Supporting a Flexible Parallel Programming Model on a Network of Non-Dedicated Workstations

by

Shih-Chen Huang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

January 2000

Approved: _____

Zvi M. Kedem

To my parents and my wife

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Zvi Kedem, for his guidance, support, and encouragement over these years. After leaving school for four years, he helped me to finish my study and made this dissertation possible.

I am especially indebted to Arash Baratloo who brought to my attention the starvation problem in my system. His insights and suggestions have greatly improved the content and clarity of this dissertation.

Special thanks to my committee members, Benjamin Goldberg and Ernest Davis, for giving me many helpful suggestions.

I awe many thanks to my colleagues at AT&T, especially to Wang Tsai and T.C. Yu. Without their encouragement, I will not have the courage and strength to come back to school and finish my study.

This work is dedicated to my family, for their love. I would like to thank my parents and my sister for their unconditional support over the past ten years, both financially and morally. I am much obliged to my dear wife for her understanding

and encouragement during these years. I would also like to take this opportunity to thank her for taking good care of our lovely kids. I would love to share everything with her in my life.

Preface

A network of non-dedicated workstations can provide computational resources at minimal or no additional cost. If harnessed properly, the combined computational power of these otherwise "wasted" resources can outperform even mainframe computers. Performing demanding computations on a network of non-dedicated workstations efficiently has previously been studied, but inadequate handling of the unpredictable behavior of the environment and possible failures resulted in limited success only.

This dissertation presents a shared memory software system for executing programs with nested parallelism and synchronization on a network of non-dedicated workstations. The programming model exhibits a very convenient and natural programming style and is especially suitable for computations whose complexity and parallelism emerges only during their execution, such as in divide and conquer problems. To both support and take advantage of the flexibility inherent in the programming model, an architecture that distributes both the shared memory

management and the computation is developed. This architecture removes bottlenecks inherent in centralization, thus enhancing scalability and dependability. By adapting available resource dynamically and coping with unpredictable machine slowdowns and failures, the system also supports dynamic load balancing, and fault tolerance—both transparently to the programmer.

Contents

List of Figures

Chapter 1

Introduction

There are many benefits for using workstations (including personal computers) over mainframes or specialized parallel computers for computationally demanding parallel applications. Workstations are widely available, much more cost effective, and easier to maintain. However, networking many workstations together does not give users the computational power equivalent to mainframe computers unless the parallel programs can be executed on the network efficiently, utilizing all the available resources. It is especially advantageous to use distributed, non-dedicated workstations that are shared by different users, since these workstations can provide computational resources at minimal or no additional cost. Non-dedicated machines are shared by different users whom may run local applications. A typical example is the workstations used in business or university environment where they

spend a lot of time idling or running non-CPU intensive tasks like web browsing. The combined computational power of these otherwise *wasted* resources can outperform mainframe computers if they are harnessed properly. This observation leads us to search for a way of doing demanding computations on top of a network of non-dedicated workstations efficiently. Research with similar goal has previously been undertaken, but with limited success. We plan to advance the state of the art in this area.

1.1 Challenges

There are many challenges in designing an efficient system on top of a network of non-dedicated workstations. A critical difficulty is that the execution environment is often unreliable and unpredictable. The environment is often asynchronous, failure prone, and have a dynamically changing computing environment, as we explain next.

- *The environment is asynchronous*: The network often consists of machines with varying configurations. Even workstation of the same model may have different performance characteristics since they may have different amount of RAM, different CPU speed, and different disk/virtual memory size. Therefore, workstations perform at different effective speeds.

- *The environment is failure prone*: The execution environment is vulnerable to failures. Non-dedicated workstations are subject to being turned off, disconnected from the network, or run out of resources since other users have access to the workstations. A collection of networked workstations used by people in a building is a typical example of a local area network of non-dedicated workstations. In such environment, the individual user of each workstation may start his/her local computation and cause failure of the global parallel application. The system must ensure that withdrawal of some individual machines from the parallel computation will not cause the entire program to abort. The computation should quickly re-adjust itself.

- *The environment changes dynamically*: The non-dedicated nature of the system allows workstations to be shared by other users, who may claim the computational resources of the workstations at any time. This results in an unpredictable, dynamically changing environment, in which the resources available to our system varies from time to time. This encompasses the environment that is asynchronous and failure prone, since the availability of the resources causes machines to execute programs in our system at different effective speed and to become intermittently available to the system. Hence, the system must adapt to the continuously changing environment to utilize all available resources.

Beside the challenges of the execution environment, the programming model also needs to be considered. Many of the programming models for distributed platform require the programmers to take care of many tedious details when writing distributed programs, including messaging, data and computation distribution, process control, fault handling, etc. These tasks make the user programs difficult to construct and very complex. A well-designed programming system should provide a simple yet flexible parallel programming models, hiding many details that reflect the underlying execution environment. Failures and dynamic load balancing should be handled transparently by the runtime execution system. More importantly, this programming model must be feature rich. Synchronization and structured parallelism with nested jobs that allows any jobs to create additional parallel jobs should be included.

1.2 Features

Our goal is to provide programmers a flexible, easy-to-use parallel programming model and a high performance execution environment that can carry out parallel programs written for such model over a network of non-dedicated workstations. To achieve this goal, our system must be dependable, efficient, and be able to the adapt to the dynamically changing environment.

The features of our system is summarized below:

- *Simple, easy-to-understand parallel programming model* We augmented C/C++ language with simple parallel constructs to make the parallelization of a sequential program easy. The parallel programming model uses virtual shared memory to hide the actual memory structure of the underlying environment, adds few simple keywords to construct and to control structured parallelization, and includes some advanced features like nested parallelism and synchronization.

- *Separation of logical parallelism from physical execution environment* Our system provides an *ideal parallel machine* for the programmers to write their program on. Such ideal machine has unlimited number of processors, a globally shared memory, and never fails. In reality, the targeted environment is a collection of non-dedicated workstations with limited computational resource and prone to failure. Our system bridges the gap between the ideal machine and the actual execution environment transparently and efficiently, and does it without modification of the underlying operating system.

- *Nested parallelism* Nested parallelism allows programs to explore additional parallelism within a parallel block. Such parallelism allows very natural and convenient programming style and is especially suitable for computations whose complexity and parallelism emerge during their execution, like parallel quicksort where the pivot position cannot be determined in advance. Our sys-

tem supports nested parallelism with arbitrary parallelism depth by explicit definition and/or by recursion.

- *Synchronization* Our system provides two types of synchronization, *implicit barrier synchronization* at the end of each parallel block, and *explicit locking mechanism* specified by the programmers. Synchronization is a convenient vehicle for parallel jobs to exchange data and to control the execution sequences. We provide simple *lock/unlock* construct for users to guard the critical section.

- *Dynamic load balancing* Our system dynamically distributes the work of a parallel program over a network of asynchronous workstations. Although these workstations have different computational resources available as we stated earlier, the system can utilize all available resources to achieve high performance. The unique *prioritized eager scheduling algorithm* allows us to find available resources and to distribute dynamically created parallel jobs efficiently.

- *Adaptability* Our system is able to adapt the dynamically changing computational environment in a network of non-dedicated workstations. Workstations can be added to or removed from the system during the execution of a parallel program. Slow machines or workstations with little computational resources

do not become the performance bottleneck.

- *Fault tolerance* The execution of user programs are resilient to failures in our system. The system allows all but one machines to fail while maintaining a correct computation of the user program. Fault-tolerant features are an integral part of our system, without additional fault detection and recovery mechanisms like checking points and roll back. These is no additional overhead if no failures actually occurred.

1.3 Contributions

Our work is a continuation of research reported in [29, 20, 4, 39, 37, 35, 36, 38, 40]. We developed several unique features in our system:

- *Novel techniques to handle nested parallelism and synchronization* Several new techniques are developed in our system in order to cope with nested parallelism. These techniques include *nested two-phase idempotent execution strategy* and *prioritized eager scheduling algorithms*. We also developed a locking mechanism in a fault-tolerant system without using traditional check pointing and roll back techniques. Duplicated copies of lock requests are allowed, and techniques are developed to detect and solve duplicate lock problems, including *variable versioning* and *request sequence*.

- *Well-distributed execution environment* We developed a novel technique that allows user jobs to turn into memory management servers while they are waiting for their children jobs to finish. Hence the execution environment in the computational and memory management functionality is almost fully distributed. The result is high degree of scalability, more flexible and adaptive load balancing, and higher degree of fault masking. Although scheduling and synchronization services are centralized, they provide essential functions that are not computational intensive and have low network traffic.

- *Fault-tolerant services* Beside failure masking of the user programs, we developed fault tolerant techniques for each of our system modules Unlike some systems where all the essential system functions reside on a single machine and cannot fail, our system modules can reside on any machines and tolerate failures. Recovery of the failed modules is automatic and invisible to the users.

We provide a detailed comparison to related research in Section 7.2.

1.4 Outline of the Dissertation

This dissertation presents the features, design, implementation, and experimental results of our system. It is organized as follows:

Chapter 2 describes the key concepts and techniques used of this research.

Chapter 3 introduces the programming model, i.e. the syntax and semantics used to express parallelism. An example program, with the step-by-step parallelization of the sequential quicksort algorithm, is included at the end of this chapter.

Chapter 4 discusses the system architecture. In particular, the various management functions and execution strategies are described in detail.

Chapter 5 introduces the failure model and fault tolerant techniques to mask failures in various system components.

Chapter 6 presents the experimental results.

Chapter 7 describes several aspects of parallel computing and distributed systems related to the system.

Chapter 8 summarizes our work.

Chapter 2

Key Concepts and Techniques

In this chapter, we will discuss the key concepts and techniques used in our system. Some of these concepts and techniques were developed in previous research [29, 20, 4, 39, 37, 35, 36, 38, 40], including the *abstract execution model*, *two-phase idempotent execution strategy*, and *eager scheduling*. However, the introduction of the new features in our system like nested parallelism and synchronization make these techniques inadequate. We will present both the original ideas and the improved techniques used in our system when we encounter them in the following sections.

2.1 Network of Non-dedicated Workstations

We concisely restate the properties of the target platform due to its importance to our design. A network of non-dedicated workstations is a set of workstations connected by a network. We assume that the workstations are homogeneous in terms of machine code compatibility and operating system, i.e. they can all execute program codes generated for a specific target, like the Intel x86 family machines running Linux. These workstations, although homogeneous in machine code and operating system, may have different configurations for each machine like CPU speed and memory size. Hence they are not uniform in processing speed.

Non-dedicated workstations are shared by different users. Users of these machines may run various applications ranging from text editor to complex mathematical calculation. Sharing workstations with other user applications causes the workstations to be only partially or transiently available to our system. Furthermore, these machines may become totally unavailable when being turned off, disconnected from the network, run out of resources, or other mishaps. Hence, network of non-dedicated workstations is an unpredictable, dynamically changing environment where the load, resources, and availability of each workstation vary from time to time.

2.2 Abstract Execution Model

We provide an abstract parallel machine with shared memory to the programmer, so the users are not concerned with message passing, data and execution distribution, machine failures, and the configuration of the system. The actual execution environment, a network of distributed non-dedicated workstation, is different from the abstract parallel machine, as illustrated in figure 2.1. This abstract execution model separates the actual execution environment of the parallel program from the parallel machine presented to the programmer. In fact, programs written for shared memory parallel machine is expected to be executed on a distributed platform.

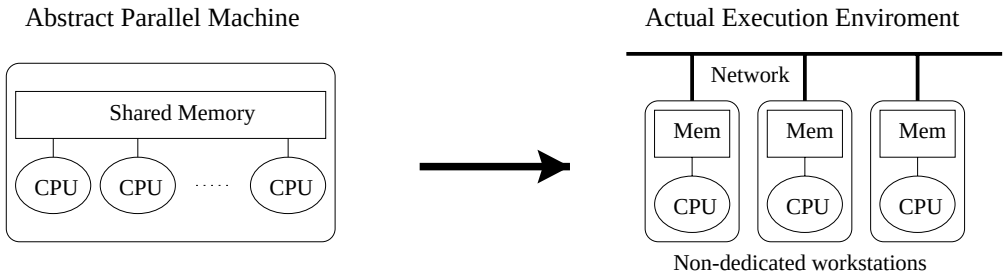


Figure 2.1: Abstract parallel machine versus actual execution environment

We consider a parallel program as a collection of sequential *jobs*. Each job is an abstract execution unit specified by the programmer. *Task* is the physical execution module that executes a single job. A job can be executed multiple times (by several tasks) and produces the correct result, thanks to the idempotent execution strategy described in section 2.4. The benefit of separating jobs and

tasks is that abstract jobs never fail, even though the corresponding tasks are subject to failure.

2.3 Nested Parallelism

The fundamental execution construct is a sequential job that can create a set of concurrently running "sub-jobs" during the execution. A sub-jobs is actually another sequential job, which itself is capable of creating sub-sub-jobs, and so on. This kind of parallel execution mechanism is referred to as *nested parallelism*. Besides being a programming abstraction, nested parallelism is useful for effective management at runtime. For instance, the following criteria can be used to manage jobs dynamically and transparently:

- intermediate computation results
- availability of resources acquired dynamically from a distributed environment

Nested parallelism may also be presented in "parent and child" relationships among concurrently running sequential jobs. At any point of its execution, a sequential job can *initiate* a set of concurrently running jobs, referred to as *children* of the *parent* job that initiates them. In general, a parent job needs the results of its children jobs to resume its execution. Thus, a parent job waits while its children are running.

When the children start running, they "inherit" the state of the shared memory "known" by the parent job since they are sub-jobs of the parent job. During the execution, children jobs only modify their "local copy" of the shared variables. In addition, certain synchronization operation can also take place that allows instant access to synchronized variables. When all children finish execution, their updates are applied to the state of the memory of the parent, thus "reporting" to the parent the results of the computation "assigned" to them. Then, the parent resumes running again. The computation terminates when there are no running or waiting jobs.

2.4 Nested Two-phase Idempotent Execution Strategy

An action is idempotent if the result of multiple execution is the same as being executed once. For example, a function that returns the nth prime number, is idempotent. We use idempotent in a related sense that multiple partial execution of an action can coexist as long as one of the execution finishes.

Two-phase Idempotent Execution Strategy (TIES) [6] allows multiple, possible partial execution of a job to produce the same result as if the job was executed exactly once. TIES divides execution into two phases: the execution phase and the update phase. In the execution phase, a job is executed with its updates to the shared memory saved in an update buffer. There may be several tasks working on

the same job in this phase, however, only one copy of the updates is kept at the end of the job execution. The result of other tasks running the same job, which may be different, is ignored. The shared memory is read-only during this phase and no updates are applied to it. After all jobs finish the execution phase, the update phase begins. During the update phase, the buffered updates created in the execution phase are applied to the shared memory. The shared memory is write-only during this phase. Each phase in TIES is idempotent since the input and output sets are disjoint. The overall execution using TIES is idempotent if the executions of the two phases do not overlap, i.e., barrier synchronization is used in each phase.

The introduction of nested parallelism creates a new challenge to the two-phase idempotent execution. A job may create additional children jobs during the execution phase. The execution of the parent job becomes non-idempotent if these children jobs update shared memory. To solve this problem, we introduce *Nested Two-phase Idempotent Execution Strategy* (NTIES) to handle nested idempotent execution. NTIES works the same way as TIES in that it divides execution into two idempotent phases, however the scope is different. In a non-nested parallel execution, all parallel jobs running at the same instance are siblings within the same parallel block. For nested parallel execution, jobs running at the same time may come from different parallel blocks, so the idempotent strategy applied to each

parallel blocks individually, with barrier synchronization among sibling jobs only. That is, each time a parent job creates a new set of children jobs, the two-phase execution strategy is applied to the set of children jobs.

Nested two-phase execution is not idempotent since the children jobs' update phase is executed within parent job's execution phase, and this update phase writes data to the shared memory. To solve this problem, the children jobs do not update the shared memory pages directly when they enter the update phase. Instead, their updates are stored to the parent's update buffer and treated as part of the parent updates in the execution phase. These updates will be applied to the shared memory after the parent job enters the update phase. The execution of a parent job is idempotent because all shared memory updates of its children jobs are integrated with its execution result without immediate update to the shared memory.

2.5 Prioritized Eager Scheduling

Eager scheduling algorithm [6] is similar to the workstation/job style scheduling algorithms used in PVM [23], MPI [27], and Linda [13]. These algorithms assign jobs that require services to available workstations in the system dynamically. The unique feature of eager scheduling algorithm is that it schedules a job eagerly to multiple available workstations when the system has more available workstations

than jobs. Eager scheduling algorithm and TIES work together to allow a system to mask machine failure and slowdown gracefully. Transient and slow machines can be used effectively, and the system can integrate newly available machines into the running environment easily.

The eager scheduling algorithm is not well suited for nested parallel programs because it does not handle the dynamic job creation very well. Non-nested parallel execution has only one level of parallelism. At the beginning of a parallel block, the number of jobs to be created in the entire system is known and fixed until the end of the parallel block. Available workstations are divided evenly among the jobs, with each machine running one task at a time. For example, if there are two jobs and six available workstations, then each job will have three tasks working for it. In contrast, nested parallel execution allows additional children jobs to be created during the execution of a parallel block so the number of jobs in the system varies. New jobs are created and destroyed dynamically when some children jobs create additional nested parallel jobs.

To handle the scheduling properly, we include the notion of *priority* for each task in our system. Priority is used to decide which task to create or suspend during execution. Tasks with lowest priority will be suspended when the system runs out of resources, or when a task with higher priority enters the system. Assigning priority to tasks allows the scheduling algorithm to use available workstations

more effectively. When a new job is added to the system, the scheduling algorithm creates a new task with highest priority and assigns it to the workstation running the lowest priority task. The workstation suspends the low priority task and start running the higher priority task. Additional tasks executing the same job are created one-by-one with decreasing priority. That is, the first task executing a job in the system is given to priority one. The second task is created with priority two, so on so forth. The scheduling algorithm continues to create new tasks and suspend low priority tasks aggressively until the newly created task has lower priority than all the running tasks.

To illustrate the scheduling algorithm, let's look at an example. In an underload system with four workstations and only one job, the scheduling algorithm creates four tasks executing the job, one for each workstation. These tasks are assigned with priority one to four as described above. Later on when another job is introduced into the system, two tasks executing the new job will be created. The old tasks with priority three and four will be suspended to yield to the new tasks with priority one and two. As the result, the system divides the available machines evenly between the two jobs.

2.6 Synchronization

Synchronization in our system is limited to variables associated with the synchronization instead of the entire shared memory. That is, only variables associated with synchronization are up to date after a synchronization request. Synchronizing the entire share memory requires taking a snapshot of the system, which is very expensive and especially difficult in our system since updates to the shared memory are not applied to the shared memory directly.

The two-phase idempotent execution defers the updates to the shared memory during the execution to the update phase. However, synchronized operations require the synchronized variables to be updated immediately so that different jobs can communicate with each other. That is, even if both jobs are in execution phase, the updates to the synchronized variables of one job must be made available to the other job after synchronization. In order to preserve the idempotent characteristic of the execution, the system have to maintain several versions of the synchronized variables. In particular, the before and after images of the synchronized variables are saved for each critical section. Further execution of the same job will refer to the before image of the synchronized variable to maintain the correct execution.

The separation of abstract jobs and physical tasks in our system introduces additional problems with synchronization. Mutual exclusion and synchronization semantics are defined in terms of jobs at the user level, but there may be several

tasks executing a job in the system. Hence, we have to redefine the synchronization operation for tasks. For example, if a task gets the lock, all tasks executing the same job own the lock. Tasks executing the same job will be able to enter the critical section without being blocked. The lock is considered released after any one of these task release the lock. Thus, other jobs may acquire the same lock while some of the tasks are still in critical section. We claim this is the correct behavior since the outcome of the critical section is already known at this point; all further execution of the same job can be safely ignored. The detail discussion of synchronization operation in terms of tasks is given in section 4.5.

Despite all the complexities, extending the idempotent execution to allow explicit synchronization introduces desirable properties. The synchronization becomes non-blocking since the abstract jobs will never fail (although the actual running tasks may). The system will never be blocked due to failures of jobs that hold locks, as instances of jobs will be initialized automatically to mask the slow or failed tasks.

2.7 Randomized Computing

Randomization is a very powerful algorithmic technique. Unfortunately, randomness cause many problems for idempotent executions in a nested parallel environment.

Randomized programs can be executed in an idempotent fashion, as long as the system maintains the "exactly once" semantic. Due to the randomness effect, tasks executing the same job may produce different results. The system achieves exactly once semantic by killing all tasks but one that completes.

Nested parallelism increases the difficulty in handling randomness. Children tasks are valid only if the parent task that spawns them is non-faulty since each parent task may be in different state when it reached parallel block. The children tasks will have to be killed once their parent task failed since the computation result may not be repeatable.

The situation is even worse when the number of jobs to create depends on the current state of execution. For example, a job may decide to create hundreds of children jobs or no jobs at all depending on some random variables. Furthermore, each of the children jobs may decide to create more jobs randomly. Hence, one execution of the job creates hundreds of jobs while other execution of the same job creates no children jobs at all. The system will perform poorly in such case as hundreds of potentially useless jobs are running, consuming precious resources.

It is also problematic to execute synchronization operations in programs employ randomization. Task failures may cause domino effect and force the system to restart many non-faulty tasks since the execution results may not be repeatable. For example, if task T_2 acquires a lock after task T_1 released it, then T_2 becomes

reliant on T_1. Task T_2 will be abandoned if T_1 fails because different execution will yield different result, so the state of T_1 cannot be reconstructed. This makes the system unreliable since a failure may cause the entire program to restart.

Due to these reasons, the support of the randomization in our system is limited. The only randomized programs allowed are these without nested parallelism and synchronization. The ray-tracing program in our experiments uses random numbers and fits in this category.

Chapter 3

Programmer's Model

To ease the work of application programmers, an abstract execution model of a virtual shared memory parallel machine is provided. This virtual machine supports globally shared memory and has an unlimited number of non-failing sequential processors. These processors do not operate in lockstep, instead, they can be synchronized by a variant of a barrier construct or locking. A programmer *is not concerned* with issues like fault tolerance, load balancing, etc. Programs with rich parallel constructs written for such an "ideal" machine could be executed dependably on any unreliable distributed platform.

In this chapter, the programmer's view of the system will be discussed. However, the actual execution environment which is invisible to programmers, is quite different from the programmer's view and will be discussed in detail in Chapter 4.

3.1 Syntax and Semantics

Our formal syntax extends that of Calypso [6] and supports the functionality which is similar to that of the "parallel part" of CC++. (See related work in Section 7). We found it more convenient to use a somewhat different syntax, but this is not of inherent importance.

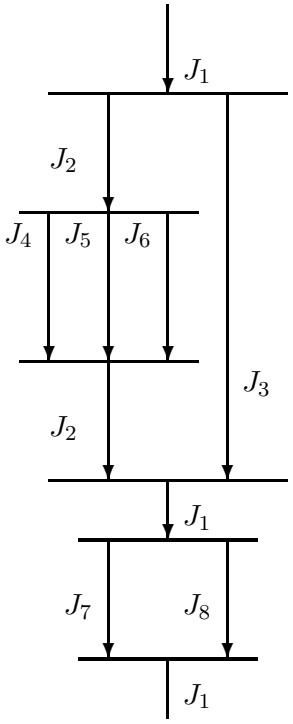
We start with C++ and augment it with new keywords like **parbegin**, **parend**, and **routine** to express parallelism. The keyword **shared** is added to identify the shared variables. Additional keywords are included for synchronization purposes: **sync_t**, **lock**, **unlock**, and **assoc**. We will describe these keywords in turn.

3.1.1 Expressing Parallelism

A compound statement bracketed by **parbegin/parend** declares a set of concurrent sequential jobs. This statement can be invoked "inside" another **parbegin/parend** statement to provide nested parallelism.

Each **parbegin/parend** statement brackets *a sequence* of **routine** statements. Each routine statement is a sequential program fragment resembling a procedure with one positive integer parameter. During the execution, each routine will create several *children jobs*, with their number specified by the parameter. Each such job will have two parameters passed to it: *width*, the number of jobs created by this **routine**; and *id*, the "serial number" among these jobs. The default width is

one, so **routine** and **routine**[1] are equivalent. Id starts with zero and ends with $width - 1$. The statement **parbegin routine**[S_1]; **routine**[S_2]; ...; **routine**[S_n]; **parend** creates j new jobs, where $j = \sum_{i=1}^{n} S_i$. The parent job that reaches the **parbegin/parend** block is suspended after creating the children jobs. The execution resumes from the point immediately following **parend** after all the j children jobs are terminated.




```
 1: main(){  // job J1
 2:   ... // sequential statements
 3:   parbegin
 4:     routine[1] {  // job J2
 5:       int m=3; ... // sequential statements
 6:       parbegin
 7:         routine[m](int w, int i) { //J4, J5, J6
 8:           ... // sequential statements
 9:         }
10:       parend;
11:       ... // sequential statements
12:     }
13:     routine[1] { ... }  // job J3
14:   parend;
15:   ... // sequential statements
16:   parbegin
17:     routine[1] { ... }  // job J7
18:     routine[1] { ... }  // job J8
19:   parend;
20:   ... } // sequential statements
```

Figure 3.1: A sample program employing nested parallelism

For example, consider Fig 3.1. The program on the right defines the jobs shown in the graph on the left. The horizontal bars indicate the **parbegin** or **parend** statement, while the vertical lines indicate the jobs. This is the *programmer's*

view of the execution—the actual execution will be very different, as we describe in Chapter 4.

The execution starts with a single sequential job, J_1. In lines 4 and 13, two concurrent children of J_1, namely J_2 and J_3 each with its own execution code, are initiated. Job J_1 waits while the children jobs are running. In job J_2, additional concurrent jobs are initiated inside its parallel block. At line 7, the **routine**$[m]$ statement initiates three jobs ($m = 3$), with identical execution code, but different parameters are passed to them. Thus, jobs J_4, J_5, and J_6 are passed the value of 3 as the total number of "siblings," and get their serial number as 1, 2, and 3, respectively. After all of J_4, J_5, and J_6 complete at line 10, job J_2 resumes its execution at line 11. Similarly, job J_1 resumes execution at line 15 after its children J_2 and J_3 are completed at line 14. Subsequently, it initiates jobs J_7 and J_8, etc.

The **routine** statement has a parameter, like "m" above, which can be an arbitrary expression computed at runtime. Its value could depend on, for instance, the complexity of the problem that job J_2 needs to accomplish (large problem may benefit from more concurrent jobs to complete), and on the availability of computational resources.

3.1.2 Types of Variables

There are three types of variables that can hold user data, namely *globally shared* variables, *synchronization* variables, and non-shared variables. Variables declared in the program are non-shared variables by default. Non-shared variables are only visible to the job that defines them and cannot be accessed by other jobs.

The shared variables are declared only once in the program via the keyword **shared**:

shared *variable-declaration*

Shared variables can be accessed by all jobs in the system during run-time. However, the updates to the shared variables do not propagate to other jobs until the end of a parallel block. We will describe the precise semantics of the globally shared variables later when the memory coherence model is introduced.

In addition to shared variables, there are also synchronization variables to control the execution sequence and to provide communication between jobs. The synchronization variables are declared using keyword **sync_t**:

sync_t *variable-declaration*

The synchronization variables are used to guard mutually exclusive critical sections. Multiple critical sections guarded by different synchronization variables can proceed in parallel while each synchronization variable allows only one critical section to be executed at any time.

3.1.3 Expressing Synchronization

Synchronization provides another way to control the execution as well as communication between the jobs. In our system, a compound statement bracketed by a pair of **lock/unlock** defines a synchronization block:

lock *synchronization variables*

critical section

unlock

There can be only one synchronization block executing at any point in time for each synchronization variable. Synchronization blocks can be nested but should be done carefully as it may introduce deadlock easily. Parallel statements like **parbegin/parend** inside a synchronization block should be avoided since this introduces a new set of problems. The following questions have to be answered: should the children jobs inherit the lock? What happens when the children jobs try to access the shared variable associated with the synchronization variable? In our system, if there are parallel blocks inside the synchronization block, then all the children jobs can access the shared variables associated with the lock. However, they do not own the lock and they cannot acquire or release the lock.

Each synchronization variable is associated with one or more shared variables. As we will see later in this section, we use *entry consistency* memory coherence model, which requires all shared variables that require synchronization to be as-

sociated with a synchronization variable. Associating shared variables with a synchronization variable is done by using the **assoc()** routine:

assoc(*synchronization variable, shared variables ...*)

The associating shared variable cannot be a pointer, but it can be an array element like `A[0]`. A shared variable can associate with only one synchronization variable. A synchronization variable can associate with many shared variables by invoking the **assoc()** routine multiple times. Once the synchronization variable is used in a critical section, the association between the synchronization variable and the associated shared variables cannot change.

By associating shared variables to synchronization variables, the associated shared variables are always synchronized inside the critical section. The current value of associated shared variables are acquired from the system during the lock operation. Unlock operation writes the updated variables back to the system. Accessing associated shared variables outside the synchronization block is prohibited.

Synchronization blocks with different lock variables can co-exist at the same time since they do not operate on the same shared variables. For example, if there are two different jobs entering synchronization block, one with synchronization variable a and the other with synchronization variable b, then both jobs can enter synchronization blocks since they will access different associated shared variables without any conflict. In contrast, if both jobs want to enter synchronization block

with synchronization variable c, then one of them will be blocked.

3.2 Memory Coherence

A job can have access to the parameters passed to it, its own local variables, and the variables declared as shared. A popular model for memory coherence is the *release consistency model* [14], in which the global shared memory is not "continuously updated," but the updates are applied only at specific, well-defined points in time during the execution. Our memory access model is similar to release consistency in that the shared memory is *acquired* at the beginning of a job execution and *released* at the end. Detailed description of the popular memory consistency models is given in section 7.1.3.

Explicit Synchronization operations use a different memory consistency model, similar to the *entry consistency model* [10]. Shared variables that need to be synchronized have to be specified explicitly. Only shared variables associated with synchronization variables are synchronized inside a critical section.

3.2.1 Shared Memory Access

We think of a job as "working for" its parent and thus being "encompassed" by it. Broadly speaking, the acquire/release operations by each job are done at the beginning and the end of its execution and with respect to the state of the memory

of its parent. Thus, when a job is started, it obtains the state of the shared memory which is known to its parent job. Referring back to Fig. 3.1, job J_4 sees the updates to the shared memory as applied by the (waiting) job J_2, but J_3 does not see these updates.

During the execution of a job, the updates to the shared memory will be stored locally. When all the children jobs complete their execution, the changes they make to the shared memory are then applied to the shared memory as seen by their parent. The parent then resumes its execution and updates to the grandparent (if any) will be passed only after the parent completes. For updates, we have selected the Common Concurrent-Read-Concurrent-Write semantics. This means that shared variables can be read and written by all the sibling jobs at the same time, however all the updates to the shared variable must be the same when they are reported to the parent. Also, the updates to the shared variables are not synchronized between sibling jobs without explicit synchronization operations.

3.2.2 Synchronized Memory Access

Shared memory accesses cannot be used for process communication or synchronization between jobs since the shared variable updates are made available to the parent *after* the job is done. In order to provide immediate data communication, the system allows shared variables to be associated with synchronization variables.

The shared variables that are tied to a synchronization variable are synchronized inside the critical section guarded by the corresponding synchronization variable. Shared variables are synchronized only if they are inside a critical section that is guarded by their associated synchronization variable.

3.3 A Sample Program

Figure 3.2 shows the source code of a parallel quicksort program. Several sequential subroutines used by the program are not shown in the figure. The subroutine *findPivot*() finds the pivot value and the *partition*() routine actually partitions the problem according to the pivot.

To transform a sequential quicksort program into a parallel program in our system, variables to be shared globally are chosen first. Since access to the shared variables may be expensive in a distributed system, the number of shared variables should be kept minimal. In quicksort, the only variable that needs to be shared is the problem/solution array.

By adding `parbegin/parend` around the recursive function calls, the two subproblems are solved in parallel. These jobs are independent since each subproblem takes a portion of the array without overlap. No explicit synchronizations are required.

This example also shows how to pass additional arguments to the children jobs.

```
const int MaxSize = 160000; // problem size
const int BubbleThreshold = 1000; // solve using Bubblesort
shared {
   int A[MaxSize];
   int arg[4];
};

void bubblesort(start, end); // sort the problem using bubble sort
int findPivot(int, int, int); // select a pivot element
int partition(int, int, int); // divide A[] according to pivot
void quicksort(int start, int end)
{
      if (end - start) < BubbleThreshold) {
        bubblesort(start, end); return;
      }
      // partition the list and sort
      int pivotPoint = findPivot(start, end, (start+end)/2);
      int k = partition(start, end, pivotPoint);
      // store function arguments for children in a shared array
      arg[0] = start;
      arg[1] = k - 1;
      arg[2] = k + 1;
      arg[3] = end;

      parbegin
         routine[2](int width, int id) { // run 2 routine in parallel
            quicksort(arg[id*2], arg[id*2 + 1]);
         }
      parend;

      arg[0] = arg[1] = arg[2] = arg[3] = 0; // reset argument array
   }
}

// main program invoke quicksort after filling up the problem array
main() {
   ... // fill up the problem array
   quicksort{0, MaxSize - 1);
}
```

Figure 3.2: A quicksort program

The `routine` function provides the parallelization width and id for each child. To pass arguments to the children, the parent job prepares a shared array with the arguments. The children jobs take the arguments from the array according to their id's. Since the parallelism is nested, this shared argument array can be used in the subsequent calls to grandchildren and all the descendants. The children jobs reset the argument arrays after use.

Chapter 4

System Architecture

We designed and implemented a software execution environment supporting the programming and the abstract parallel execution model described in Chapter 3. As for the underlying platform we chose a network of non-dedicated workstations. To achieve *dependable high performance*, the features of our architecture include: *scalability*, *dynamic load balancing*, and *fault tolerance*. The architecture of our system is introduced next, emphasizing how it supports these goals.

4.1 Execution Strategy

The features will be supported by *dynamically* distributing all the work among the available machines, replicating some of the work as appropriate. In our architecture:

1. no machine will be (greatly) overloaded compared to other machines, thus supporting scalability and load balancing

2. a machine will be able to do work already started by another machine, thus supporting load balancing and fault tolerance.

We hasten to add that, as we see later, for a specific function, high-level scheduling among available machines we have one dedicated coordinating module. However, as we will also see, it has little work to do and therefore it is not a bottleneck. Also, its state can be reconstructed in case of its failure and thus fault tolerance is maintained.

We start with a sketch of the execution of some specific program. The system will view it as a set of *jobs*, with some dynamically maintained subset of them executing at any time.

As described in Chapter 2, thanks to the idempotent execution strategy, a job can have several identical executing *copies* in the system, while maintaining exactly-once semantics. We referred to each of such copy as a *task*.

By extending the parent/child relation among jobs, we can define a parent/child relationship among tasks. We will say that

> T_1 is a parent of T_2 if and only if T_1 is a copy of J_1, T_2 is a copy of J_2, and J_1 is the parent of J_2.

Note, that a task can have more than one parent. For instance, if J_1 is the parent of J_2, T_1 and T_1' are copies of J_1, T_2 is a copy of J_2, then both T_1 and T_1' are parents of T_2.

In the following, it is useful to consult Fig. 3.1 on page 25. Assume first that there is no nested parallelism and there is only one task T_1 executing the "main" job J_1. Assume that T_1 spawns children tasks T_2 and T_3. Both should inherit the state of the shared memory of T_1 when they are spawned. When the children start executing, they do not know the state of the shared memory and will rely on their parent T_1 who has a complete set of shared variables to provide it to them. Whenever a child task touches a shared page for the first time, the system requests and obtains this page from the parent. The child executes on its local copy, and at the end of the computation returns the changes to the parent. The parent accepts the results of the first completed task for each job and then merges all the updates once the execution of all the children is completed. This exploits basic mechanisms employed in running non-nested parallel programs. However, nested parallelism makes the situation more complex than in a non-nested parallelism.

First, in a non-nested parallel program there is only one parent task and only one level of parallelism and thus the (single, system-wide) parent task always has a complete master set of values for all shared of variables. (The parent was the main task, which started with some, maybe system default, values for all shared

variables.) In a nested parallel program, a child task could also be a parent task for other tasks. Task T_2, for instance, may need to supply data to its own child, task T_4, but T_4 may need data not used by T_2. Thus T_2 may not be able to serve the needs of its child directly.

Second, in nested parallel programs multiple copies of a parent task may exist. Suppose that T_1 spawns two copies T_2 and T_2' (of J_2). It is possible that T_2 also spawns copies of its children, that is T_4, T_5, and T_6. T_2 could service the memory requests of T_4. But, another case is possible too. Suppose T_2' also spawns children copies T_4' , T_5', and T_6'. Now, both T_2 and T_2' are parents of all of T_4, T_4', T_5, T_5', T_6, and T_6'. At the time T_2 and T_2' spawn their children, the state of the shared memory is identical in both, as they are running identical jobs. (However, jobs must be deterministic, as we discussed in Section 2.7.) Therefore, each of them can serve any subset of T_4, ..., T_6', enhancing scalability and fault tolerance, as the shared memory needed may be distributed on several machines. This extra flexibility benefits performance too. For instance, if T_2 and T_4 run on the same machine and T_2' and T_5 run on another machine, it may be appropriate for T_2 to serve the memory requests of T_4, and for T_2' to serve the memory requests of T_5.

This richness in structure adds to the complexity of the design but makes the execution more flexible, and increases scalability, load balancing, and fault tolerance.

4.2 Architecture Overview

The system is divided into two parts, the set of user *tasks* to be executed and the *management services* that support the execution of these tasks. The management services include *memory management*, *execution* management, and *lock management*. Similar to the execution of the underlying application program, the memory management and execution management are distributed. In contrast, lock management is centralized to improve performance. The lock management provides synchronization and is discussed in Section 4.5.

Memory management services have *logical structure*, supporting the execution of a specific program, are shown in Fig. 4.1. They are responsible for handling shared memory requests of the tasks while maintaining a coherent view of the distributed shared memory. The memory managers are not bound to machines, instead they are created according to the dynamic evolving nested parallel program structure. That is, there may be zero or many memory managers in one machine, but there is at least one memory manager for each set of children jobs.

Execution management has *physical structure*, supporting the specific set of networked machines participating in the computation. Each *execution manager* is bound to a machine, managing the progress of tasks running on local machine as well as cooperating with other machines to support load balancing and fault tolerance.

In general, a task that was started but not finished, is either *running*, *waiting* (for completion of its children execution), or *suspended* (to allow a task of higher priority to execute, with the suspended task "resurrected," if for instance, its completion is needed to mask faults). While it is waiting, a task becomes a *memory manager* for its children tasks, but in fact no children need to be assigned to it for servicing requests.

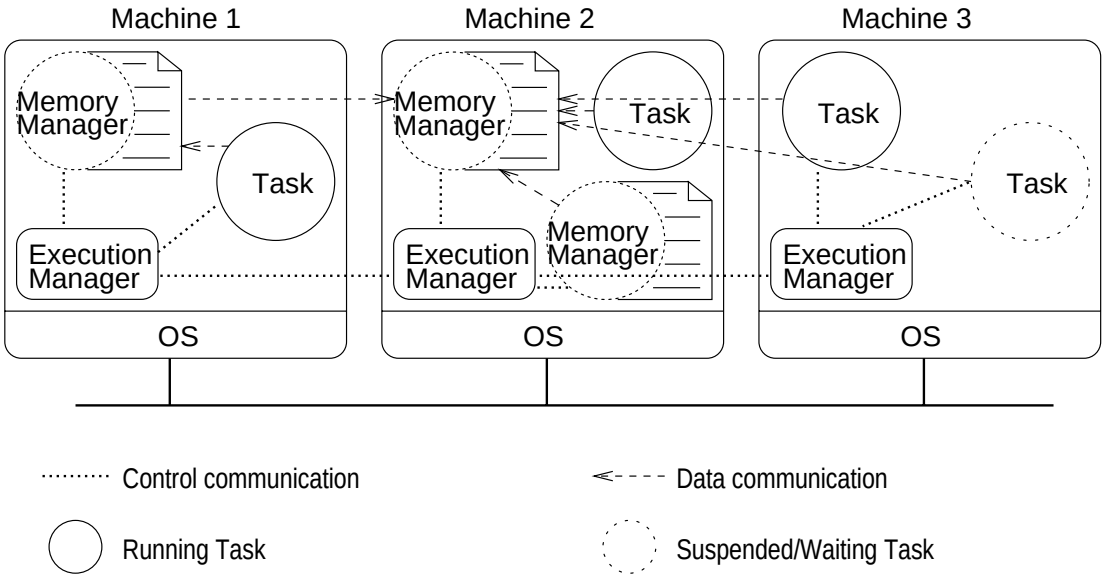


Figure 4.1: Components in the system

Fig. 4.1 shows the relationships between the tasks and the management services. Each circle in the figure represents a user process in the machines. The dotted circles represent tasks that are currently waiting or suspended. Waiting tasks become the memory managers for their children tasks. Each task is associated to exactly one memory manager. There is a process on each machine running as

execution managers. Lock manager is missing from this picture, since there is only one lock manager in the system and it is only associated with tasks that require synchronization services.

4.3 Execution Management

The execution management services provide scheduling and execution controls in the system. Even though they are computationally non-intensive, they do provide a critical service.

There is a local execution management service, the *execution manager*, running on each workstation. The execution manager is responsible for controlling the tasks running on the local machine such as creating new tasks, deleting obsolete tasks, and suspending and resuming tasks.

One of these execution managers, the *coordinating execution manager* or the *coordinator*, maintains the overall information of all execution managers and makes global scheduling decisions based on the information. If the coordinator fails, the global information can be collected from all of the remaining execution managers to form a new coordinator, as long as there is at least one non-faulty workstation.

Our scheduling policy attempts to allocate resources fairly to the tasks so that the computation proceeds fast. At any time, there is at most one task running (as opposed to waiting or suspended) on each machine. In an underloaded system in

which there are more available machines than jobs, more than one copy of a task may be spawned to take the advantage of the underutilized machines. To decide which task should be running and on which machine, we utilize natural heuristics. Each task is assigned a priority—the lower priority, the more likely the task will be suspended in the presence of competing demands. The priority is lowered if there are several tasks executing a specific job, as it is sufficient that only one task completes. For tasks of the same priority, the task that is currently running will not be interrupted by a new one. Besides priority, other aspects of the scheduling strategy are considered in our system. For example, as we will see, we try to schedule the parent and one of its children tasks on the same machine to minimize network traffic.

Consider the example in Fig. 4.2, which presents a possible execution scenario for the program in Fig. 3.1, p. 25 on a network of six available workstations 1 through 6. Each of the machine is running an execution manager before the program starts. In our discussion we comment briefly on issues related to fault tolerance and load balancing.

Snapshot 1. The program starts with a single job J_1 running as task T_1 on Machine 1. (In general there could be several tasks for J_1 also—we do not discuss this here.) Since there is only one machine involved in the execution, the execution manager in machine one is the coordinator.

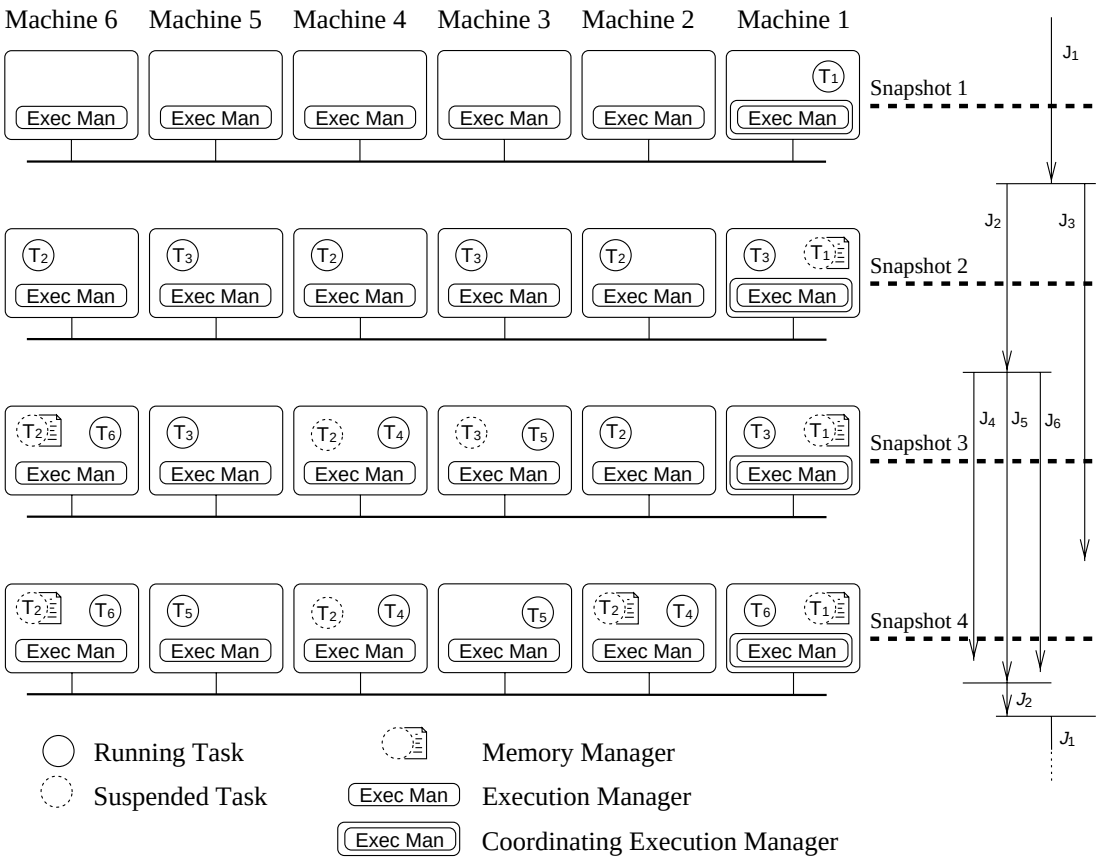


Figure 4.2: Execution snapshots

Snapshot 2. When T_1 reaches the first parallel block, two children jobs are created. At this point, T_1 turns in a memory manager to provide access to shared memory for the all children tasks (which will be discussed in detail later) and is not computationally intensive, so the machine becomes available again to utilize resources effectively. The system spawns three tasks for each children job to take advantage of all the six available machines. A priority is set for each task according to the order they are assigned. The first task of a job has higher priority than later

tasks of the same job. Note that up to two copies of each of T_2 and T_3 can fail without affecting the computation at this point. In fact, if say, all three copies of T_2 fail, the system can spawn another copy of T_2, possibly suspending a copy of T_3—we do not elaborate on this here.

Snapshot 3. One of the T_2 tasks (the one on Machine 6) reaches a parallel block and spawns three new jobs. Again, machine 6 becomes available as T_2 turns into memory manager and a task T_6 is created on this machine. The system spawns task T_4 and T_5 and *suspends* task T_2 on Machine 4 and T_3 on Machine 3 to make room for the new tasks since these tasks have lower priority than the newly created tasks.

Snapshot 4. One of the T_3 tasks (on Machine 1) completes its execution and reports its updates to T_1. The other copies of T_3 are removed from the system as the result. Also at this point the second copy of T_2 on Machine 2 reaches its parallel block and turns into a memory manager. Now there are three free machines. Another copy of T_4, T_5 and T_6 is spawned to take advantage of the three available machines. After one copy of each of T_4, T_5, and T_6 completes, both copies of T_2 end their parallel block and continue executing.

4.4 Memory Management

We now turn our attention to the *memory management* service, which provides distributed shared memory facility to the system.

The memory management service handles shared memory requests to provide a layer of distributed shared memory to the application program. Each task is assigned to a memory manager when it is started. A task is usually assigned to the memory manager that spawns it, but other factors may be taken into consideration like the location of the memory manager. The execution manager assigns memory managers to tasks.

When a task tries to access the shared memory, the access activates an interrupt procedure which sends a shared memory request to the memory manager for the page. As the shared page returns, the data is stored in the local memory and all subsequent access will refer to the cached data. Fig. 4.3 provides an example of the shared memory accessing. In this figure, `read x` and `write z` operate on locally cached shared memory while `read y` operation causes the system to send a shared memory request to the memory manager. Once a task is done, all the dirty pages will be collected and the *updates* to the shared memory will be sent to the memory manager. Memory manager will buffer those updates and apply to the shared memory after all the children of the task are done. A copy of the shared memory updates is also sent to the local execution manager when a task is done.

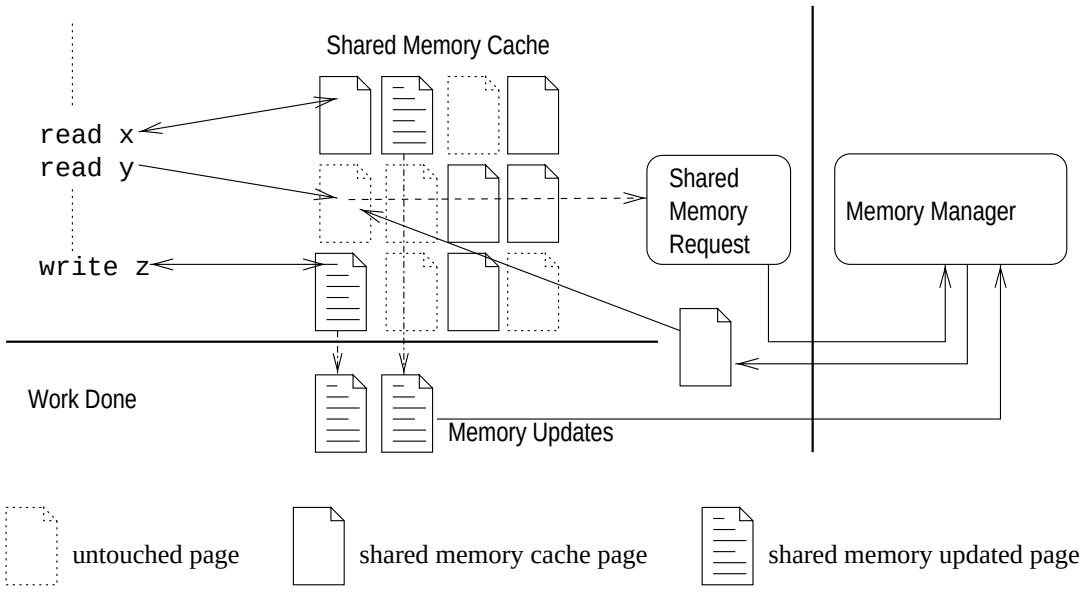


Figure 4.3: Accessing shared memory

This copy of updates is preserved to help late parent tasks, so they do not have to spawn an additional task but use the saved updates directly.

We turn now to the discussion of the dynamically evolving hierarchical memory management structure which reflects the program execution structure. A memory manager is actually a parent task who is waiting for its children tasks to finish. Instead of waiting passively, a parent task switches to a memory management routine when it reaches a parallel block. The behavior of the memory manager is depicted in Fig. 4.4.

A memory manager may serve all its children tasks or none at all, since it is possible to have more than one memory manager per job. In fact, any parent memory manager can satisfy the read shared memory request since all parent

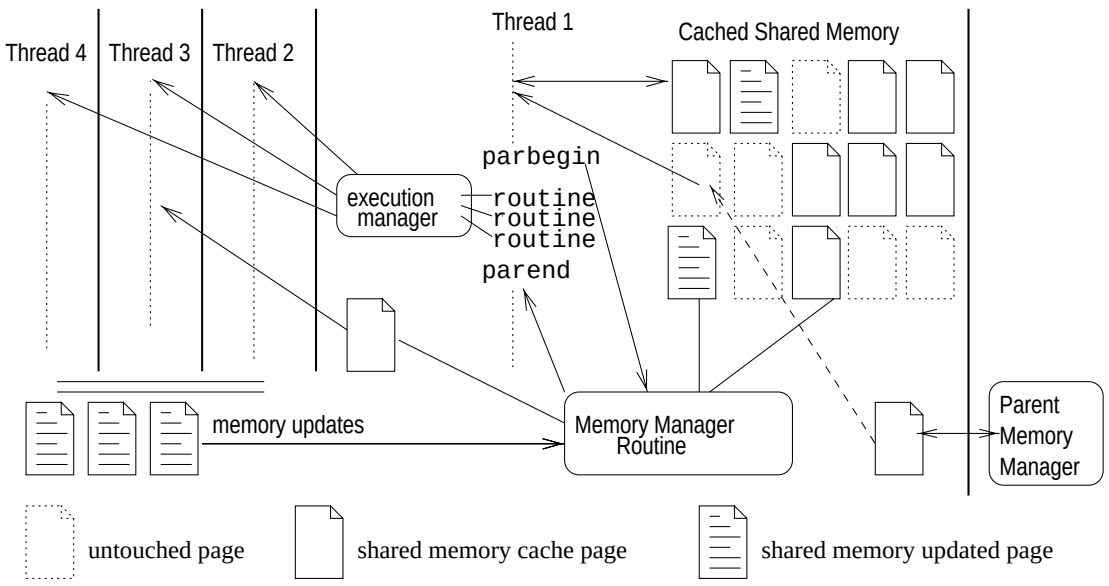


Figure 4.4: Creating child tasks and serving shared memory

memory managers contain the same data. However, all parent memory manager must be notified when updating the shared memory. In Fig. 4.1 on page 40, we see three machines in which three memory managers are running. Dashed lines connect a memory manager to the tasks or other memory managers that it is servicing. One memory manager in Machine 2 serves tasks in Machine 2, Machine 3, and also "being its parent" services the memory manager in Machine 1. Memory manager in Machine 1 serves a task in Machine 1. One memory manager in Machine 2 is not serving any tasks. Let's look at the example in Fig. 4.2 on page 43. At snapshot 4, the second copy of T_2 on Machine 2 reaches its parallel block. It turns into a memory manager. There are two copies of T_2 both running as memory manager at this point, therefore the system can assign either one of the memory managers

to the children tasks T_4. In fact, the children tasks are always assigned to the memory manager residing on the same machine to reduce overhead. Thus, task T_4 on machine 2 will be assigned to the memory manager on the same machine. When one of the T_4 tasks completes, its updates are sent to both of the relevant T_2 managers (in case one of them fails later during the execution). After one copy of each of T_4, T_5, and T_6 completes, the parallel block is ended, the state of the memory of T_2 is updated, and both copies of T_2 can continue executing.

The memory management routine provides memory management functionality using locally cached shared memory. That is, a memory manager always tries to satisfy a shared memory request by going through its local cache first. If the memory requested is not available locally, a shared memory request is sent to the parent task of the memory manager. The chain of requests goes as far up as necessary. Note that the first task reached that has the page, in fact has the correct value for the page.

To summarize the role of memory managers, each memory manager serves the shared memory requests of the children tasks assigned to it. If a child task requests a page that the manager cached locally, it is sent to the child. If the manager does not have the page, the memory manager asks its own memory manager for the page. After the page is received, it is cached and sent to the requesting task. Once a task completes, its updates are returned to its memory manager. When all

children tasks of a parent task are done, the updates are applied and the parent task resumes its execution.

4.5 Synchronization Management

Synchronization in a parallel program serves two purposes, to control the execution sequence and to exchange synchronized data. To provide synchronization functions, there is a synchronization manager in the system. The synchronization manager is responsible for serving synchronization requests and maintaining all synchronization variables and the variables associated with them. For efficiency consideration, the synchronization manager is centralized but requires no dedicated machines. That is, there is only one copy of synchronization manager running in the system in any given time, residing in one of the machines. A new synchronization manager will be created when the synchronization manager fails, using similar fault tolerant techniques employed in the coordinating execution manager. The fault tolerant aspects will be discussed in Chapter 5.

Once a task reaches a critical section, it notifies the execution manager with a synchronization request. The execution manager knows the location of the synchronization manager and forwards the request. On a scenario where the synchronization request is granted, the synchronization manager will send back the synchronized data that is associated with the synchronization variable. The data is

cached locally for the subsequent reference inside the critical section. When the execution reaches the end of the critical section, all the updated variables associated with the synchronization variable are sent back to synchronization manager.

If the lock is not available at the time of the request, the execution manager will *suspend* the task and move on to other available tasks. Once the lock becomes available, the execution manager holding the suspended task is notified. The execution manager will *resurrect* the task and schedule it for execution as the system resource becomes available.

Fig. 4.5 shows a possible execution scenario for synchronization. Here we only look at a portion of the system, say machines 1 through 4, running jobs 6 to 9 in parallel in the middle of some program execution. These jobs may be created by a single parent job, or they may be created by different jobs at different time – we do not elaborate on this since it makes no difference from the synchronization point of view. The coordinator and memory managers are left out in this description to simplify the example.

Snapshot 1. Task T_6 executing job J_6 reaches a critical section. T_6 sends a synchronization request to the local execution manager, which forwards the request to synchronization manager on Machine 1. The synchronization manager grants the request since no one else is holding the lock. It returns the values of shared variables a, b, c that are associated with synchronization variable X. As task T_6

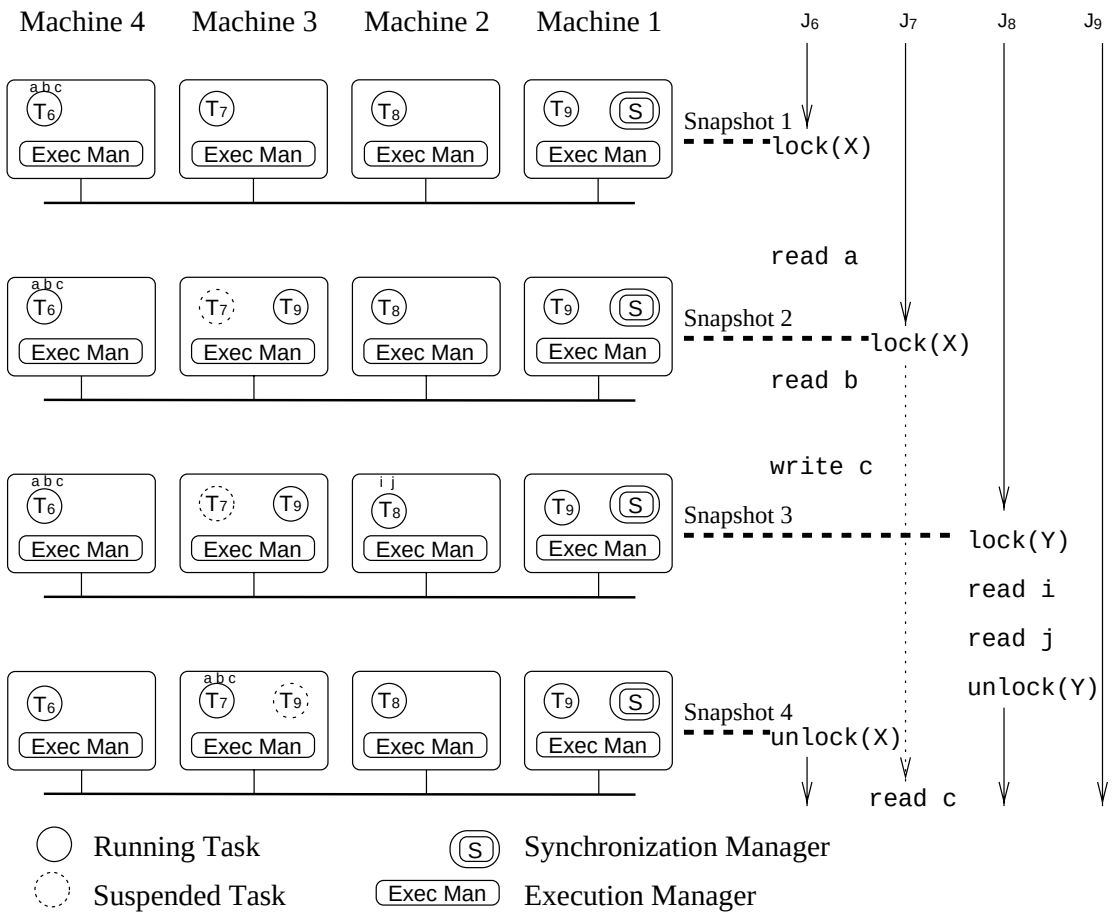


Figure 4.5: Synchronization snapshots

enters the critical section, it references to the local copy of variables a, b, c.

Snapshot 2. As task T_7 executing job J_7 reaches the critical section, it sends to the execution manager a synchronization request for lock X. The execution manager forwards the request to synchronization manager, and gets a *blocked* message as a result, since the lock is owned by T_6 at this point. The execution manager suspends task T_7 and asks the coordinator (not shown) for another task to execute.

The coordinator "happens" to schedule another copy of T_9 on machine 3.

Snapshot 3. Another task, T_8, executing job J_8, reaches a critical section. It requires the lock Y which is available at the moment. The synchronization manager grants the request and returns the data of shared variables i, j which are associated with the synchronization variable Y. Sometime later task T_8 finishes the execution of the critical section and releases the lock. The updated variables are sent back to synchronization manager.

Snapshot 4. Task T_6 reaches the end of the critical section. It notifies the synchronization manager with the updated shared variables. Lock X becomes available again and the synchronization manager sends a notice to the execution manager of T_7. The execution manager decides to resurrect T_7 and puts T_9 to sleep since T_7 has higher priority. Now T_7 resumes its execution inside the critical section.

A synchronization block guarded by lock/unlock statements ensures that two different jobs cannot enter the critical section at the same time. However, a job can be executed by several tasks in our system. Thus, a lock is owned by a group of tasks executing the same job, while tasks executing other jobs are excluded. In order to refer to individual requests, we number all synchronization requests a job invoked successfully during the execution. When a task T_1 executing job J_1 enters a critical section after a successful request, all other synchronization requests from

other job are blocked. Another task T_2 executing the same job can enter the critical section without blocking when it makes the same request. Once task T_1 finishes and leaves the critical section, the lock is released. At this point, other jobs may request the lock and enter the critical section successfully even though task T_2 may still be inside the critical section. Our system property requires that T_1 and T_2 behave the same, as discussed in Section 2.7. Hence, the computation results of T_2 inside the critical section are ignored.

Tasks and synchronization requests have the following relationship:

request *Each synchronization request of a job occurs logically once, while multiple copies of the same request may be generated by several tasks executing the job.*

Even though several tasks executing the same job makes the same request, the system treats them as one request which occurred when the first request is received.

acquire *If a task successfully acquires a lock in its lock request i, all tasks executing the same job are considered having acquired the lock for lock request i.*

Once a task acquires a lock, subsequent requests by the other tasks executing the same job will be granted immediately. For example, suppose task T_1 and T_2 executing job J_1 are running on different machines. When T_1 successfully acquires a lock in its ith synchronization request, all tasks executing job J_1 are considered having acquired the lock for the ith synchronization request at

this point. When task T_2 makes its ith synchronization request, the request will be granted regardless the current state of the lock. In fact, T_1 may have already finished the execution of the critical section and released the lock. Thus, the content of the shared variables associated with the synchronization variable may have been modified by now. In order to allow all tasks executing the same job to see the same set of value, the system remembers the state of the variables before the modification.

release *When a task releases a lock for lock request i, the lock is considered released immediately (for lock request i) regardless the state of the other tasks of the same job.*

A task may think it is holding the lock while other tasks of the same job have released the lock. At this point, other jobs may request the lock and enter into the critical section. Mutual exclusion is guaranteed by maintaining versions of the variables and ignoring the late updates.

Our synchronization manager uses *request sequence* and *variable versioning* to support the behavior described above. Request sequence is a list of synchronization requests that maintain the history of invocations. Each synchronization request can be uniquely identified by the job and request numbers. The request sequence is essential to ensure that multiple tasks of a job will all reference to the same invocation for the same request. For example, task T_1 may have reached synchro-

nization request i while another task T_2, executing the same job, is just started. All the subsequent synchronization requests $1, ..., i$ for task T_2 will refer to the history. Variable versioning is used keep multiple versions of shared variables that are updated in different synchronization requests. The correct version of variables are returned according to the history in the request sequence. New version and new entry in the request sequence are created when a new request is received. Late updates to the variables are ignored for the repeated request.

Starvation problem

There are programs that always terminate if the execution platform has an unbounded number of machines and never fail (we will call this an ideal platform), but may not finish while running on a finite number of realistic machines in our system.

Consider the program in Fig. 4.6 where job J_1 waits for the associated shared variable m to be set to zero in a while loop, and job J_2 sets m to zero. Reading and writing m is guarded by a pair of lock/unlock operations so the variable is up-to-date. This program will never terminate in an execution environment where there is only one machine, and job J_1 is executed first. Job J_1 will be busy waiting for m to be set to zero in a loop and job J_2 will starve. This is due to the fact that our system executes at most one task per machine at any time, and does not

```
...  // beginning of the program skipped
sync_t a;
shared int m=1;
assoc(a, m); // associate shared memory m with a

parbegin
  routine { // job J1
    int n=1;
    while (n) {
      lock(a); n=m; unlock(a);
    }
  }

  routine { // job J2
    lock(a); m=0; unlock(a);
  }
parend
... // remaining of the program skipped
```

Figure 4.6: A program that could cause starvation

preempt running tasks. Running multiple preemptive tasks on a machine at the same time could hurt the overall performance in our system because we schedule redundant tasks on available machines. Therefore, if we run two tasks on one machine at the same time in a system with two machine and two jobs, the other machine may do the same. The system could take twice the time to finish since it runs twice as many tasks at the same time.

This problem could also occur in a system with multiple available machines. For example, if there are four available machines and five jobs, J_1 to J_5, in the system. Let J_1 through J_4 run the busy waiting loop described above, each with

a different lock variable. Job J_5 resets the variables associated with these lock variables to zero. If the four available machines happen to run jobs J_1 through J_4, then no one will run job J_5 and the program will not terminate. It is also possible that there are five available machines initially, but the one that executes J_5 fails; and then the same problem occurs.

We will only consider programs that terminate on an ideal platform in the following discussion. We will say that an execution of a program on a realistic platform reached a starvation point if the execution will never terminate. We show how to modify the system so that no execution will ever reach a starvation point. We detect "suspect" tasks that, without intervention, it is possible that a system has reached a starvation point. The system suspends the task and runs other tasks. To ensure that tasks will not starve, we lower the priority of that task.

Following is an example of how the system prevents the execution from reaching a starvation point on a single machine. For now let us assume the system can detect suspect tasks that may cause the system to reach a starvation point. We consider again the program in Fig. 4.6. Suppose a task t_1 is created and executed first on this machine. The system finds that t_1 is suspect, so it suspends t_1 and lower its priority. As the result, the other task t_2 starts to run. Task t_2 will either finish its execution or will enter busy waiting loop that cause others to starve. If task t_2 finishes its execution, other tasks of equal priority will start and eventually all tasks

will have a chance to run. It is also possible that task t_2 may cause the system to reach a starvation point. In that case, the system suspends and lowers the priority of t_2 as it did to t_1, and other tasks will be able to make progress. When all tasks of higher priority have either finished or suspended, task t_1 resumes. If t_1 is still suspect when it resumes, the system will suspend it again and lower its priority even further. Eventually t_1 becomes a task with lowest priority in the system. At this point, the condition t_1 is waiting for should be satisfied since all other tasks have been executed. Hence, the system will progress and all tasks will be able to execute without starvation.

The situation is more complicated for a system with multiple available machines. It is possible that multiple copies of job J_1 in Fig 4.6 are running on several machines, while all machines running copies of job J_2 have failed. The same solution, suspending and lowering priority of the suspect tasks, works for this situation as well. Copies of J_1 will be suspended with lower priority and eventually copies of J_2 will be executed.

Detecting suspect tasks that may cause the system to reach a starvation point is the key to the solution. We employ a simple detecting scheme that makes tasks suspect after they make a large number of lock requests. This will guarantee that all tasks that will cause the system to reach a starvation point are detected. False alarms will not impact to the correctness of the execution, but will increase the

system overhead. However, the overhead is small comparing with the time required to satisfy a large number of lock requests.

Chapter 5

Fault-tolerant Computing

One of the goals of our system is to provide a fault-tolerant computing environment transparent to the users. In our system design, we paid special attention to fault-tolerant aspects of all modules. To achieve highly dependable computing, the ultimate goal is to handle machine failures gracefully with no single point of failure.

We will first describe the types of failures tolerated in our system and then discuss the fault-tolerant aspect of each module in turn. The fault-tolerant features of all the components except centralized managers are implemented in the current version.

5.1 Types of Failures

In this section, we state our assumptions about the types of failure that can be tolerated by the system. Failures can be categorized in many ways. Based on to the behavior of the faulty components, Cristian et al. [18] classifies failures into four categories:

Crash fault causes a component to halt or lose its internal state.

Omission fault causes a component not to respond to some inputs.

Timing fault causes a component to respond either too early or too late.

Byzantine fault causes a component to behave in a totally arbitrary manner. A component could act maliciously if a Byzantine fault occurs.

Various components could fail in a system, including processors, memories, storage devices, and networking equipments. In a distributed environment, component failures can be categorized as *node failure* or *communication failure*. Node failures include arbitrary component failures within a machine that cause the machine to behave incorrectly. Communication failures include failures caused by the networking components like corrupted messages and lost connections.

Node Failures

Our system tolerates crash faults of up to $n-1$ machines from n given machines. A failed machine may recover and restart from a predefined state. Byzantine faults are not tolerated in our system; however, it is possible to detect arbitrary faults using multiple processors as described in [52].

Omission and timing faults are more relevant to communication failures.

Communication Failures

Our system tolerates crash faults in networking communication system by assuming that the message is either delivered correctly or is not delivered at all. Omission and timing faults are detected using time-out, which cause the originating task to re-send the messages. Byzantine faults are not tolerated.

Communication failures may lead to network partitioning where the network failed in a manner that the remaining machines are partitioned into *groups*. Machines in each group can communicate with each other, but cannot communicate with the machines in other groups. When network partitioning occurs, a group of machines lose contact with machines in other groups; so other machines in other partitions become unavailable to them. In our system, machines in each partition will attempt to recover by creating new coordinators and synchronization managers for the partition if none available. Therefore, each group of machines become a

self-sustained system if possible. The system running on a partitioned network turns into several independent systems, all running the same computation.

5.2 Fault-tolerant User Jobs

Execution of the user jobs is always fail free due to the separation of abstract jobs and physical tasks as described in Section 2.2. Failure of tasks are masked by running additional tasks executing the same job. However, starting additional tasks must be done carefully to avoid unnecessary overhead in a nested parallel environment. A job J_1 may create many children jobs during its execution, and some of these children jobs may have already finished before additional tasks executing J_1 started. Restarting a job should not cause its children jobs to be restarted or a *chain reaction* may occur since each child job is capable of creating more jobs, and restarting children jobs may cause all the descendant jobs to be restarted. Our system avoids this chain reaction by associating the restarted jobs with their children jobs that are running. For children jobs that are already done, the execution results (shared memory updates) of the children tasks are stored in the local machine they ran. Tasks executing a restarted job (or late tasks) use these results instead of starting another copy of the children jobs.

Task failures occurring while children tasks are running present another kind of problems. During the children tasks execution, the parent task becomes memory

manager which serves shared memory for its children tasks. In the next section, we discuss how to mask memory manager failures.

5.3 Fault-tolerant Memory Manager

When a memory manager fails, its children tasks will be blocked when they request memory services provided by the failed memory manager. If there are multiple copies of the same memory manager running in the system, the children tasks redirect all their memory requests to other copies of the memory manager that are available. However, if no memory managers for the tasks are available, these children tasks can not continue their execution.

The fault-tolerant policy for the memory managers is similar to tasks, which is to start additional copies of the job. After the task has been restarted and reached the same parallel block where it failed, it became the memory manager and all its blocked children tasks can resume their execution.

Since children tasks may themselves be parents of their children tasks, a failed memory manager may cause all its descendants to be blocked. Thus, a memory manager may be unable to respond to memory requests due to the ancestor failures instead of its own. The system must restart only the failed memory manager but not all related memory managers.

In our implementation, we construct an *orphan list* that keeps track of blocked

tasks due to their ancestor memory manager failures. When a memory manager is revived, all its blocked descendants will become available according to the orphan list. To identify the memory manager that failed, the highest level of ancestor that failed in the orphan list is reported.

5.4 Fault-tolerant Coordinator

Coordinating execution manager (coordinator) has a centralized function of scheduling tasks globally among all machines. Failures of centralized managers usually lead to the failure of the entire system. In order to handle machine failures gracefully, centralized manager must tolerate failures so the system can continue to make progress with available machines.

The basic idea for tolerating centralized manager failures is to keep enough information in each machine so that the system can reconstruct the state of the manager when it fails. However, complete replication of the manager's data on each machine is expensive since storing and synchronizing the data across the network increases the traffic and storages. To minimize the data stored in each machine, the following scenario is examined. When there is only one machine left in the system, this machine have to re-execute all tasks that failed. However, it should not re-execute any of the tasks done locally. Thus, the machine stores enough information to validate its local computation, so no redundant computation is required upon

failure. If more than one machine is available, then information stored in available machines are collected to reconstruct a global image of the state of the system if possible.

A fault-tolerant coordinator must be able to run on any available machines since the coordinator is an execution manager with global information, we can promote any available execution manager to replace a faulty coordinator. In order for the system to continue making progress, this new coordinator must have global information of all the tasks and all available execution managers. This global information can be reconstructed by collecting data from all available machines. Local data stored in each machine includes the execution history, the tasks created, and the computation results of executed tasks. This information is enough to validate the local computation so that the tasks running locally do not need to be re-executed if the coordinator fails. It may still be necessary to restart the tasks running on the failed machines. For example, in a system with two machines and the works are distributed evenly among the two, our system can guarantee that no more than half of the work is lost if any one of the two machine fails. The system can recover from such failure by restarting some of the failed tasks, which is no more than half of the total work.

Upon coordinator failure, some execution managers will detect the failure as the requests to the coordinator get rejected. These managers will send out vot-

ing messages to execution managers on all machines to choose a new coordinator. The voting mechanism can be implemented using an algorithm similar to the three-phase atomic broadcasting algorithm [11] to determine the ordering of the messages. The first execution manager that sends out the voting message becomes the new coordinator.

Once the new coordinator is elected, all available execution managers will send their local copy of *progress tables* to the new coordinator. The progress table contains a list of tasks executed in the system, and it includes the progress of each task, execution history which contains tasks execution on each machines, parent/child relationship, memory managers for each task, and sometimes the execution result of a task. Local progress table and global progress table have identical format except the local one only contains the tasks executed locally, while the global progress table have all the tasks executed in the system. The new coordinator reconstructs the global progress table and a list of available managers according to the information collected. The new global progress table may not be complete, as some of the tasks running on the faulty machines may be missing. Restarting the appropriate tasks (when necessary) will generate the missing data.

5.5 Fault-tolerant Synchronization Manager

Synchronization manager provides a centralized service, which faces the same problem as the coordinator when failed. To tolerate the synchronization manager failure, we employ a similar method used in Coordinator. That is, to elect a new synchronization manager upon failure, and to reconstruct the state of the new manager with information available at hand.

In order for this strategy to work, the information stored in the synchronization manager must be replicated and distributed through out the system. However, different from coordinator, there is no local synchronization manager on each machine. The system has to rely on the execution manager resides on each machine to store extra data. Furthermore, a complete history of synchronization requests with correct order must be remembered since the exact order of synchronization requests must be maintained for the program to run correctly. Data collection and management for synchronization manager are also more complex than the case of the coordinator.

To manage the synchronization data efficiently, each machine stores only enough data sufficient to continue making progress even if all other machines fail. Minimizing the data stored on each machine is essential since the time and bandwidth required for moving data across the network depends on the volume of the data. Broadcasting the information every time synchronization manager

receives a request can be very costly.

Each machine maintains a complete synchronization invocation history and a version of shared variables associated with each synchronization request invoked locally. The synchronization sequence maintained in each machine is complete but may not be the latest, since we are only interested in history upon failure. Any synchronization invocation beyond local task's latest synchronization request can be ignored since it will not affect local computation.

Storing versions of shared variable associated with each synchronization request is necessary for the system to repeat the execution of the tasks inside the synchronization sequence. For example, suppose task t_1 invoked `lock(`x`)` before task t_2 makes the same request. When t_2 starts, invoking `lock(`x`)` will return the associated variables that t_1 modified. Without them, restarting a failed task may cause non-faulty tasks to restart and create domino effects since the execution of a synchronization request relys on the result of previous invocation. Thus, the synchronization manager sends the latest synchronization sequence and the current version of the associated variables to the requesting machine upon a successful synchronization request.

The system gathers the synchronization data on local machines upon failure, including the synchronization invocation sequence and versions of the associated variables. The latest synchronization invocation sequence is used by the new man-

ager, together with the versions of associated variables collected from the available machines. Some of the versions of the associated variables may be missing since each machine only maintains the versions that is related the its synchronization requests. These missing states can be generated, if necessary, by restarting the invoking tasks again. Note that these re-executed tasks are either failed tasks or completed tasks executed on a failed machine. Thus, it is possible to cause a completed task to be re-executed when restarting a faulty job. However, tasks executed on non-faulty machines will not be restarted.

Chapter 6

Experiment Results

The goals of our experiments are to investigate the performance characteristics of our system, to compare the performance of our system with other systems, and to measure the performance benefit of a nested parallel program over a non-nested one.

A number of parallel programs have been implemented in our system. Furthermore, programs developed in Calypso can be executed in our system with little or no modification since the programming syntax and semantics of our system are very close to Calypso system as described in Section 7.2.3. This enlarged the number and variety of programs available to our system. An important task of the experiments is to examine how much additional performance penalty our system incurs compared with Calypso.

Granularity of a problem exerts a big influence on the performance of our system. We will discuss the effects of granularity in detail in the ray trace experiments in Section 6.2.

Quicksort is a recursive sorting algorithm that is well suited for nested parallelism. We created two version of the parallel program to measure the performance benefits of nested parallelism. The experiment is described in Section 6.3.

Our experiments were conducted on 8 identical machines, each with 200 MHz Pentium Pro processor and 64 megabytes of memory. These machines are connected with 100 Mbps Ethernet in an isolated local area network. All time measurements are done using "wall clock," which calculates the actual time using `gettimeofday()` system calls instead of CPU utilization times. The elapsed time does not include the time required to set up our system on each machine (i.e. `rsh()` calls to create execution manager), but it does include the start up time for parallel tasks. Speedup of a parallel program is calculated by comparing the result of a parallel program with the sequential version of the algorithm, and not by comparing it with a parallel program running on a single machine.

6.1 Performance Characteristics

The first set of experiments measures the overhead of the system by measuring the time required to create a parallel job, to access a remote shared memory page,

and to perform synchronized operations.

Task Creation

To calculate the cost of task creation, the first thing we did is to create jobs with no operation. However, the underlying operating system optimized the `fork()` system call so that creating a dummy process only adds an entry to the process table without actually allocating and copying the resources, so the job creation time cannot be measured correctly. Therefore, we measure the job execution time where a job reads and updates a shared memory location. The overhead is measured by creating this job 1000 times on single machine, using a `for` loop which contains a single job inside a parallel block. It takes about 40 ms to process a job, which includes suspending the parent task, scheduling and forking a new child process, reading/writing a shared memory location, and resuming the parent task.

Creating 1000 jobs sequentially, as described above, is more expensive than creating 1000 parallel jobs all at once (i.e. in one `routine[1000]` statement), since the cost of suspending and resuming the parent tasks is repeated 1000 times in the sequential case. By comparing sequential and parallel job creation, we can better understand the overhead of processing a parallel block. Creating 1000 jobs in parallel takes about 20 seconds, about half the time compared with sequential job creation. This shows that processing parallel block alone, without considering

user jobs, takes less than 20 ms. This is calculated by subtracting the time required to process one parallel blocks from the time required to process 1000 parallel block, divide by 999. The actual cost for processing each parallel block is lower since running many jobs in parallel better utilizes the available resource in the system than running them sequentially. Processing a parallel block includes suspending/resuming parent tasks, creating new progress table entries, and applying children updates.

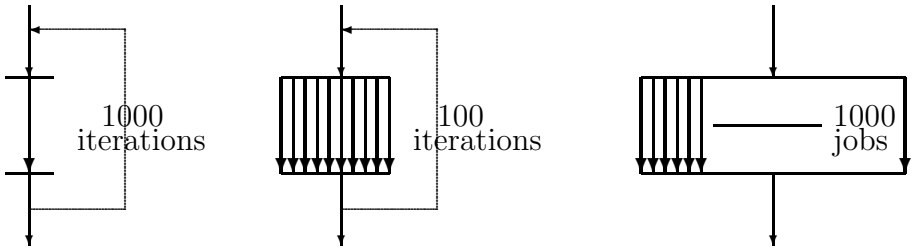


Figure 6.1: Create 1000 jobs using different number of parallel blocks

In addition to the two extreme cases, we also perform experiments with a mixture of parallel jobs and parallel blocks, as shown in Fig. 6.1. The result is shown in Fig. 6.2. In this experiment, the cost for creating a parallel job averaging 20 ms, and the cost for processing a parallel block is also around 20 ms. The overall execution time is equal to the cost of creating all parallel jobs plus the cost of processing all parallel blocks. For example, the overhead of creating 1000 jobs in 100 parallel blocks is 2 s more than creating 1000 parallel jobs in one parallel

block, since additional (100 × 0.02) s are used to process the 100 parallel blocks.

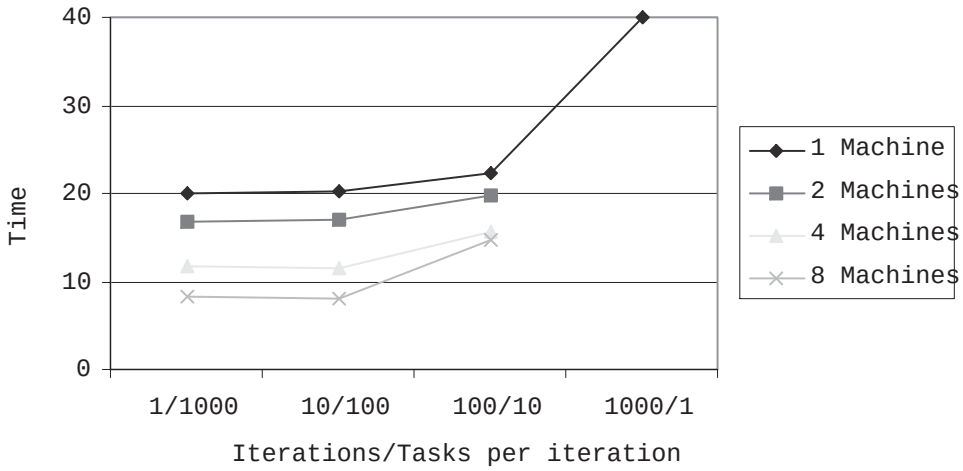


Figure 6.2: Time to create 1000 jobs

The network latency needs to be measured when multiple machines are used in the experiments. We run the same experiment on 2, 4, and 8 machines to see the speedup as functions of number of machines. In Fig. 6.2, we see that the speed up is not very good since the granularity of the task is too small. In fact, since the size of the tasks is so small, the coordinator machine has a 4 to 1 advantage over other machines. That is, since the scheduling and memory requests are served locally, the coordinator can finish roughly four tasks while the other machines get their assignments over the network, request one remote shared page, and return the updated shared memory.

The cost of nested parallelism is also measured in a similar fashion. At first we create 1000 nested tasks, with a task creating one child task and the child task creating a grandchild task, and so on. The system runs out of resources as the result since 1000 nested tasks create 999 memory managers, which creates thousands of sockets. After reducing the number of tasks to 100, we found the cost to create a nested parallel task is about 109 ms. This cost includes the 40 ms task creation time we measured previously and the additional time required for a task to become a memory manager and to create a listening socket.

Shared Memory Access

To measure the overhead of shared memory access, we create a job that access 1000 shared memory pages. The time for accessing 1000 pages is 576 ms when both parent and children jobs are on the same machine. This cost includes trapping memory accesses, sending and receiving messages through the sockets, and handling memory requests. When parent and child jobs are on different machine, the cost increases to 0.792 ms per page due to the network latency.

Synchronization Operation

The overhead of processing critical sections is measured by using methods similar to measuring memory access. A job that loops through a critical section one thousand times is created. The critical section contains a single statement that increases the

associated variable. It takes an average of 1.3ms to go through each critical section, which includes two synchronization operations: a lock and an unlock.

It is more expensive to process synchronization operations when several tasks compete to enter critical sections guarded by the same lock. In the next experiment, we use two machines running two tasks where each task runs through 1000 critical sections, all guarded by the same lock. The executions of the critical sections are interleaved between the two tasks since the synchronization manager grants the lock in the order the requests are received. For example, when task T_1 enters the critical section, task T_2 waits for the lock. When T_1 leaves the critical section, T_2 enters the critical section and T_1 begins to wait for the lock.

The tasks waiting for the lock are suspended in our system, to allow others to utilize the resource. When a task is waiting for a lock, the system schedules and executes other available tasks. As the result, each machine runs a copy of both tasks in the experiment with two tasks and two machines, so we have a total of four tasks running on the two machines. Synchronization manager detects duplicate lock requests and handles them properly.

Each task takes approximately 7.0s to finish, which is much longer than the time required by a single task. Since the time spent outside critical section is so short, the tasks always have to wait when they try to enter a critical section. Extra messages for suspending and resuming the waiting tasks and context switching

between tasks also increase the overhead.

Adding more machines does not increase the speed, since the critical sections are executed in sequence. Furthermore, duplicate requests increase when machines are added to the system. In our experiment with four machines and two tasks, the number of total lock requests increase to nearly 8000. As the number of machines doubles from the previous experiment, the number of requests doubles also. Various mechanisms have been employed in our system to reduce the impact of duplicate requests. For example, the unlock operation is ignored when other tasks executing the same job already released the lock at the time of the lock operation. The time required to finish all tasks is 8.8s for four machines and two tasks.

6.2 Ray Trace

Ray trace is an application that renders three-dimensional scenes. The program simulates the reflection and diffusion of light by shooting light rays from the view point into the scene. The color and brightness of each pixel is calculated according to the trace of the rays.

Parallelization of ray trace program is done by dividing the pixels in the image into sets and trace these sets in parallel. The granularity of the problem is controlled by number of pixels in each set. These sets of pixels are independent from

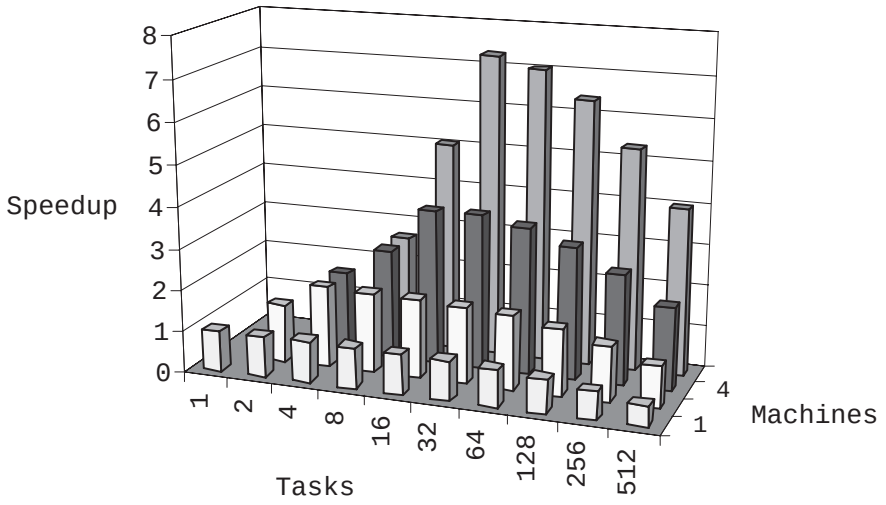
each other and can be executed in isolation without synchronizations. The ray trace program implemented in our system spawns all the sets at once. Therefore, it contains only one parallel block with no nested parallelism.

The ray trace program we used in this experiment is a parallel version of the sequential ray trace program developed by George Kyriazis at PRI. The parallelization of this program is implemented in the Calypso project and we use this program in our system without modification. The ray trace program reads a scene file that describes the ambient setting and objects in the image, including the parameters of the properties. The only objects allowed in this program are spheres and squares. The ray trace program constructs an image according to the scene file. The scene we use in this experiment contains 36 spheres and produces an image of 512 by 512 pixels.

We compare our performance for executing the parallel ray trace program with two other systems, Calypso, and PVM. The experiment calculates the speedup with 1 to 512 jobs running on 1 to 8 machines.

All three systems did poorly when there is only one job per machine in the system. When the granularity is very coarse, the slowest job dominates the performance of the system since there are not enough jobs to balance the load.

Fig. 6.3 shows the result of PVM system. PVM works best when there are just enough jobs to work with, say 4 to 8 jobs per machine. When the number

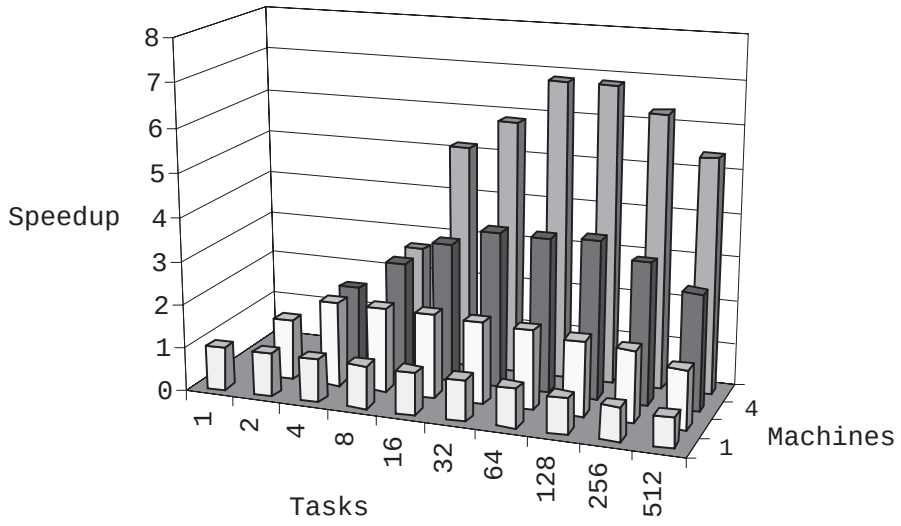


	1	2	4	8	16	32	64	128	256	512
1	1.001	0.996	0.995	0.988	0.973	0.946	0.892	0.803	0.670	0.502
2		1.409	1.973	1.922	1.926	1.870	1.787	1.608	1.339	1.004
4			1.973	2.661	3.730	3.746	3.563	3.203	2.678	2.007
8				2.663	5.073	7.292	7.030	6.378	5.349	4.013

Figure 6.3: Ray trace using PVM

of jobs increases, the overhead of job management starts to affect on the system performance and shows a noticeable drop of speedup.

With all the features of adaptive parallelism, load balancing, and fault tolerance added, the performance of our system is comparable with the others. In Fig. 6.4, we see that our system has a similar performance characteristic to PVM system. That is, the speedup of our system decreases as the number of jobs increases. This is largely due to the overhead required to create new jobs and to load the shared

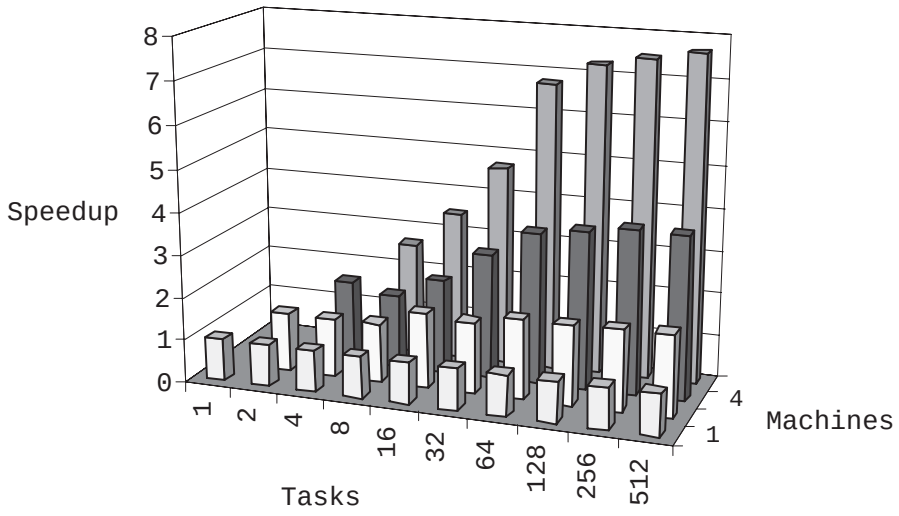


	1	2	4	8	16	32	64	128	256	512
1	0.991	0.988	0.985	0.978	0.964	0.938	0.896	0.821	0.776	0.698
2		1.398	1.960	1.954	1.919	1.871	1.834	1.712	1.633	1.342
4			1.958	2.648	3.221	3.600	3.571	3.635	3.248	2.668
8				2.686	5.171	5.860	6.839	6.830	6.286	5.392

Figure 6.4: Ray trace using our system

memory for each job. However, our system has better performance than PVM when the granularity is fine.

In Fig. 6.5, Calypso system shows a uniform speedups with respect to various granularities of the problem. This is largely due to its caching mechanism that allows later tasks to fetch shared memory cached by earlier tasks instead of sending expensive memory requests. There is also a bunching technique that ties several jobs together and schedules them as one to reduce scheduling overhead. Our system



	1	2	4	8	16	32	64	128	256	512
1	0.981	0.983	0.982	0.982	0.981	0.979	0.977	0.981	0.981	0.980
2		1.384	1.383	1.383	1.770	1.684	1.887	1.874	1.929	1.941
4			1.944	1.724	2.194	2.951	3.560	3.714	3.858	3.856
8				2.635	3.497	4.700	6.754	7.268	7.490	7.664

Figure 6.5: Ray trace using Calypso

does not implement caching and bunching mechanism due to the dynamically changing nature of nested parallelism, as described in Section 7.2.3. An interesting behavior of the Calypso system is that it does not perform as well as others when the granularity of the problem is coarse, like two jobs per machine. Improper bunching of imbalanced jobs may be the cause of the performance drop.

6.3 Quicksort

Quicksort is a very efficient algorithm for sorting a large number of elements. It is a recursive algorithm that repeatedly divides a problem into two smaller subproblems with a pivot. The way pivot is chosen is described later. All values in one of the subproblem are no larger than the pivot and the other subproblem contains all values that are no smaller than the pivot. The subproblems are further partitioned in the consequent steps until it cannot be partitioned further, i.e. when size of the subproblem is two or less. Parallelization of this algorithm is done by converting recursive function invocations into nested parallel jobs. Each job divides the subproblem into two and creates two new sub-jobs to be solved in parallel. The recursive partitioning algorithm fits well in the nested parallel structure.

All the data are stored in a globally shared array with in place data exchange. Since each subproblem takes a portion of the array without overlapping with others, all the sibling tasks are independent. No explicit synchronizations are required in this algorithm. The parallelization process of the quicksort program is discussed in detail in Section 3.3.

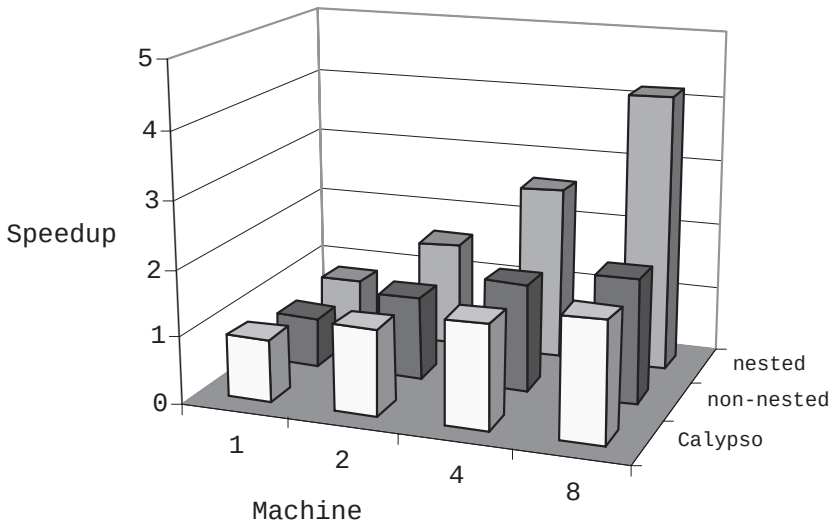
Two optimizations are applied to the standard quicksort program in our experiment. Bubblesort is used to sort small subproblems, and median-of-three (begin, end, and middle points) is used to find the pivot point. To control the granularity of the task size, *parallel threshold* is used. Sequential quicksort is used whenever

the size of a subproblem is smaller than the threshold.

To compare the nested parallel algorithm with non-nested one, we wrote another version of quicksort that does not incorporate nested-parallelism. This version of algorithm solves the problem in rounds of subproblems with the subproblems stored in a shared task array for each round. During each round of execution, all tasks in the shared task array are executed in parallel. However, instead of recursively creating new tasks, each parallel task divides the problem into two and puts the two subproblems in the shared task array. At the end of each round, the sets of new subproblems are collected from the task array for the next round of execution. The bubblesort, median-of-three, and parallel threshold are also implemented in this version of quicksort to make it comparable with the nested version.

We run the nested version of the quicksort using our system, and compare the result with the non-nested quicksort running on our system and Calypso. The result is shown in Fig. 6.6.

Unlike ray trace experiment, this experiment shows a poor overall speedup for all systems. The reason for poor speedup is because the optimal speedup of the parallel quicksort is not linear. For an array of n elements, it takes $n-1$ comparison to divide the array into two. It takes at least $\log(p)$ iterations to produce enough threads for p processors. In [48], an upper bound of approximately 3.5 for



	1	2	4	8
nested	0.85	1.59	2.62	4.17
non-nested	0.74	1.25	1.63	1.88
Calypso	0.93	1.29	1.56	1.80

Figure 6.6: Quicksort experiments

the optimal speedup for fine-grained Quicksort execution utilizing 5 processors of a parallel multiprocessor was theoretically derived. This was also confirmed by experiments on Cm* [49, 21]. Similar behavior was also observed in [9].

Comparing nested parallel quicksort with non-nested one, the nested version shows a much better speedup. This is largely due to the more parallelized execution style in the nested parallel quicksort. In the non-nested version of quicksort, jobs need to wait for other sibling jobs of the same round to be finished before the new

set of subproblems can be processed in next round. Dividing jobs into rounds limit the amount of parallelism that can be achieved.

Other factors that contribute to the better performance of the nested parallel quicksort include distributed memory manager and data locality. The memory managers are distributed across the system because each job becomes a memory manager after it creates two new jobs; hence reduce data contention and network "hot spot" in the system.

The quicksort algorithm has a data locality property since each of the subproblems works on a subset of problem array which the parent job has worked on. By scheduling children tasks on the same machine of their memory manager, the inter-machine communication overhead can be reduced.

6.4 Experiments Conclusion

In this chapter, we showed that our system has a comparable performance with other systems. Although our system has higher overhead than some other systems like PVM and Calypso, our system is more flexible and rich in features. Our system did especially well on a nested parallel program. In the quicksort experiment, we saw that an algorithm that is suited for nested parallel paradigm is not only easier to program, but also runs faster than the non-nested version.

Our system is sensitive to problem size and granularities. The overhead of the

system becomes noticeable for a small problem that can be solved in seconds. Similarly, fine granularity jobs are also not suitable for our system since the overhead to create each job becomes expensive compared with the time required to execute the job.

Chapter 7

Related Research

Software environments for high performance parallel computing on networked machines is an ongoing area of research. We review only the related research that is most relevant to our work. We will stress the criteria of *programming ease*, *scalability*, and *dependability*. Dependability encompasses the ability to handle the realistic needs of imperfect platforms, including load balancing and fault tolerance.

7.1 Relevant Fields of Study

Our research is related to various fields including parallel programming languages, parallel processing in distributed environment, and fault-tolerant computing.

7.1.1 Parallel Programming Languages

There are several ways to express parallelism in a programming language: creating a new language that is suitable for parallel programming, augmenting existing sequential language using explicit parallel constructs and library calls, and using the sequential language by exploiting implicit parallelism.

Creating new parallel languages requires a considerable effort in designing the language syntax and semantics. It is also more difficult for the programmers since they have to learn another programming language.

Exploiting implicit parallelism of a language is often used in a functional or logical language like lisp or prolog. The execution of programs in these languages consists of reduction and expression evaluation, which can often be processed in parallel. Automatically finding implicit parallelism using a compiler in procedural language like C often results in a parallelism that is very small in granularity. Such parallelism may be very useful for multiprocessor machines, but too costly to implement in a distributed environment.

Augmenting existing sequential language with parallel constructs/library calls has several benefits. First of all, since the sequential language is well understood, the system designers can concentrate on the parallel features of the language. It is also easier for the programmers to learn. Secondly, instead of implementing a compiler, a pre-processor is often used to translate the parallel constructs into

control structures and library calls, which reduces the design effort of the system. The preprocessor can be omitted completely if the augmentation of the language is done using library calls or classes only. Our system uses this approach and is based on C/C++.

There are a number of parallel programming languages based on C/C++. The general goals are to provide an easy path for users to migrate their work from C/C++, and to alleviate the frustrations of parallel programming by adding a few extra keywords or classes to the already well understood language. The approaches to parallelize C/C++ can be divided into two categories: data-parallel and task-parallel extensions.

Data-parallel extensions of C/C++ such as pC++[12] describe parallelism in *SPMD* (Single Program Multiple Data) model. Multiple threads of processes execute the same program in parallel on different set of data. The drawback of this type of languages is that expressing concurrency is less flexible, and porting sequential programs to efficient data parallel programs often require significant reorganization and rewriting of the original program.

Task-parallel extensions extend C/C++ with new keywords, objects, and library functions, to express concurrency. Systems in this category includes Compositional C++(CC++)[15], ICC++[16], and Charm++[34]. These keywords can be roughly divided into parallel block and parallel loops.

Language constructs that define parallel blocks can be found in number of parallel programming notations. For example, our `parbegin/parend` notion is equivalent to the use of `cobegin/coend` in CSP[28], `par{}` in CC++ and `conc{}` in ICC++. Statements inside the parallel block are executed in parallel, and the statements are usually independent to one another. ICC++ is an exception that its *conc* block allows mixtures of dependent and independent statements inside a block where statement dependency is based on local data dependence. Independent statements are executed in parallel while the part that depends on others are executed in sequence. This usually result in a small granularity of parallelism, as each statement inside the block might be a single instruction.

In addition to the structured parallel block, CC++ allows unstructured parallelism using `spawn`. The spawn statement acts like UNIX `fork` and creates a parallel task that has no relation to other tasks. It is up to the user to coordinate the spawned task with the rest of the program.

Parallel loops are often implemented by unrolling the loops into parallel tasks. Iterations of the loop body are executed in parallel with a local copy of the loop control variables. CC++ implements parallel for loop with the `parfor (;;) {}` syntax. The statements inside the `parfor` block are executed *sequentially*, while the loop is executed in parallel. We think this syntax is misleading since the programmer will often think the statements in `parfor` are executed in parallel,

as of `par` block. ICC++ allows various kind of looping structure, including `conc for`, `conc while`, and `conc do while`. This is achieved by unfolding the loop dynamically into `conc` blocks, with local variables renamed for each iteration. Our system does not provide explicit parallel loop structures, however, our `routine[]` statement inside a parallel block allows the user to easily create parallel loops.

7.1.2 Parallel Computing in a Distributed Environment

Parallel computing in a distributed environment can be divided into three categories according to the communication model: message passing, remote procedure calls (RPC), and distributed shared memory.

Message passing systems provide messaging services for the tasks to exchange information. Accessing remote data requires composing and decomposing messages to be sent with explicit primitives like `IN/OUT` or `send/recv`. Examples of message passing systems include PVM[53] and Orca[33].

Message passing systems can be efficiently implemented across the network as they resemble the underlying communication mechanism. The users have controls over the messages being sent across the network. However, programming and debugging on these systems are hard and tedious. Porting from sequential program to message passing systems is often difficult. Support of fault tolerance is often limited or is an add-on feature in these systems [44], which increases the

programming difficulty.

In addition to the basic primitives like `send/recv`, remote procedure calls [46] and distributed objects (i.e. CORBA [8]) are built on top of message passing to provide a more convenient means of communication between tasks. The programmers can concentrate on dividing the functions of a program across the network since they are relieved from message composition and data marshaling. Beside user convenience and additional layer of abstraction to communication, the pitfalls of these type of systems is very similar to message passing systems. Programming in these systems is still difficult as the users have to specify the interface of the remote procedures or objects, and to create proxy functions on both ends that actually send messages across the network. Fault tolerance features on these systems are still limited. RPC systems includes GLU[31] and Concert/C [3].

In contrast to message passing and RPC systems, distributed shared memory (DSM) systems provide a shared view of the memory and hide the detail of data distribution by offering a virtual global address space across loosely coupled machines. Parallel tasks communicate with each other through the distributed shared memory. A parallel program running on a parallel machine can be easily converted into a distributed application running over networked workstations with distributed shared memory. Notable systems that built software distributed shared memory include IVY[45], Clouds [19], Munin [14, 9], Midway [10], TreadMarks [1],

and Quarks [42]. High cost of distributed synchronization and lack of fault-tolerant support are the disadvantage of these systems.

A system with similar DSM concepts but with a different approach is Linda[13]. It provides a globally shared space using (database like) *tuple space*. The tuple space is a virtually shared collection of tuples, and a tuple is a sequence of typed values like `("hello", "world")`. Tasks communicate with each other using the tuples. Accessing the tuple space requires special primitives like `in/out/rd` to insert, remove, and read the tuples from the tuple space. Linda is not as easy to program as distributed shared memory systems since it requires explicit primitives to manage the tuple space and it also requires marshaling and unmarshaling data in a tuple. It also suffers some drawbacks of the DSM system like higher communication cost and lack of fault-tolerant support. Several variant of Linda has emerged to address load balancing and fault-tolerant issues. Piranha [24] is built on top of Linda that dynamically balances system load across available machines. Piranha, like the fish, aggressively harnesses idle machine's resources during program execution. However, it does not handle failures. Extensions to handle failures are implemented in FT-Linda [5] and PLinda [32].

7.1.3 Memory Coherence Models of the Shared Memory

Memory consistency is an important aspect in shared memory systems that deal with the question: what is the correct results when multiple tasks read and write to the same memory location.

Maintaining coherent shared memory often increases the overhead in distributed shared memory systems. Therefore, most of the distributed shared memory systems do not implement *strict or sequential consistency* [43]. Instead, weaker memory consistency model are used to achieve high performance.

Following is a list of some popular memory consistency models, in the order of decreasing strictness.

Strict Consistency Requires any read operation of a memory location to return the latest write. A global clock is used to define the order of each operation. This is the same behavior as the single processor system.

Sequential Consistency Lamport defined sequential consistency as: *Result of any execution is the same as if the operations of all processors were executed in some total order, and the operations of each individual processor appear in this sequence in the order specified by its program* [43]. In other words, the order of execution is total ordering and each processor executes in the order specified by the program. Sequential consistency is the same as strict consistency, but no notion of global clock.

Processor Consistency The total ordering is relaxed in processor consistency [26] so that read operation can ignore the concurrent write operation. It requires all previous read operation to be performed before a read is allowed, and all previous read/write operation to be performed before a write operation.

Weak Consistency Synchronization operations is distinguished from ordinary memory access in weak consistency [22]. Synchronization operations are strongly ordered (i.e. total order) while memory access has a weaker ordered. All data access must be performed prior to the execution of subsequent synchronization operation.

Release Consistency The weak consistency is further relaxed by categorizing synchronization operations into *release* and *acquire* in release consistency [25]. The acquire and release operation acts very similar to read and write operation in processor consistency model. The acquire operation, like read in processor consistency, requires all preceding acquire operations to be performed before it is allowed to perform. The release operation requires all proceeding operations to be performed before it is allowed to execute.

Entry Consistency Weak and release consistency requires all data access to be performed before the subsequent synchronization operation. Entry consis-

tency [10] relaxes this constraint and allows data to be associated with synchronization variables. Only the associated shared variables are guaranteed to be up-to-date at the synchronization point.

The first page-based distributed shared memory systems, IVY, implements sequential consistency. Munin was the first distributed shared memory system to use release consistency. It implements multiple memory consistency protocols including sequential and release consistency. Entry consistency is first introduced in Midway system. Other relaxed memory consistency, like the lazy release consistency [41] used in TreadMark and the scope consistency [30], has no direct relation to our system and is not discussed.

7.1.4 Fault-tolerant Computing

Fault-tolerant systems can be categorized according to their approach to handle failures: systems that are designed to handle failures as their first priority, and systems that provide fault tolerance as an add-on feature. The first type of systems are represented by ISIS[11], which provides reliable and ordered multicast messages among process groups to mask network asynchrony and to detect failures. The users are responsible for handling the failure once detected. The system incurs high overhead to maintain reliable and ordered messages, with or without failures.

Other type of systems address fault tolerance separately and provide as an

add-on feature. Fault-tolerant techniques includes *check-pointing*, *replication*, and *migration*. Systems that provides fault tolerance features with these techniques are CIRCUS[17], LOCUS[47], Clouds[19], Fail-safe PVM [44], PLinda [32], and FT-Linda [5]. These systems often provide fault tolerance features independent to other system functions and require user intervention when failures are present.

Calypso [6], Chime [51], and our system, belong to a different group in which load balancing and fault tolerance are naturally supported by the software architecture itself. There is no extra cost for handling failures.

7.2 Relevant Systems

7.2.1 CC++

CC++ [15] addresses task parallelism at a high level and is close to our programming model. CC++ divides the C++ extension into two parts, constructs for parallel machines, and constructs for distributed environments. The syntax for parallel programming uses parallel blocks and loops to express parallelism, while processor objects are used to represent the underlying machine in the distributed environment. In our system we do not employ such a dichotomy—the same program could run both on parallel and distributed platforms. To emphasize the closeness between our programming language and that of CC++ we provide a brief comparative discussion.

Both systems handle static, dynamic, and nested parallelism. In CC++, static and dynamic parallelism is typified by the `par{}` and by `parfor(;;)` statement, respectively. In our system, we use the `parbegin/parend` block to express both dynamic and static parallelism. Inside the parallel block, only `routine` statements are allowed to increase readability. The effect of parallel for loop can be achieved by using the `width` and `id` arguments in the `routine` statement.

Implementations of CC++ (e.g. as in HPC++) do not support fault tolerance, or automatic load balancing as we do.

7.2.2 Dome

Dome [7, 2] is based on C++, and supports data parallelism. It is relevant to our work as it provides load balancing. Dome provides a library of classes and runs on top of PVM. During the execution, when an object of an existing class is instantiated it is partitioned among the available machines, each computing part of the result. During the computation the system estimates the availability of the machines participating in the computation by keeping track of how fast they computed their assigned tasks. If some machines seem tardy, some of the work originally assigned to them may be migrated to (presumably) faster machines. Fault tolerance is provided by using a dedicated layer based on standard techniques, completely independent to load balancing.

There are several differences between our approach and that of Dome. In Dome, tailor-made classes must be written to enable load balancing, requiring specialized effort for the development of new applications, including additional programming complexity, testing, and debugging. Also, the initial performance results indicate that less available machines can hold back fully available machines. For instance, according to [7, 2] an experiment was conducted to evaluate load balancing. When one slow machine is added to a set of fast machines, the overall speed dropped significantly. In contrast, in our system, when a slow machine was added to a set of fast machines, the overall speed increased.

7.2.3 Calypso

Calypso system [6] has the same root as our system, which is based on the techniques of two-phase idempotent execution strategy (TIES) and eager scheduling [4, 39, 37, 35, 36, 38, 40]. Our system and Calypso use similar parallel constructs, and Calypso programs can be executed by our system with little or no modifications.

The fundamental difference between our system and Calypso is the ability to handle nested parallelism and synchronization. In contrast to our system, Calypso does not do nested parallelism and synchronization. In the following, we compare the difference in programming syntax and implementation between the two

systems.

Calypso allows pre-processing and post-processing of a `routine` statement. That is, programmers can specify `routine[][&pre, &post]()` statement, and the `pre()` function will be executed before entering the routine, likewise, the `post()` will be invoked after leaving the routine function. Our system does not implement this feature. However, we allow nested parallelism and synchronization in our system. Explicit nested parallelism like having `parbegin/parend` within a `routine` statement and implicit nested parallelism like embedding parallel block inside a recursive function (as we seen in quicksort example, p. 32) are allowed. Additional parallel constructs and library calls for synchronization are added, like `lock/unlock`, `sync_t` and `assoc`.

In order to support nested parallelism and synchronization, our system uses heavyweight processes to process tasks. That is, new processes are forked when executing tasks. Heavyweight process allows us to suspend a task for synchronization, or to convert waiting parent tasks to become memory manager, which strengthened the fault-tolerant and load-balancing capability in nested parallel programs. Memory managers are distributed across the network and can tolerate failures as the result. In contrast, Calypso uses one process on each machine, called "worker process," to handle all tasks. This is possible since there is at most one task running on each machine, and the tasks will not be suspended as in our

system.

Several performance optimization techniques are developed in Calypso, including *caching* and *bunching*. Caching technique keeps a copy of the shared memory pages previously used in local machines. New tasks arrive with page validation information, and the previously cached shared memory pages are reused if valid. Hence, read only pages are fetched only once for each machine. Our system does not implement caching technique mentioned above since we do not have a centralized memory manager to handle page validation, and we use heavy weight processes instead of reusing the same worker process.

Bunching schedules a set of tasks (a bunch) to a machine at once. The bunch size is calculated based on the number of remaining tasks and the number of currently available machines. Bunching is an effective technique in reducing scheduling overhead for problem with small granularity. However, it may not be as effective in a nested parallel environment since the tasks are more dynamic. Tasks can be suspended, or new ones can be created dynamically in a nested parallel environment. We did not implement bunching in our system.

We proposed a way to tolerate failures of the centralized managers, which is not available in Calypso.

7.2.4 Chime

Chime[51, 50] has the same goal as our system, namely, to allow reliable distributed computation with nested parallelism and synchronization capabilities on a network of workstations.

The difference between the two systems is the approach towards handling nested parallelism. Instead of dealing with jobs creating children jobs directly, Chime breaks a job with parallel blocks into several sub-jobs that contain sequential instructions only. For example, a job that contains one parallel block is broken into two: one contains the function *before* the parallel block, and the other contains the remaining of the function *after* the parallel block. When a sub-job running the first part of the job reached the parallel block, the sub-job is terminated with the current execution environment saved in the system. After all the children jobs of the parallel block are finished, a new sub-job is initiated to resume the execution the remaining job.

The system must have the ability to stop and resume a job in the middle of the execution in order to use this approach. Furthermore, job migration must be enabled to tolerate machine failures, so that sub-jobs can be executed on a different machine when the machine that executed the previous part of the job failed.

The relationship of between jobs is stored in a dependency graph. The system uses this dependency graph to maintain the parent/child and job/sub-jobs

relations. In contrast, our system maintains the parent/child dependency in the execution itself by having the parent job to wait for children jobs.

A similar strategy of splitting jobs is used to handle synchronization in Chime. When a synchronization request is encountered, the job is terminated with all the shared memory updates applied and the execution environment saved. A new job is created to resume the execution after the synchronization request is fulfilled.

This approach works with the original two-phase idempotent execution strategy and eager scheduling (see Section 2.4) without modification, since the nested structures are flattened out. However, the correctness of the computation depends on strict concurrent read exclusive write (CREW) semantic, since jobs must have independent input and output set in order for jobs in the same parallel block not to interfere with each other. No jobs can access an address location while the others may be writing to it. Thus, a variable updated by a job cannot be referenced by any of the sibling jobs or their descendants. For example, suppose a job J_1 creates J_2 and J_3, then any of the children jobs of J_2 will not be able to access the memory location if J_3 or its descendants may modify it. Comparing with Chime, our system allows common concurrent read concurrent write (CRCW-common) semantic such that several tasks can read and write to the same address location at the same time as long as the updates are the same at the end of the parallel block. The value retrieved from the location is the data before the jobs begins,

and the value written takes effect after all jobs in the parallel block are done.

The obvious drawback of Chime's approach is that the system needs to store the intermediate results and execution environment for all sub-jobs, which may require lots of space and bandwidth to handle the data.

Other significant difference between the two systems includes:

- Chime add several additional parallel constructs, including parallel for loop, and dynamic shared memory allocation `shmalloc`().
- Chime allows local variables with scoping rules to be shared as well. It grows a centralized cactus stack to store the variables.
- Chime uses the synchronization model defined in CC++, where each synchronization variable can only be set once. Reading synchronization variables that are not set will cause the thread to wait, and writing to synchronization variables that are already set will cause the thread to abort.
- Chime stores everything (including the program context, shared memory, cactus stack, synchronization information) on a single manager, which is prone to failures and may become a *hot spot* of the network.
- Chime is implemented on windows NT operating system, while ours is developed under UNIX environment. Therefore, there is no performance comparison between the two systems.

Chapter 8

Conclusions

The computational resources are usually not fully utilized in a network of non-dedicated workstations. The computing power in these machines, if harnessed properly, can be used toward computing resource demanding parallel programs at little or no additional cost. However, the unpredictable nature of the non-dedicated network and workstations makes it difficult to executing parallel programs efficiently. Systems must adapt to the continuously changing environment and make appropriate adjustments, without user intervention. A common problem with most systems is that key functions like fault tolerance, load balancing, and shared memory service, are implemented independently with each other, at different layers or as add-on features. Fault tolerance, for instance, is implemented by a layer utilizing conventional techniques at a significant cost penalty in many sys-

tems. Slow machines often dominate the overall performance when load balancing is not well integrated into the system, since the faster machines need to wait for the slower ones to finish. Lack of integrated services often results in poor performance, less fault-tolerant, and not scalable. In contrast, load balancing and fault tolerance are naturally supported by the software architecture itself in our system. The fault tolerance features are provided with no significant performance impact. Furthermore, slow machine does not hold back the faster machines as others do. Extremely slow machines and failed machines are often indistinguishable, and we handle it uniformly in our system.

The programming model of our system provides a flexible, easy to use interface for the users to implement high performance parallel algorithms. We provide the users a fail-free ideal machine to work with, while hiding the actual execution environment. Users are not concerned with the underlying data and work distribution, nor do they handle machine or network failures, so they can concentrate on the problem itself instead of the execution environment.

Our system provides a rich programming model, especially for computations whose complexity emerges only during the execution. Nested parallelism is implemented to allow dynamic creation of addition parallelism. Novel techniques like *nested two-phase idempotent execution strategy* and *prioritized eager scheduling algorithm* are developed to handle the dynamically evolving parallelism.

Explicit synchronization allows the programmers to control the parallel execution as well as to exchange data between jobs. Our model provides a locking mechanism for the users to manage critical sections easily.

One special feature of our system is the uniform treatment of the memory management service and the user tasks. In general, a task either computes or serves as a memory manager. Therefore, memory management services, like the parallel tasks, are distributed in our system. Scalability, load balancing, and fault tolerance are enhanced as the result.

We pay special attention to the fault-tolerant capability of all the components when designing our system. A special fault-tolerant mechanism is proposed to handle the failure of the centralized services, including the global scheduling and synchronization services.

Our system has a comparable performance as other systems like Calypso and PVM for coarse-grain problems. Our system is well suited for problems with dynamically evolving parallelism. With the nested parallel capability, nested parallel programs are easier to implement and run faster in our system than non-nested systems like Calypso.

Bibliography

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 1996.

[2] J. Árrabe, A. Beguilin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel programming in a heterogenous multi-user environment. Technical report, CMU, April 1995.

[3] J. Auerbach, A. Goldberg, A. Gopal, M. Kennedy, and J. Russell. Concert/C: A language for distributed programming. In *Proceedings of the USENIX Winter Conference*, 1994.

[4] Y. Aumann, Z. M. Kedem, K. V. Palem, and M. O. Rabin. Highly efficient asynchronous execution of large-grained parallel programs. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 271–280, 1993.

[5] D. Bakken and R. Schlichting. Supporting fault-tolerant parallel programming in Linda. Technical Report TR93-18, The University of Arizona, 1993.

[6] A. Baratloo, P. Dasgupta, and Z. M. Kedem. Calypso: A novel software system for fault-tolerant parallel processing on distributed platforms. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, pages 122–129, 1995.

[7] A. Beguilin, E. Seligman, and M. Starkey. Dome: Distributed object migration environment. Technical report, CMU, May 1994.

[8] R. Ben-Natan. *Corba*. McGraw-Hill, 1995.

[9] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proceedings of the International Symposium on Computer Architecture*, pages 125–134, 1990.

[10] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. In *COMPCON*, pages 528–537, 1993.

[11] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions of Computer Systems*, 5(1), February 1987.

[12] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel systems. In *Proceedings of the IEEE Supercomputing*, pages 588–597, November 1993.

[13] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[14] John B. Carter. *Efficient Distributed Shared Memory Based On Multi-Protocal Release Consistency*. PhD thesis, Computer Science Department, Rice University, 1993.

[15] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object-oriented programming notation. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. MIT Press, 1993.

[16] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. Technical report, Department of Computer Science,University of Illinois, July 1995.

[17] E. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, 1985.

[18] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omissions and performance faults, and processor joins. In *Proceedings of the 16th International Symposium on Fault Tolerant Computing Systems*, June 1986.

[19] P. Dasgupta, R. J. LeBlanc Jr., M. Ahamad, and U. Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 1990.

[20] P. Dasgupta, Z. M. Kedem, and M. O. Rabin. Parallel processing on networks of workstations: A fault-tolerant, high performance approach. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 467–474, 1995.

[21] J. Deminet. Experience with multiprocesor algorithms. *IEEE Transactions on Computers*, 31(4), April 1982.

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

[23] A Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - a users' guide and tutorial for networked parallel computing*. The MIT Press, 1994.

[24] David Gelernter, Marc Jourdenais, and David Kaminsky. Piranha scheduling: Strategies and their implementation. Technical report, Department of Computer Science, Yale University, 1993.

[25] 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 International Symposium on Computer Architecture*, 1990.

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

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

[28] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[29] S.-C. Huang and Z. M. Kedem. Supporting a exible parallel programming model on a network of workstations. In *Proceedings of the 16th IEEE International Conference on Distributed Computing Systems*, 1996.

[30] L. Iftode, J. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[31] R. Jagannathan and A. A. Faustini. GLU: A hybrid language for parallel applications programming. Technical Report Technical Report SRI-CSL-92-13, SRI International, 1992.

[32] K. Jeong and Dennis Shasha. Plinda 2.0: A transactional/checkpointing approach to fault tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, 1994.

[33] M. Kaashoek, R. Michiels, H. Bal, and A. Tanenbaum. Transparent fault-tolerance in parallel Orca programs. In *Symposium on Experiences with Distributed and Microprocessor Systems*, 1992.

[34] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *OOPSLA Proceedings*, pages 91–108, 1993.

[35] Z. M. Kedem. Methods for handling faults and asynchrony in parallel computations. In *DARPA Software Technology Conference*, pages 189–193, 1992.

[36] Z. M. Kedem and K. V. Palem. Transformations for the automatic derivation of resilient parallel programs. In *IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, pages 15–25, 1992.

[37] Z. M. Kedem, K. V. Palem, M. O. Rabin, and A. Raghunathan. Efficient program transformations for resilient parallel computation via randomization. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 306–317, 1992.

[38] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Combining tentative and definite executions for dependable parallel computing. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 381–390, 1991.

[39] Z. M. Kedem, K. V. Palem, A. Raghunathan, and P. G. Spirakis. Resilient parallel computing on unreliable parallel machines. In A. Gibbons and P. Spirakis, editors, *Lectures on Parallel Computation*. Cambridge University Press, 1993.

[40] Z. M. Kedem, K. V. Palem, and P. G. Spirakis. Efficient robust parallel computations. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 138–148, 1990.

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

[42] D. Khandekar. Quarks: Distributed shared memory as a building block for complex parallel and distributed systems. Master's thesis, Department of Computer Science, The University of Utah, March 1996.

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

[44] J. Leon, A. Fisher, and P. Steenkiste. Fail-safe pvm: A portable package for distributed programming with transparent recovery. Technical Report CMU-CS-93-124, CMU, 1993.

[45] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions of Computer Systems*, 7, 1989.

[46] B. J. Nelson. *Remote Procedure Call*. PhD thesis, Computer Science Department, Carnegie Mellon University, 1981.

[47] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. *Operating Systems Review*, 15(5), December 1981.

[48] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.

[49] L. Raskin. *Performance evaluation of multiple processor systems*. PhD thesis, Carnegie-Mellon University, August 1978.

[50] S. Sardesai. *Chime: A Versatile Distributed Parallel Processing Environment*. PhD thesis, Arizona State University, July 1997.

[51] S. Sardesai, D. McLaughlin, and P. Dasgupta. Distributed cactus stacks: Runtime stack-sharing support for distributed parallel programs. In *the International Conference on Parallel and Distributed Processing Techniques and Applications*, 1998.

[52] R. D. Schlichting and F.B. Schneider. Fail stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions of Computer Systems*, 1(3), August 1983.

[53] V. Sunderam, G. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: evolution, experiences, and trends. *Parallel Computing*, 20:531–545, 1994.