with computation, the overall performance is increased. As explained in the previous sections, a distributed shared memory parallel program consists of some number of distributed client programs that runs parallel. Other than these DSM client programs, there are DSM servers whose job is to support a distributed shared memory environment for the client processes. They keep the shared pages consistent, provides the synchronization primitives to DSM client processes. While doing this functions, they do not interrupt the clients if it is not necessary. Also the communication with other DSM systems is supplied by the DSM servers, i.e. the DSM parallel processes does not deal with communication, they just do the computation related with the parallel program.

Another point is the combining of the synchronization with consistency mechanisms. As explained earlier, the synchronization messages contain also the consistency information. For example, when a DSM system acquires a lock from another, lock acknowledgement message includes also the diffs from the releaser system. The acquirer first applies these diffs to make their shared pages consistent then makes the synchronization related task. Using this method, only synchronization related messages pass through the network and dramatically decrease the total number of network messages, which is a costly operation.

As a future work to this study, fault tolerance property can be added to the DSM system. In the current implementation, when one of the DSM servers crashes the shared memory programs running on the same network node are also

disconnected from the DSM system. All the information on that DSM server is lost in this situation including the dirty pages, and locks. With fault tolerance properties, the remaining DSM server can regenerate the lost information so that the other shared memory programs which has not been crashed can continue with the execution.

REFERENCES

[1] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers, C:28:690-691, Sep 1979

[2] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-Related Information in Distributed Shared Memory Systems. ACM Transactions on Computer Systems, 13(3):205-243, Aug 1995.

[3] P. Keheler, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In Proc. of the 19th International Symposium on Computer Architecture, pages 13-21, May 1992.

[4] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessor. In Proc. of the 13th Annual International Symposium on Computer Architecture, pages 434-442, May 1986.

[5] S. V. Adve and M. D. Hill. A unified formalization of four shared memory models. IEEE Transactions on Parallel and Distributed Systems, 4(6):613-624, Jun 1993.

[6] J. R. Goodman. Cache consistency and sequential consistency. Technical Report, CS-1006, University of Wisconsin-Madison, Feb 1991.

[7] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway Distributed Shared Memory System. CMU Technical Report CMU-CS-93-119, Feb 1993.

[8] Liviu Iftode, Jaswinder Pal Singh, and Kai Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. Proc. of the 8th ACM Annual Symp. on Parallel Algorithms and Architectures (SPAA'96), pages 277-287, Feb 1996.

[9] Steve Deering. RFC 1112, Host Extensions for IP Multicasting.

[10] Gudjon Hermannsson and Larry Witty. Fast Locks in Distributed Shared Memory Systems. 27th Ann. Hawaii Int. Conf. on System Sciences, I:574–583, January 1994.

[11] B.N. Bershad and M.J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Sept. 1991.

[12] Peter Keleher. Lazy Release Consistency for Distributed Shared Memory. Doctor of Philosophy thesis. Rice University Computer Science Department. Jan 1995.

[13] T. Johnson and R. Newman-Wolfe. A Fast and Low Overhead Distributed Priorty Lock.

[14] Alain Kagi, Doug Burger and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In Proc. of the 24th International Symposium on Computer Architecture, pages 170-180, Jun 1997.

[15] John M. Mellor-Crummey and Michael Scoot. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM Transactions on Computer Systems, 9(1):21-65, 1991.

[16] Honghui Lu, Sandhya Dwarkadas, Alan L. Cox and Willy Zwaenepoel. Message Passing Versus Distributed Shared Memory on Network of Workstations. In Proceedings SuperComputing '95, Dec 1995.

[17] Ron Minnich. ZOUNDS: Zero Overhead Unified Network DSM System. Sarnoff Technical Report. 1996.

[18] John B. Carter. Efficient Distributed Shared Memory Based On Multi-Protocol Release Consistency. Doctor of Philosophy thesis. Rice University Computer Science Department. Sep 1993.

[19] Oliver E. Theel and Brett D. Fleisch. A Dynamic Coherence Protocol for Distributed Shared Memory Enforcing High Data Availability at Low Costs. Technical Report THD-BS-1995-03, Department of Computer Science, University of Darmstadt, May 1995.

[20] Cristiana B. Seidel, R. Bianchini and Claudio L. Amorim. The Affinity Entry Consistency Protocol. Proc. of the 1997 International Conferance on Parallel Processing, pages: 208-217. August 1997.

[21] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Rice University ECE Technical Report 9512. Sep 1995.

[22] Cristiana Amza, Alan Cox, Karthick Rajamani and Willy Zwaenepoel. Tradeoffs Between False Sharing and Aggregation in Software Distributed Shared Memory. In the Proc. of the Sixth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP'97), pages 90-99, June 1997.

[23] Madhavan Parthasarathy. Implementing Shared Memory on Large Scale Multiprocessors. Thesis of Master of Science in Electrical Engineering of the University of Illinois, 1992.

[24] Weixiong Zhang. Depth-First Branch-and-Bound versus Local Search: A Case Study. In Proc. 17th National Conf. on Artificial Intelligence (AAAI/IAAI), pages 930-935, 2000.

[25] V. Nageshwara Rao and V. Kumar. Parallel depth-first search, part I: Implementation. International Journal of Parallel Programming, 16 (6):479-499, Dec 1987.

[26] Nihar R. Mahapatra and Shantanu Dutt. Sequential and Parallel Branch-and-Bound Search Under Limited-Memory Constraints. In Proc. Parallel Optimization Colloquium, pages 147-166, Mar 1996.

[27] D. Applegate, R. Bixby, V. Chv'atal, and W. Cook (1999). Finding tours in the TSP, Report Number 99885, Research Institute for Discrete Mathematics, Universitat Bonn.

[28] B. Abali, F. Ozguner and A. Bataine. Load Balanced Sort on Hypercube Multiprocessors. 5th Distributed Memory Computing Conference, pages: 230-236, 1990.

[29] J.S. Huang and Y.C. Chow. Parallel Sorting and Data Partitioning by Sampling. COMPSAC, pages 627-631, 1983.

[30] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling Journal of Parallel and Distributed Computing. 14(4):361-372, 1992.

[31] Hui Li and Kenneth C. Sevcik. Parallel Sorting by Overpartitioning. Technical Report CSRI-295, February 1994.

T.C. YÜKSEKÖĞRETİM KURULU
DOKÜMANTASYON MERKEZİ