COMPARATIVE STUDY OF

TASK QUEUE ORGANIZATIONS

By

WERAKUL LAOWORAKIAT

Bachelor of Engineering

Kasetsart University

Bangkok, Thailand

1990

Submitted to the faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1996

COMPARATIVE STUDY OF

TASK QUEUE ORGANIZATIONS

Thesis Approved:

Mansur Samadzadeh

Thesis Advisor

Blayne E. Mayfield

Mitchell T Nelson

Thomas C. Collins

Dean of the Graduate College

PREFACE

Scheduling is an important part of multiprocessor and parallel processor systems. Over the past several years, a number of scheduling management policies have been proposed for multiple processor systems. Among those is the hierarchical task queue organization. The performance of this method has been shown to be better than both the centralized organization (using a single global queue of ready tasks) and the distributed organization (using local ready queues associated with each processor). The aforementioned comparative performance study was carried out on and is generally applicable to large systems such as a system in which processors are connected using a multistage interconnection network. The performance of the hierarchical task queue organization on smaller systems such as the Sequent had not been studied before. A number of scheduling management alternatives have been studied on the Sequent system. Both centralized and distributed organizations were implemented and compared. In that study, the hierarchical task queue was not mentioned. The objective of this thesis work was to compare the performance of the scheduling management of a hierarchical task queue to a centralized task queue and a distributed task queue.

Each of the scheduling management systems was implemented on the Sequent system initially using the C++ task library and subsequently using UNIX processes with shared memory due to some constraint of the C++ task library. Benchmark programs were applied to each scheduling method and the results were analyzed. The resulted indicated that the Hierarchical Organization was superior to both the Centralized and Distributed Organizations in terms of task queue contention and load sharing.

ACKNOWLEDGMENTS

I wish to express my appreciation and gratitude to my advisor Dr. Mansur H. Samadzadeh for accepting to be my major advisor, his advice, his intelligent guidance, and his assistance. His constructive criticism, direction, wisdom, and counsel during my graduate studies have been a constant source of inspiration and motivation that helped me gain confidence academically and professionally. I also wish to thank Drs. Blayne E. Mayfield and Mitchell Neilsen for serving on my graduate committee.

Additionally, I want to thank Dr. George J. Sabbagh, my supervisor at the Biosystem and Agricultural Engineering Department, Oklahoma State University, for his support and for employing me as a Graduate Research Assistant. I would also like to thank Mr. Mark Vasoll for his helpful advice and for changing the system configuration (kernel parameters) to facilitate this project.

Last but not the least, I would like to express my sincere gratitude to my parents, Sunseng Hear and Supawadee Laoworakiat for their continued support and encouragement, without which this endeavor would not have been successful.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Increasing the power of computer systems to attain faster speeds is of continuing interest. To satisfy the ever-increasing computing needs and to handle the increasing complexity of the resulting programs, new technology to improve throughput and reduce job latency is needed. Figure 1 shows the performance growth of microprocessors from 1984 to 1995.

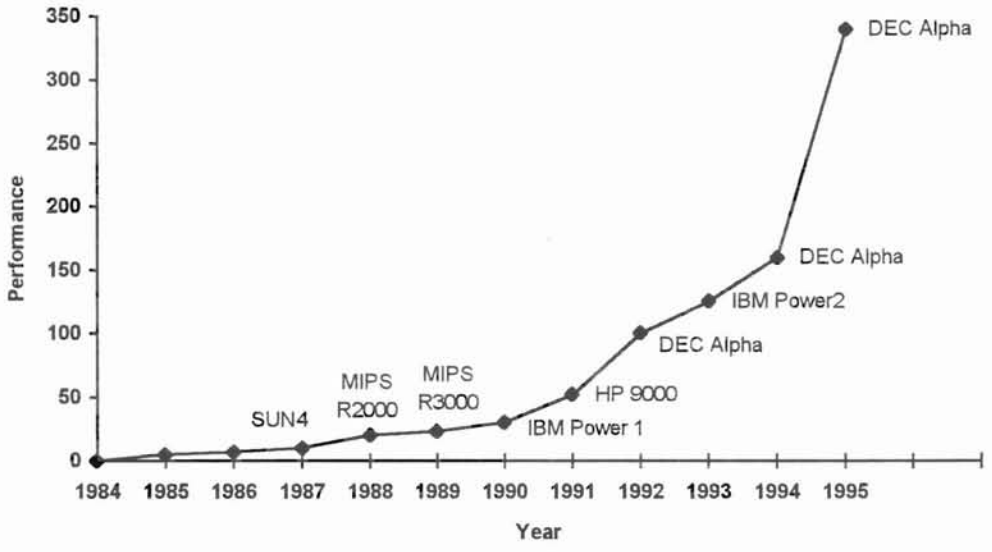

Figure 1. Growth in microprocessor performance from 1984 to 1995 (adapted from [Hennessy and Patterson 96])

To achieve better system performance, one approach is to improve the clock rate or the clock cycles per instruction (CPI) and instruction count of a processor. The other approach is to use multiple processors running simultaneously. A multiprocessor has obvious advantages over a uni-processor because a task is separated into many threads and executed by many processors at the same time. As Steven mentioned [Steven 90]:

> Multiprocessor systems have many key advantages over their uniprocessor counterparts. One is the obvious potential for greater computing power, allowing otherwise impossible performance levels to be achieved. This increased computing capacity can be realized at a relatively low cost, making cost/performance another important advantage. Moreover, the scaleable design for many systems supports easy expansion of the system's computing capacity by adding more processors. A final important benefit is increased reliability since with careful design the failure of one processor will only decrease the system's computing power rather than halting the entire system.

Although multiprocessor systems have advantages over uni-processors, the way to organize a system with many processors running at the same time needs more complicated handling and management than a single-processor system to deliver good performance. Otherwise, it might give worse performance than that of one processor running alone. Such handling and management constitute a major factor in parallel system performance, namely system scheduling.

Scheduling is an important part of parallel processing systems. In multiprocessor environments, the selection of a particular processor to execute an incoming task directly impacts the performance of the system. As Dandamudi and Cheng mentioned, "processor

scheduling is an important factor that influences the overall system performance" [Dandamudi and Cheng 95].

There are two basic ways to approach the problem of scheduling management: centralized organization and distributed organization. Centralized organization is the scheduling management approach in which there is one single task queue maintained for all processors. In contrast, distributed organization is the scheduling approach in which there is a separate task queue for each processor. If there is no ready task queue access contention, the centralized organization provides better performance due to its load sharing characteristic [Dandamudi and Cheng 95]. However, when the system gets larger (e.g., in a system in which processors are connected using a multistage interconnection network), the increasing access to the single global task queue can decrease the performance of the system due to task queue access contention [Dandamudi and Cheng 95]. On the other hand, the distributed organization, while eliminating the access contention problem, suffers from the major problem of how to find an appropriate ready task queue for the arrival tasks. As a result, the performance of the distributed organization can be substantially worse than that of the centralized organization in the absence of the ready queue access contention [Anderson et al. 89].

Although in Dandamudi and Cheng's research the hierarchical organization has been shown to be superior to the centralized and distributed organizations, the research was done only on large systems [Dandamudi and Cheng 95]. The performance of the

hierarchical task queue on a shared bus system such as the Sequent, as in Anderson et al.'s research, has not been studied before.

The main thrust of this study was to compare the performance of the scheduling management of a hierarchical task queue to centralized and distributed task queues on the Sequent system. The study was based on Anderson et al.'s research by applying the hierarchical task queue organization. The C++ task library [AT&T 90] was initially used to implement and simulate each of the scheduling management systems. However, due to the built-in synchronizing aspects among processes that are part of the C++ task library environment, the real scenario which is composed of many non-synchronizing tasks didn't seem to be simulated using the library. The simulation program was reconstructed by replacing the C++ task library calls with UNIX processes along with UNIX semaphores. The benchmark programs that work the same as the ones used in Anderson et al.'s research [Anderson et al. 89] were used and the results were analyzed.

Chapter II of this thesis provides a review of the current literature on scheduling of systems in parallel environments. Chapter III provides a discussion of the design and the implementation details of the software that was developed as part of this thesis. The testing and evaluation of the software developed are discussed in Chapter IV. The thesis ends with Chapter V that provides a summary, the conclusions drawn from the study, and some suggestions for future work.

CHAPTER II

LITERATURE REVIEW

The major objective of this chapter is to offer some insight into the principles underlying the performance of scheduling strategies in parallel systems. The basic model and policies from previous studies are shown. Studies about Centralized and Distributed Task Queue Organizations are discussed. The new data structure for scheduling management, the Hierarchical Organization, is introduced at the end of this chapter.

2.1 Scheduling Management and Policies

2.1.1 Scheduling Management

Researchers have realized the importance of scheduling management for years. A number of studies have been carried out and tools have been created to help in scheduling and dispatching of tasks. According to Anderson et al., there are five alternative job scheduling management approaches [Anderson et al. 89].

- Single Lock: Central data structures protected by a single lock.
- Multiple Locks: Central data structures each protected by a separate lock.
- Local Free List: Per-processor free lists without locks; a central locked ready queue.

- Idle Queue: A central queue for idle processor; per-processor free lists.
- Local Ready Queue: Per-processor ready queues; per-processor free lists.

Single Lock: All data structures are protected under a single lock. When a processor needs access to get a task to execute, it has to acquire the lock, then do what is needed to the shared data, and finally release the lock when finished.

Multiple Lock: Each shared data structure has its own lock. Each operation on the data structure can then be surrounded by a lock acquisition and release. The throughput of the system is better than that of a single lock because the activity of the job is split among several locks. However, the latency increases because more lock accesses are needed.

Local Free List: The single lock and multiple lock approaches mentioned above apply locks to all shared data structures including the memory list needed to execute the task. The Local Free List reduces lock acquisitions between different processor. Each processor itself maintains its own free list of the memory blocks while there is still a single shared ready task queue whose accesses are locked. As a result, fewer lock acquisitions are needed per task. Consequently, the latency is lower and the throughput is better than multiple or single lock approaches.

Idle Queue: A queue of idle processors is maintained in addition to the central queue of tasks. When there are ready tasks waiting to be executed in the task queue, the procedure will be no different from the other task management strategies. When a processor is idle, it preallocates memory, initializes the stack, and puts itself into the processor queue before the task comes. This method will reduce latency when there are

many idle processors. At the same time it will increase latency when all processors are busy due to the added complexity.

Local Ready Queue: For this task management approach, each processor not only maintains its own free memory list, but it also maintains its own task queue. The problem of task queue contention is lessened because there is no more a need to lock a single global queue among processors to access a task. At the same time, it introduces the new problem of how to keep the tasks in each local queue balanced. As a result, the performance of the system is determined by how to insert the job into those sets of local queues, instead of how to dequeue the job to execute.

In this study, the major interest is on the Local Ready Queue that is the main characteristic of the Distributed Organization, and the Local Free List that is the main characteristic of the Centralized Organization.

2.1.2 Scheduling Policies

In addition to the task management approaches mentioned above, scheduling policies play a significant factor in the performance of parallel system. Most of the parallel scheduling literature discusses policies along with scheduling management approaches. Basic scheduling policies can be classified into two categories [Majumdar et al. 88] [Tanenbuam 92], as outlined below

- Policies Independent of Job Characteristics: Processes are scheduled independently without explicit knowledge of job characteristics. Policies that belong to this category include:
  - First Come First Served (FCFS): When a job arrives, each of its processes is placed consecutively at the end of the shared process queue. When a processor becomes idle, it simply removes the first process from the queue and runs it to completion.
  - Round Robin (RR): When a job arrives, it is assigned to the ready queue and the ready queue is served in a cyclic fashion.
  - Random Scheduling: The arriving jobs are assigned to the ready queue randomly.
- Policies Based on Job Characteristics: Processes are scheduled based on the knowledge regarding job characteristics. Policies that belong to this category include:
  - Priority Scheduling: The processes are allocated on the basis of their pre-assigned priorities.
  - Shortest Job First (SJF): The shared process queue is organized as a priority queue, with the highest priority given to the task that has the shortest period of execution time.

A search of the relevant literature indicates that many other policies besides those mentioned above have been proposed and studied. Majumdar et al. proposed the Smallest Number of Processors First policy and the Smallest Cumulative Demand First policy

[Majumdar et, al 88]. Leutenegger and Vernon proposed the Coscheduling policy and the Dynamic Partitioning policy [Leutenegger and Vernon 90]. Steven [Steven 90] investigated the comparative aspects among various scheduling policies including Fixed Processor (FP), Last Processor (LP), Minimum Intervening (MI), Limited Minimum Intervening (LMI), and Limited Minimum Intervening Routing (LMR). Zahorjan and McCann's study compared the classes of Static Policies and Dynamic Polices [Zahorjan and McCann90].

In this study, the focus is made on the solution of the contention problem which related principally to Scheduling Management. Only the first come first served and round robin policies were applied in the simulation program. The round robin policy was applied to the Distributed Task queue organization simulation because of the best results obtained compared to other policies [Andersone et al. 89]. For the Centralized and Hierarchical Organizations, the first come first serve policy was applied.

2.2 Centralized and Distributed Task Queue Organizations

In Ni and Wu's study [Ni and Wu 89], the consequences of mutually exclusive access to a centralized organization were studied. They showed that for the M/M/n queueing system [Kleinrock 75] with a single shared ready queue, when the number of processors increases, the scheduling overhead cannot be ignored (as it is in traditional M/M/n queueing systems). According to Ni and Wu:

> To ensure mutually exclusive access of the shared ready queue, only one processor is allowed to access the ready queue at a time. Thus, the scheduling overhead is nonnegligible and the actual system performance will be worse than the theoretical system performance.

Ni and Wu also proposed strategies to make the distributed organization more effective [Ni and Wu 89]. In their model, system processors are partitioned in to k groups where each group has a dedicated queue. When a task arrives, it is randomly routed to one of the k queues; and when a processor becomes idle, it selects a single task from its dedicated queue. A large value for k decreases contention for system queues but increases load imbalance; a small value for k has the opposite effect. However, with these strategies, the performance of a distributed organization is reportedly still worse than that of a centralized organization without task queue contention.

As expected, further studies have been reported in the literature that try to improve on the performance reported previously in the literature. Zahorjan and McCann proposed a more promising approach to solve the problem of task scheduling for shared memory multiprocessors using a single queue of runnable tasks [Zahorjan and McCann 90]. Their proposed technique is called "two-level schedulers". This technique was reportedly implemented by assuming that, at any point in time, each job is composed of one or more ready tasks and that the operating system deals solely with allocating processors to those ready tasks. The study assumed that the task queue contention is negligible, so this method cannot be applied to large systems with a large number of processors and task queue contentions.

In small systems, the performance of both scheduling techniques (i.e., centralized and distributed) was studied by Anderson et al. [Anderson et al.89]. That research was done under the DYNIX operating system on a Sequent Symmetry computer system, which is a shared memory multiprocessor system. Figure 2 shows the result from

Anderson et al.'s work. Like the result from Ni and Wu's work [Ni and Wu 89], the graph shows that for the centralized organization, as the number of processors increases, the performance of the system decreases due to task queue contention.

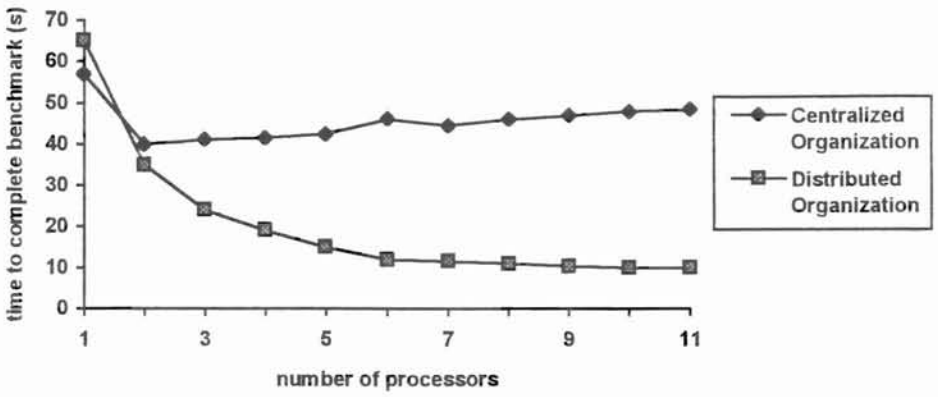

Figure 2. Performance comparison between Centralized and Distributed Organizations (adapted from [Anderson et al. 89])

2.3 Hierarchical Task Queue Organization

Another technique in scheduling management, called the Hierarchical Organization, was recently proposed by Dandamudi and Cheng [Dandamudi and Cheng 95]. This technique avoids the shortcomings associated with the centralized and distributed organizations. A set of ready task queues is organized as a tree with all the processors attached to the bottom of the tree, as depicted in Figure 3. The branches of the tree can be adjusted to improve performance. All incoming tasks are inserted at the root queue. When a particular processor is looking for a job to execute, it will check the local queue first. If the local queue is empty, it moves up to check the higher level queue until

it finds a task to be scheduled. To reduce the queue access contention, when a higher level task queue is being accessed, a number of jobs in the queue are moved down from that queue. In Dandamudi and Cheng's study [Dandamudi and Cheng 95], the number of tasks moved down from a queue and the number of branches of the tree can be adjusted to get a better performance than the centralized and distributed organizations.

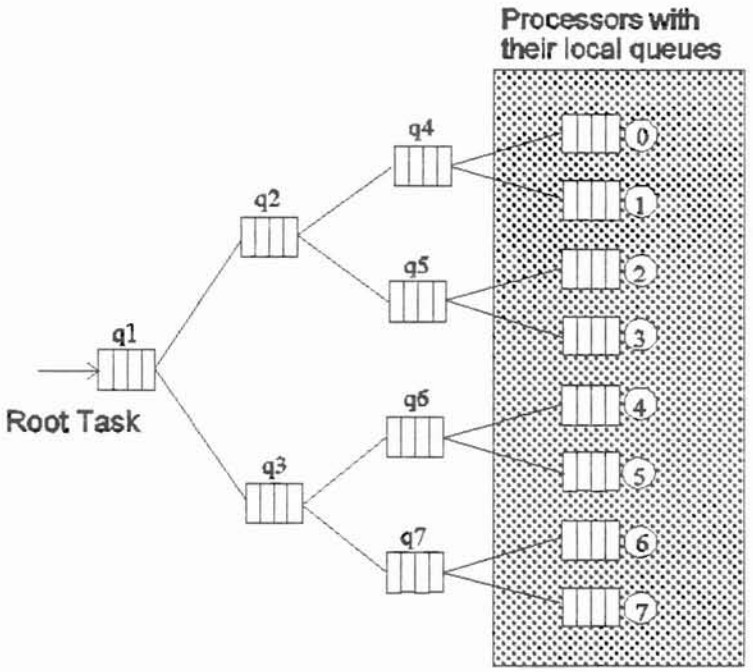

Figure 3. Hierarchical Organization for a branching factor of two with 8 processors (source: [Dandamudi and Cheng 95])

The set of tasked moved down from a queue is determined by a parameter called transfer factor (Tr). This parameter is defined to give the relation between the number of jobs transferred and the number of processors below the queue. The reason behind defining such a parameter is that it makes sense to have the number of jobs correlate to the

number of processors under each branch of the tree. As the result, with the different value of the transfer factor, we can find the optimum performance of the system. The transfer factor is defined as follows:

$$Tr = \frac{number\ of\ tasks\ moved\ one\ level\ down\ the\ tree}{number\ of\ processors\ below\ the\ child\ task\ queue}$$

To further illustrate how the Hierarchical Organization task queue works, an example of the task queue for a 128 processor system and a tree structure with a branching factor of 8 is shown in the Figure 4. The depth of the three therefore equals to 2 due to the number of processor and branch factors. Suppose there are 200 jobs waiting to be executed at the root queue. When a processor is available to execute jobs, which are all initially in the root queue. With the transfer factor of one, 64 jobs are transferred from the root queue to Q1. Of the 64 jobs, 8 jobs are transferred to Q5 leaving 48 jobs at Q1. Finally, processor 1 schedules a task by moving one task into its local queue. The task transfer process with the trasfer factor of two is also shown in Figure 4.

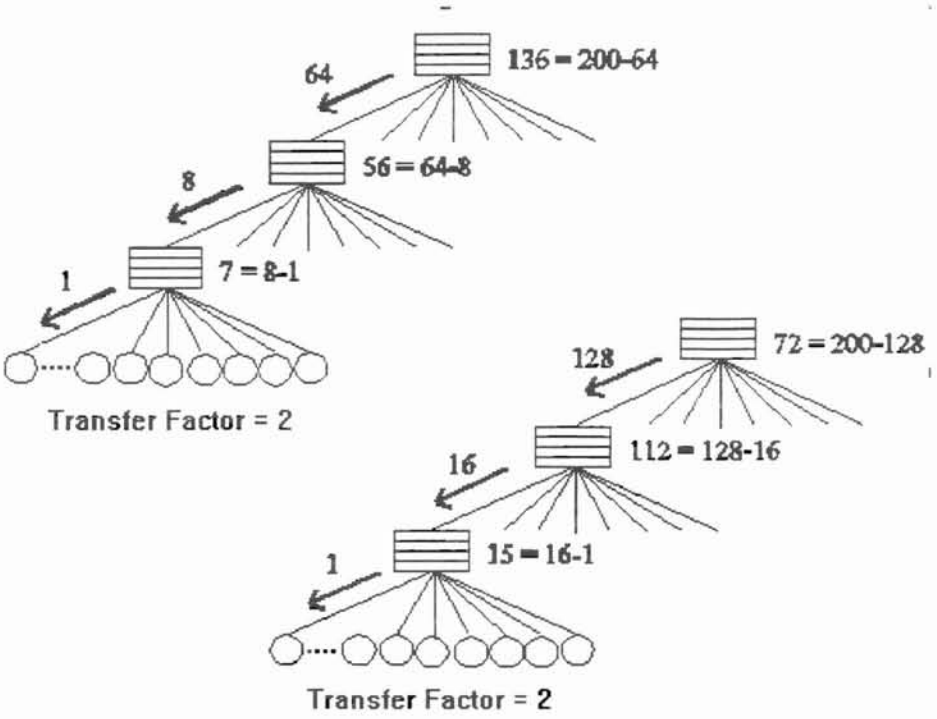

Figure 4. Task transfer process in the Hierarchical Organization for a 128 processor system with a branching factor of 8 (for simplicity, the local queues of the processors are not shown)

The set of task queues in the Hierarchical Organization distributes the set of tasks to different memory modules, so task accesses can be carried out concurrently. Consequently, it avoids the ready queue bottleneck problem and achieves good load sharing. Moreover, the branching and transfer factors can be adjusted; hence the optimum performance of the system can be reached. Being a superior data structure to both Distributed and Centralized Organizations, the Hierarchical Organization nonetheless introduces an overhead, in that more task queues will have to be accessed. As for the actual impact of these factors, Dandamudi and Cheng showed that the average number of queue accesses is very close to one for higher values of the branch factor and the job transfer factor [Dandamudi and Cheng 95].

CHAPTER III

DESIGN AND IMPLEMENTATION ISSUES

3.1 Implementation Platform and Environment

3.1.1 Sequent Symmetry S/81

The Sequent Symmetry S/81 is a mainframe class computer system with a multiprocessor architecture that was developed by Sequent Computer System, Inc. The multiprocessing and shared memory architecture consist of the following elements [Sequent90]:

- A parallel architecture that utilizes multiple industry-standard microprocessors.
- Either the DYNIX v3.0 or the DYNIX/ptx operating system (both being UNIX system ports).
- A standard set of interfaces to the network such as Ethernet, SCSI, VMEbus, and MULTIBUS.

The operating system of the Sequent Symmetry S/81 has been engineered to incorporate features that support its parallel architecture. In addition, software that has been built for the UNIX operating system can run on the Sequent Symmetry S/81 with

little or no modification. In the case of multi-user applications, the operating system of the Sequent Symmetry S/81 automatically distributes the tasks to multiple processors in an attempt to reduce the response time and increase the system throughput [Sequent90].

The DYNIX v3.0 operating system supports the two major command sets of UNIX, namely the Berkeley UNIX and UNIX System V. On the other hand, the DYNIX/ptx operating system is compatible with AT&T System V v3.2 only [Sequent 90].

3.1.2 C++ Task Library

The task library is a coroutine support system for C++. Tasks can communicate by explicit sharing of data, by messages, or by data pipes. A task is an object with the associated coroutines. Each task is an instance of a user-defined class derived from the class task, and the program of the task is the constructor of its class. The tools available in the task library provide facilities for several styles of multi-thread programming in a single-language, single-address-space environment [AT&T90].

However, due to the fact that all tasks in the task system run as a single UNIX process, the initial commitment to utilize the library had to be reconsidered. Specifically, each task would release the processor and let another task execute in the non-preemptive mode, so access to the shared data structure, which needs to be done asynchronously, seems not to be simulated. As a result, UNIX processes with shared memory and semaphore were used in place of the C++ task library.

3.1.3 Semaphore

To ensure mutual exclusion on queue accesses, semaphores were used in the simulation program. The semaphore concept was first put forward by the Dutch theoretician, E. W. Dijkstra as a solution to the problems of process synchronization. A semaphore *sem* can be seen as an integer variable on which the following operations are allowed.

```
wait(sem)
      if(sem!=0)
            decrement sem by one
      else
            wait until sem becomes non-zero

signal(sem)
      if(queue of waiting processes not empty)
            restart first process in wait queue
      else
            increment sem by one
```

The active part of both operations must be indivisible to insure the integrity of the value of the semaphore variable. In other words, they must run atomically, which means that only one process can ever change `sem` at any time. Otherwise, it could lead to a race condition. In UNIX system V, the implementation of semaphores is done in the kernel, where it is possible to guarantee that a group of operations on a semaphore is done atomically with respect to other processes. This implementation is expanded in two directions [Stevens 91] as listed below.

1. A semaphore is not a single value but a set of nonnegative integer values. The number of nonnegative integer values in the set can be from one to some system-defined maximum.

2. Each value in the set is not restricted to zero and one. Instead each value in the set can assume any nonnegative value, up to a system-defined maximum value.

In the simulation program, semaphores were used together with shared memory and UNIX processes generated by the `fork()` system call (in place of the C++ Task library). The design and implementation of the simulation program is discussed in the next section.

3.2 Design and Implementation

There are three major parts in the simulation program: queue simulation, processor simulation, and lock simulation.

3.2.1 Queue Simulation

The concept of object oriented program was utilized in the design of the queue structure, i.e., the queue class. The class of queue is composed of a job list, semaphore id which is used to prevent two processors to access the queue at the same time, and pointer to parent queue in the case of Hierarchical Task Queue Organization. The job list was implemented as a circular link list where the next field of the last element pointed to the first element in the list. Each element was designed to be an object such that when it is sent a message to execute these objects, it will execute the instruction set in the assigned task. This design made it possible to perform thread simulation where the simulated threads were not fixed to a certain operation. The threads can help simulate a scenario closer to the process running in the real world.

Another major requirement for the queue class was that the instances of queue class needed to be in shared memory so that any process can access them. The `new` operator of each class was overwritten to allocate the space of class instance in shared memory before it was initialized by the constructor. The `delete` operator was also correspondingly replaced. The shared memory manager was made possible by `shmalloc()` and `shfree()` system calls otherwise we had to manage memory ourselves by using the `shmget()` and `shmat()` system calls.

3.2.2 Processor Simulation

Each processor was simulated by a UNIX process created by `fork()` system call. There was an extra process (from the number of processors) to generate jobs using the `new` function and to enqueue them to the queue. The rest of the processes just access the queue and execute jobs and delete them using `delete()`.

3.2.3 Lock Simulation

In Hierarchical Organization, each queue object in the tree structure is equipped with its own lock into its private part of data. The lock was formed by using a semaphore. Three semaphores were used to simulated each lock. The first one is implemented as the actual semaphore value that can be initialized, incremented, and decremented. Functions are provided to increment and decrement by one, or by some other integer value. The second semaphore is used as a counter of the number of processes currently using the first semaphore. This is so that the main semaphore can be deleted when no more processes are using it. The third semaphore is a lock variable for the main semaphore. This setup is

required to protect from the race condition when the main semaphore is being initialized and closed. The initial design was to equip each queue with its own semaphores which is composed of a set of three semaphores. However, due to the system constraint (i.e., the kernel constraint of DYNIX/ptx) that there can be only 52 sets of semaphores in the system, the design was changed to one set of semaphores shared by all queue classes in the program. Before all queue objects were created, the semaphore set is initialized with the number of semaphores equal to the number of queues in the system multiplied by 3. There is no race condition resulting from accessing semaphore sets among processes that access different queues because the ids of the semaphores in different queues are different.

CHAPTER IV

EVALUATION

4.1 Benchmark Programs

The study assumed the input to the simulation to be a set of jobs composed of a number of independent tasks that could be run on the system concurrently. The benchmark programs contained one thousand threads which simulate a brief computation burst by "sleep"ing for a specified amount of time. There were two versions of the benchmark programs. One consisted of threads which sleep a fixed amount of time, i.e., have a fixed execution time. The other version used a random function to generate the time for each thread to sleep so that the simulation would be closer to the real world in which each thread has a potentially different execution time. Each of the benchmark programs was created and inserted at the root queue by a process which functioned as a job producer.

4.2 Simulation Result and Analysis

Figure 5 shows the result from running the benchmark with the fixed amounts of execution time on all of the three task queue organizations. We can see from the graph

that there are no differences among those three task queue organizations due to the shared memory queue access time. The access time is much lower compared to the thread execution time. The contention for accessing the queue therefore does not cause much problem for the performance of the system. Figure 6 shows the same result even though the input benchmark has been changed to randomly variable execution times.

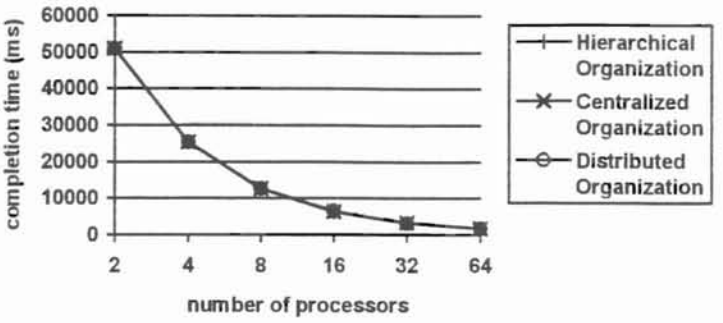

Figure 5. Performance comparison among the Centralized, Distributed, and Hierarchical Organizations with branch factor 2 and job transfer factor 1 (the benchmark is a set of one thousand threads with fixed 100 ms idle time each)

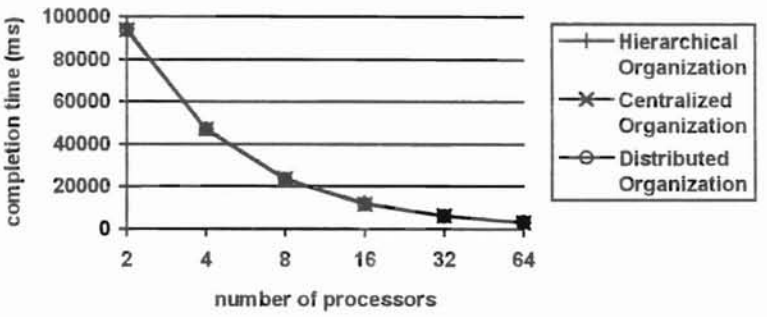

Figure 6. Performance comparison among the Centralized, Distributed, and Hierarchical Organizations with branch factor 2 and job transfer factor 1 (the benchmark is a set of one thousand threads with a random execution time of 0, 100, 200, or 300 ms)

To obtain more realistic results, a delay was introduced into the queue access times of each queue to study the effect of access times on each queue. Figure 7 shows the result from running the simulation program with different queue access times. The access time is represented as a delay which is introduced by a loop with a controlled number of iterations. The delay is made by having the loops upper bound range from zero to a certain number (28,000 in the case of Figure 7). The result shows that, for the Centralized Organization, the task queue contention overhead increased when the queue access time got larger. On the other hand, the delay does not have as much effect on the Hierarchical

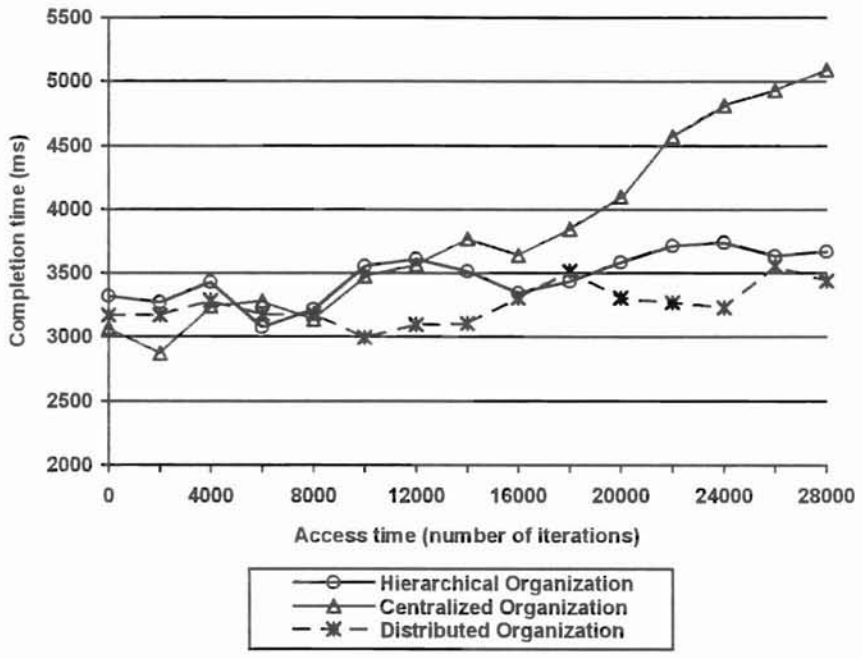

Figure 7. Time in milliseconds to finish the benchmark program with different queue access delays introduced for queue access time (X axis = the number of loop iterations representing access delay); the benchmark program is a set of threads with random execution times of 0, 100, 200, or 300 ms

or Distributed Organization as it does on the Centralized Organization. This result supports Dandamudi and Cheng's study [Dandamudi and Cheng 95] in that task queue contention doesn't cause much trouble in the Hierarchical Organization. Figure 8 shows the slight effect of the task queue access delay time, which is the main cause of contention, to the Hierarchical Organization in both fixed time and random time benchmarks. Figure 9 also shows that the characteristics of the Heretical Organization doesn't depend on the queue access delay time.

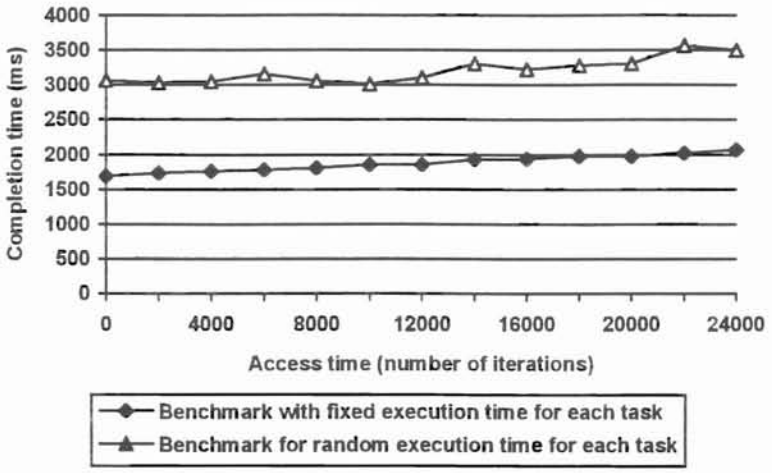

Figure 8. The result from running the Hierarchical Organization with branch factor 2 and task transfer factor 1 with different queue access time delays

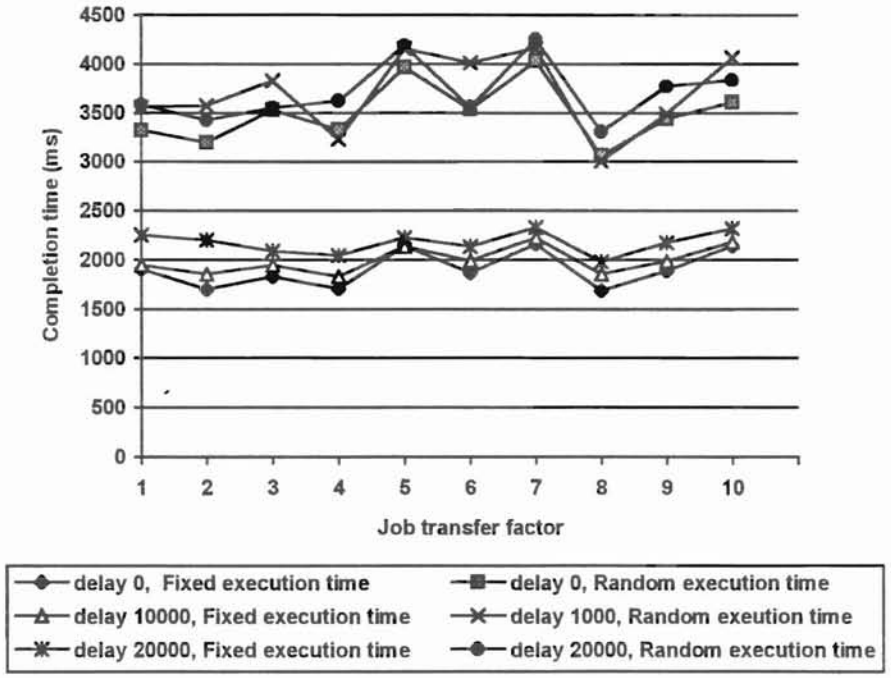

Figure 9. The result from running different benchmarks in the Hierarchical Organization

The preliminary results (Figure 10) were further investigated by adding a variety of execution times to 0, 100, 200 , 300, 400, and 500 to study the advantages of the Hierarchical Organization over the Distributed Organization in terms of load sharing. Figure 10 shows that in the Distributed Organization with the round robin policy, the result from running the benchmark is significantly different from the Hierarchical Organization with a branch factor of 2. This is worse when compared to the Hierarchical Organization with a branch factor of 8, because the load sharing is better when the number of branches increases.

For the Distributed Organization with the round robin policy, the load sharing is not possible because the scheduler has no feedback information about the processor(s)

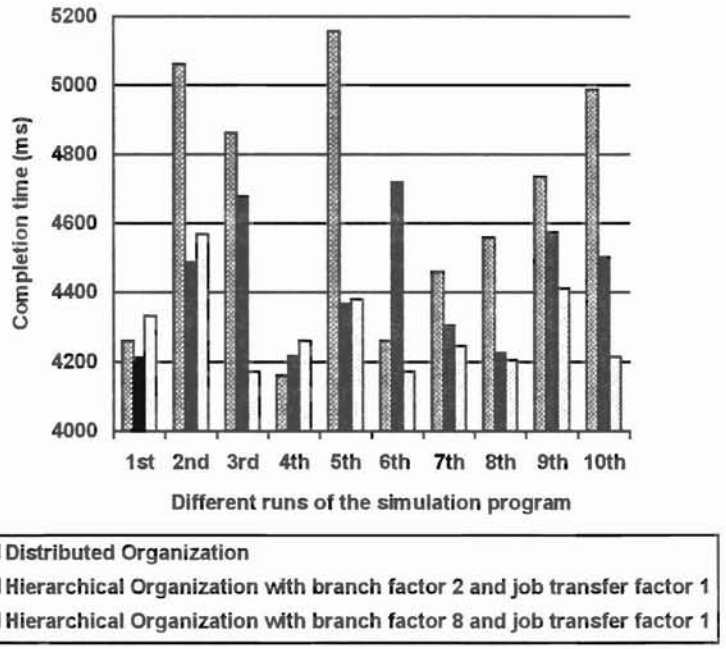

Figure 10. Different runs of the simulation program on the Hierarchical and Distributed Organizations with 64 processors

load. On the other hand, load sharing in both the Centralized Organization and the Hierarchical Organization is better due to the fact that the processors themselves are responsible for bringing the task from the queue to execute. Therefore, if a processor has a short execution time job, there is more chance to get another job to run than with a processor that is running job with a longer execution time. On the other tradeoff, there is task queue contention in the Centralized Organization because each processor has to make the other processors wait while it accesses the queue while the queue accessing can be made parallel in the Hierarchical Organization.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

The importance of scheduling in multiprocessor environments was discussed in Chapter I. Chapter II presented a survey of the literature on scheduling management and policies, including on introduction to the Hierarchical Task Queue Organization. The implementation details of the simulation program for each task queue organization were outlined in Chapter III. Chapter IV provided the results from running benchmark programs in each task queue organization.

As expected, the simulation model did not show much difference among the running times for the three different task queue organizations. This was due to the queue access time being small compared to the execution time of each task The queue access time and the execution time do not have much difference (as in the simulation model) in the real situation for shared memory systems. Therefore, a delay was added to represent the queue access time to make the simulation model closer to the reality. As a result, the outcome became much closer to what was expected. The task queue contention problem is solved by the Hierarchical Organization, while the load sharing aspect is also

maintained, hence making the Hierarchical Organization more effective than both the Centralized Organization and the Distributed Organization.

5.2 Future Work

In this study, the Hierarchical Organization was investigated based on the first come first served policy. Other alternatives of queue access in the Hierarchical Organization can be explored. Another improvement to the Hierarchical Organization would be to apply different branching factors to each queue node in the task queue organization. In the case where the system is composed of different kinds of processors (a heterogeneous system), it appears that the branching factor should be a major factor in making the system reach the optimum performance.

REFERENCES

[Anderson et al. 89] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Computers,* Vol. C-38, No. 12, pp. 1631-1644, December 1989.

[AT&T90] UNIX System V AT&T C++ Language System Release 2.1, Library Manual, 1990.

[Dandamudi and Cheng 95] S. P. Dandamudi and P. S. P. Cheng, "A Hierarchical Task Queue Organization for Shared-Memory Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 6, No. 1, pp. 1-16, January 1995.

[Hennessy and Patterson 96] J. L Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.

[Kleinrock 75] L. Kleinrock, *Queueing Systems, Vol. 1*, John Wiley and Sons Ltd., New York, NY, 1975.

[Leutenegger and Vernon 90] S. T. Leutenegger and M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* Boulder, CO, pp. 226-236, May 1990.

[Majumdar et al. 88] S. Majumdar, D. L. Eager, and R. B. Bunt, "Scheduling in Multiprogrammed Parallel Systems," *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* Santa Fe, NM, pp. 104-113, May 1988.

[Ni and Wu 89] L. M. Ni and C. E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems," *IEEE Transactions on Software Engineering,* Vol. SE-15, No. 3, pp. 327-334, March 1989.

[Sequent 90] *DYNIX/ptx User's Guide,* Sequent Computer, Inc., 1990.

[Steven 90] S. M. Steven, "Issues in Shared Memory Multiprocessor Scheduling: A Performance Evaluation," Doctor of Philosophy dissertation, Department of Computer Science and Engineering, University of Washington, Seattle, WA, 1990.

[Stevens 91] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Tannenbaum 92] A. S. Tannenbaum, *Modern Operating Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[Zahorjan and McCann 90] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors," *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, CO, pp. 214-225, May 1990.

APPENDICES

APPENDIX A

GLOSSARY

Benchmark: A program or a set of programs used for testing purposes.

Clock Cycle per Instruction: The number of clock cycles to execute one instruction of a program.

Clock Rate: The inverse of clock cycle time, usually measured in Mhz

M/M/m: A queueing system with Markovian or Poisson arriving tasks, Markovian or Poisson task departures and m identical processors.

Job: A system command, a user program, or a task given to a scheduler for scheduling.

Multiprogramming: Allowing more than one program to be in some state of execution (not necessarily executing) at the same time.

Parallel Processing: A type of information processing that emphasizes the concurrent manipulation of data elements belonging to one or more processes solving a single problem.

Prescheduled: A type of partitioning algorithm in which each processor is allocated its share of the computation at compile time.

Scheduling: An allocation of tasks to the processor(s).

Thread: A lightweight process with a separate program counter and stack of activation records.

APPENDIX B

TRADEMARK INFORMATION

DYNIX/ptx: A registered trademark of Sequent Computer Systems, Inc. DYNIX/ptx is an operating system for Sequent Computers.

Symmetry S/81: A registered trademark of Sequent Computer Systems.

UNIX: A registered trademark of AT&T.

APPENDIX C

PROGRAM LISTING

```
/*****************************************************************
FILE:     thes.c
FUNCTION: main program, create a Hierarchical Task Queue Object.
          Fork processes, one of them create task and add into
          Hierarchical Task Queue.  The rest function as a processor
          to get the task from queue and execute
*****************************************************************/
#include <iostream.h>
#include <memory.h>
extern "C"{                          // from "C" library
    #include <sys/types.h>
    #include <sys/times.h>
    #include <sys/ipc.h>
    #include <sys/shm.h>
    #include <parallel/parallel.h>
    #include <sys/wait.h>
    #include <sys/types.h>
    #include "sem.h"                 // semaphore function written in 'C'
    pid_t fork();                    // system call used to create a UNIX
                                     // process to simulate processors

    pid_t getpid();                  // system call used to get process
                                     // id for the identity of a process,
                                     // used when referring to a
                                     // processor in the simulation program

    void exit(int);                  // system call used to terminate a
                                     // process after finishing execution

    void perror(char *);             // error handling function

    int atoi(char *);                // change ascii input to integer,
                                     // used to change the parameter of
                                     // the program to a number
}

#include "queue.h"                   // Queue Header file which is composed
                                     // of the interface and prototype of
                                     // various queues in the simulation
                                     // program
#define CHILD 0
int Tr;                              // integer used for Transfer Factor
                                     // of Hierarchical Task Queue

int DELAY;                           // integer used to specify the amount of
                                     // iterations to create a delay

main(int argc, char **argv)
```

```
{
    int JOBS;                   // number of jobs to be executed

    int PROCESSOR;              // number of simulated processors
                                // in the system

    int BRANCH;                 // the branching factor of
                                // each Organization

    int pid;                    // Integer for process id used to
                                // tell the identity of the simulated
                                // processors
    int status;
    struct tms buffer1, buffer2;
    clock_t t1,t2;
    clock_t t3,t4;

    PROCESSOR=atoi(argv[1]);    // get the number of processors used
                                // for the simulation program

    BRANCH   =atoi(argv[2]);    // the number of branches of the
                                // tree in the queue organization

    Tr       =atoi(argv[3]);    // the number of transferred jobs from
                                // one queue to another queue (job
                                // transfer factor)

    JOBS     =atoi(argv[4]);    // the number of all input jobs to
                                // be executed

    DELAY    =atoi(argv[5]);    // the amount of job transfer delay

    hier_q *q1=new hier_q(BRANCH,PROCESSOR);
                                // create Hierarchical task
                                // queue for incoming jobs
                                // by specifying how many
                                // processors and branches are needed

    t1=times(&buffer1);         // start timer to record the
                                // execution time
    for(int j=0;j<PROCESSOR+1;j++){
        //if(j==1) sleep(5);
        if((pid=fork())==-1){   // create a process simulating
                                // processors
            perror("fork");     // call the error system call if
                                // something happens that cannot fork
                                // the process
            exit(1);
        }
        if(pid==CHILD){         // check for the process that
                                // functions as task producer
            if(j==0){           // this piece of code is for adding
                                // a task into the task queue organization
               for(int i=0;i<JOBS;i++){
                   Job *f=new Job(i);

                   q1->enqueue(f);
                               // add to q1 which is Hierarchical
               }               // Queue Organization
               for(i=0;i<=PROCESSOR;i++)
                   q1->enqueue(NULL);
                               // send the terminating mark to
                               // processors to notify them that
               exit(1);        // there are no more jobs
```

```
            }else{              // The other processes get the job
                                // from queue and execute
                for(;;){
                    Job* f=q1->dequeue(j);
                                // dequeue job from queue
                                // if null, there is no more
                                // job in the queue
                    if(f==NULL){
                        exit(1);
                    }
                    cout<<"in pid "<<getpid()<<":";
                                // show the identity
                                // of the processor
                    f->execute();
                                // execute job
                    flush(cout);
                    delete f; // remove already executed job from
                                // the  memory
                }
            }
        }
    }
    for(int k=0;k<PROCESSOR+1;k++){
                                // wait for all processors to finish
                                // before determining the execution
                                // time
        pid=wait(&status);
    }

    t2=times(&buffer2);         // determine the time used to
    cout<<t2-t1<<"\n";          // execute input jobs

    delete q1;                  // deallocate queue from memory
                                // after finishing
}
/*************************************************************************
FILE:   list.h
FUNCTION: Contains Job class, which is the class for each task
          simulation, and list class
*************************************************************************/
#ifndef _LIST_
#include <memory.h>
extern "C"{sleep(int);}
class Job{
    private:
        int data;
    public:
        Job(int i){data=i;}
        void *operator new(size_t sz){
                                // each task needs to be accessible
                                // from every process, so the new
                                // operator is overwritten to put
                                // the task into shared memory
            Job *p=(Job *)shmalloc(sz);
            return p;
        }
        void operator delete(void *p){
                                // remove the task from shared
                                // memory after finishing execution
            shfree((char *)p);
        }
        void execute();         // execute the task
};
```

```
struct  slink{                  // each element in the queue list
    Job *thread;                // is composed of a pointer
                                // to a task

    slink* next;                // the pointer to next node

    slink(Job* a){              // constructor with a created job
        thread=a;               // as the first job in the list
        next=this;
    }

    void *operator new(size_t sz){
                                // put the list into shared memory
                                // to make accessible to all processors
        slink *p=(slink *)shmalloc(sz);
        return p;
    }
    void operator delete(void *p){
                                // remove list from shared memory
                                // after finishing
        shfree((char *)p);
    }
};

// the list for the node queues which are not the root
class JobList{
    private:
        slink* last;
    public:
        void insert(slink* a);// insert job at the front for
                                // enqueuing

        void append(slink* a);// append job at the end of the list
        Job* get();             // get a job from the list

        void clear(){ last=NULL; }
                                // clean the list
        JobList() { last=NULL; }
                                // constructor
        JobList(slink* a) { last=a->next=a ; }


        void *operator new(size_t sz){
                                // overwritten new operator to put
                                // the object into shared memory
            JobList *p=(JobList *)shmalloc(sz);
            return p;
        }

        void operator delete(void *p){
                                // overwritten delete operator to clear
                                // shared memory
            shfree((char *)p);
        }
        slink* transfer(int i) ;
                                // move i jobs from list at
                                // one time for transferring jobs
                                // down to the Hierarchical
                                // Organization
};

// The list in the root queue, which allows the task producer and the
// processors, that get tasks from queue to execute, to be able to work
// concurrently.
```

```
class mainList:public JobList{
    private:
        slink *head, *tail;
    public:
        mainList();
        ~mainList();
        void insert(slink&a); // put job in the list in the root
                              // queue
        slink* remove();      // get job from the list in the
                              // root queue

        void *operator new(size_t sz){
                              // put the list into shared memory
                              // to be accessible from all
                              // processors
            mainList *p=(mainList *)shmalloc(sz);
            return p;
        }
        void operator delete(void *p){
                              // remove the list from shared
                              // memory after finishing
            shfree((char *)p);
        }
};
#define _LIST_
#endif

/*************************************************************************
FILE:     list.c
FUNTION:    Contains the list implementation from which the processors
            can insert, append, transfer, and get the job.
*************************************************************************
******/
#include <iostream.h>
#include <memory.h>
extern "C"{                   // from the C library
    #include <sys/types.h>
    #include <time.h>
    #include <sys/ipc.h>
    #include <sys/shm.h>      // for shared memory environment
    #include <parallel/parallel.h>
                              // for parallel system function
    pid_t getpid();
}

#include "list.h"

// Execution instruction for each job inserted in the queue
void Job::execute()
{
    double random(double *seed);
                              // random function to generate
                              // the random number to let the
                              // processor sleep for a random amount
                              // of time simulating computation
    int normalize(double *seed,int lower, int upper);
                              // to make random function
                              // non-uniform
    struct tm *t;             // timer for getting the seed for the
                              // random number generator
    time_t clock;
    double seed;
    int low=0, high=3;        // the lower and upper bound of the result
                              // from random function
```

```
    time(&clock);              // get time for picking seed
    t=localtime(&clock);
    seed=t->tm_sec;

    sleep(normalize(&seed,low,high));
                               // simulating computation by sleeping
                               // for a random amount of time
    cout<<data<<"\n";
}

// The list in the root queue, which allows the task producer and the
// processors, that get tasks from queue to execute, to be able to work
// concurrently.
mainList::mainList()
{
    slink* end=new slink(NULL);
    head=tail=end;
}
mainList::~mainList()          // remove from memory after finishing
{
    delete(tail);
}

void mainList::insert(slink &a)
                               // put the coming job into the list
{
    *tail=a;                   // add node to the end of the list

    slink* end=new slink(NULL);
                               // create space for the next new coming
                               // job

    tail->next=end;            // add to the end of the list
    tail=end;
}

slink* mainList::remove()      // bring job from the list to execute
{
    slink *a;
    if(head->thread==NULL) return NULL;
                               // empty, no job in the list
    else{
        a=head;                // remove the job at the head
        head=head->next;       // and move the pointer to the next
        return a;
    }
}

void JobList::insert(slink* a)// insert job to the end of the list
{
    if(last)                   // if there is already a job in the
        a->next=last->next;    // list, put the coming job next to it
    else
        last=a;                // if there is no job in the list
                               // the coming job is the last
    last->next=a;
}

void JobList::append(slink* a)// append job to the head of the list
{
    if(last){                  // if there is already a job in the
        slink* tmp=a->next;    // list, put the coming job on head
        a->next=last->next;
```

```
        last->next=tmp;
    }
    last=a;
}

slink* JobList::transfer(int JobAmount)
                                // transfer a number of jobs down to the
{                               // Hierarchical Organization queue

    if(last==NULL) return NULL;
                                // empty list
    slink* f=last->next;
    slink* m=last->next;
    if(last->next==last){       // if there is only one job in the
        last=NULL;              // list, return that job
        return m;
    }
    for(int i=1;i<JobAmount;i++){
                                // create a small list of the jobs
        if(m->next==last){      // that are going to be removed
            m=last;             // from the list
            last=NULL;
            return m;
        }else
            m=m->next;
    }
    last->next=m->next;
    m->next=f;
    return m;                   // return the created list
}

Job* JobList::get()             // used in leave node of hierarchical
{                               // task queue to dequeue  the job and
                                // execute

    if(last==NULL) return NULL;
                                // empty list, quit
    slink* f=last->next;        // get the node from the list
    if(f==last)                 // adjust list after the dequeue
        last=NULL;
    else
        last->next=f->next;
    Job* i=f->thread;           // prepare the thread in the node
                                // to be executed
    delete f;                   // delete the node
    return i;                   // return the thread to the caller to
                                // be executed
}

/*************************************************************************
FILE:     queue.h
FUNCTION: Contains the declaration of the queue structure.
*************************************************************************/
#ifndef _QUEUE_
#include "list.h"               // list interface
extern int Tr ;                 // transfer job factor from command line
extern int DELAY ;              // delay access time from command line

// base class of queue to collect the
// common characteristic between root queue
// and the other queues in the Hierarchical
// task queue
class baseQueue{
    protected:
```

```
        static int count;       // number of queues in the organization
        class baseQueue *parent;
                                // parent of each node in the
                                // Hierarchical task queue
        int semid;              // semaphore id for locking each
                                // queue access the number of
                                // processors below this queue
        int ProcBelow;
    public:
        void*    operator new(size_t sz);
                                // make the queue accessible to
                                // all processor by putting itself
                                // into shared memory
        void     operator delete(void *p);
                                // remove from shared memory after
                                // finishing
        void     SetProcBelow(int proc){ProcBelow=proc;};
                                // set the number of processors below
                                // this queue

        virtual  slink* transfer()=0;
                                // transfer job to the child queue or
                                // processor in the case if they
                                // have no job to be executed
};

// class of the queue that is not at the root of
// task queue organization
class queue:public baseQueue{
    int sem_key;                // key to access the semaphore
    JobList *list;              // list of jobs in each node of the queue
    public:
        queue(baseQueue *papa);
        ~queue();
        slink* transfer();      // transfer a number of jobs
                                // from the queue
                                // this number is used if the
                                // queue is non-leaf node
        Job*  GetJob();         // remove only one job from the
                                // queue, this is used if the queue
                                // is the leaf node
};

//Class of the root queue of the hierarchical task queue
class rootQueue:public baseQueue{
    int sem_key1;               // two semaphores for the concurrent job
    int sem_key2;               // addition and job removal
    mainList *list;             // list of jobs in the root queue
    public:
        rootQueue();
        ~rootQueue();
        void AppendJob(Job*a);// insert job into this queue
        slink* transfer();      // remove many jobs down to the
                                // hierarchical task queue
};

// Hierarchical Task Queue Organization
class hier_q{
    private:
        int sem_id;
        class baseQueue **q_array;
                                // array of all child queues in
                                // the organization
        class baseQueue **q_access;
```

```
                                // array of construction of queue
        int leaf;               // keep the number of processors in the
                                // system which are at the leaf node
        int sub_queue;          // the number of queues in the
                                // organization
        int dept(int,int);      // function to determine the depth
                                // of the organization
        int proc_numb(int);     // function to determine the number of
                                // processors in the organization
        int number_queue(int);// function to determine the number of
                                // queues in the organization
        int create_hier(baseQueue *,int);
                                // create hierarchical queue
    public:
        hier_q(int,int);
        ~hier_q();
        void* operator new(size_t sz);
                                // make the queue organization
                                // from processors
        void  operator delete(void *p);
                                // remove the queue organization
                                // from shared memory after finishing
        Job*  dequeue(int);     // insert job into queue organization
        void  enqueue(Job *); // remove job from queue organization
};
#define _QUEUE_
#endif
/*************************************************************************
FILE:     queue.c
FUNCTION: Contains the implementation of queue.
*************************************************************************/
#include <iostream.h>
#include <memory.h>
#include <math.h>
extern "C"{                     // from C library
    #include <sys/types.h>
    #include <sys/ipc.h>        // interprocess functions
    #include <sys/shm.h>        // shared memory functions
    #include <parallel/parallel.h>
                                // some shared memory functions
    #include <sys/wait.h>
    #include "sem.h"            // semaphore functions
    void exit(int);
    pid_t getpid();             // get process id
}
#include "queue.h"

// overwritten new operator to put the object into the shared memory
void* baseQueue::operator new(size_t sz)
{
    baseQueue *p=(baseQueue *)shmalloc(sz);
    return p;
}

// overwritten delete operator to remove the object from the shared
// memory
void baseQueue::operator delete(void  *p)
{
    shfree((char *)p);
}
// initialize the number of queues to zero before counting the number of
// node, on which we use recursive function, into the queue
int baseQueue::count=0;
```

```
// constructor of each queue in the hierarchical task queue organization
// which is not at the root
queue::queue(baseQueue *papa)
{
    parent=papa;
    list=new JobList;
    sem_key=count++;
    semid=sem_open((key_t)54041);
                                // open the lock to be able to
                                // use semaphore
}

// destructor of each queue in hierarchical task queue organization
// which is not at the root
queue::~queue()
{
    delete list;
}

// return job from this queue
Job* queue::GetJob()
{
    sem_wait(semid, sem_key); // lock the queue first so that there
                                // is no other processor to be able
                                // to access this queue
    Job *thread=list->get();  // get job from the list member
    if(thread==NULL){         // if the list member is empty,
                                // transfer jobs from the parent queue
        slink* f=parent->transfer();
        if(f==NULL){          // if still empty, that means no
                                // more job to execute, so exit
            sem_signal(semid,sem_key);
            exit(1);
        }
        list->append(f);      // append the list of jobs from parent to
                                // local list
        thread=list->get();   // re-dequeue the job from list
    }
    sem_signal(semid,sem_key);// release lock
    return thread;
}

// remove a number of jobs from queue for transferring to the
// child queue below
slink* queue::transfer()
{
    int i=0;
    sem_wait(semid,sem_key);  // lock the queue first so that no
                                // other processors can access this queue
    slink* f=list->transfer(ProcBelow*Tr);
                                // get a number of jobs to
                                // be transferred down the tree
    while(i<DELAY) i++;       // delay access to the  queue
    if(f==NULL){              // if the local list is empty,
                                // transfer again from its parent node
        if(parent==NULL) f=NULL;
                                // if this queue is at root,
                                // that means all jobs are executed
        else{
            slink* g=parent->transfer();
                                // transfer jobs from its parent
            if(g!=NULL){
                list->append(g);
                                // append to local list
```

```
                f=list->transfer(ProcBelow*Tr);
                                // prepare list of jobs for its child
            }
        }
    }
    sem_signal(semid,sem_key);// unlock the queue to let other
                                // processors access the queue
    return f;                   // return the list of jobs to its child
}

// constructor for the queue at the root of hierarchical task queue
rootQueue::rootQueue()
{
    parent=NULL;                // no parent, so it is set to zero
    list=new mainList;          // create local job list
    sem_key1=count++;           // reserve semaphore used for the lock
    sem_key2=count++;
    semid=sem_open((key_t)54041);
    sem_wait(semid, sem_key1);// initialize to zero because there is
                                // no task at the beginning
}

// destructor for the queue at the root of hierarchical task queue
rootQueue::~rootQueue()
{
    delete list;
}
// move jobs from the root queue down to the tree to child queue
slink* rootQueue::transfer()
{
    slink *f,*task;
    int i=0;
    sem_wait(semid, sem_key2);// mutual exclusion among processors
    /*if(parent==NULL)*/ while( i<DELAY) i++;
    for(i=0;i<ProcBelow*Tr;i++){
        sem_wait(semid, sem_key1);
                                // wait till there is a job in
                                // the root queue
        task=list->remove();  // remove a job from the root queue
        if(i==0){
            if(task==NULL){   // no job in root queue
                f=NULL;
                break;
            }else f=task;     // create head of the jobs list
        }else{
            if(task==NULL)    // only a few jobs in the root queue
                break;        // not as many as expected to be
                                // transferred
            else task->next=f->next;
                                // add a job to the existent list
        }
        f->next=task;
        f=task;
    }
    sem_signal(semid, sem_key2);
                                // mutual exclusion among processors
    return f;
}
// enqueue incoming job to the root queue or to the hierarchical task
queue
void rootQueue::AppendJob(Job* a)
{
    slink* f=new slink(a);    // create the node containing the
                                // incoming job
```

```
    list->insert(*f);           // insert created node to the local list
    sem_signal(semid,sem_key1);
                                // signal the semaphore to notify the
                                // consumer processors
}

// Hierarchical Task Queue Organization
hier_q::hier_q(int branch,int processor)
{
    leaf=processor;
    sub_queue=number_queue(branch);
                                // determine the number of queue needed
                                //  to be created
    sem_id=sem_create((key_t)54041,1,sub_queue+1);
                                // create lock


    q_array=(baseQueue **)shmalloc(sizeof(baseQueue *)*sub_queue);
                                // allocate array of queues in
                                // the organization to be easy to refer to
    q_access=(baseQueue **)shmalloc(sizeof(baseQueue
*)*proc_numb(branch));

    create_hier(NULL,branch); // create Hierarchical Task queue
}

// Destructor
hier_q::~hier_q()
{
    for(int i=0;i<sub_queue;i++)
                                // deallocate all queue of task
        delete q_array[i];      // task queue organization
    sem_rm(sem_id);             // free the semaphore
    shfree((char *)q_array);    // free the space used to keep
    shfree((char *)q_access);   // reference of queues
}

// overwritten new operator to put object into the shared memory
// in order to be accessible from all processors
void* hier_q::operator new(size_t sz)
{
    hier_q *p=(hier_q *)shmalloc(sz);
    return p;
}

// overwritten delete operator to remove object from the shared memory
void hier_q::operator delete(void *p)
{
    shfree((char *)p);
}

// check depth of the tree to help calculating the number
// of nodes in the hierarchical task queue
int hier_q::dept(int number,int branch)
{
    int count=0;                // depth initially is 0
    int amount=1;
    while(amount<number){       // loop until the amount is
        amount=amount*branch;   // greater than number which is the number
        count++;                // of leaves in the tree
    }
    return count;               // return the depth
}
```

```
// calculate the number of leaf nodes of the hierarchical task queue
// to determine the number of processors that can access the queue
int hier_q::proc_numb(int branch)
{
    int i;
    static int key=0;
    static int a=0;
    static int count=0;

    if(a==dept(leaf,branch)){ // this shows that the recursive function
                              // reaches the leaf node
       count++;               // increment the number of leaf nodes
       return count+1;
    }
    for(i=0;i<branch;i++){
        a++;
        proc_numb(branch);    // recursive until the function reaches
                              // the leaf node
        a--;
    }
    return count;             // return the number of leaf nodes
}
// create Hierarchical Organization
int hier_q::create_hier(baseQueue *parent,int branch)
{
    int i;
    static int key=0;
    static int a=0;
    static int count=0;

    baseQueue *current;

    int deep=dept(leaf,branch);
    if(parent==NULL){
        q_array[key++]=current=new rootQueue();
                              // create the queue which
                              // functions as the root queue in
                              // the hierarchical task queue
        q_access[0]=current;  // put the root to the array so
                              // the task we can producer processor can
                              // access the root queue
    }else
        q_array[key++]=current=new queue(parent);
                              // create the queue which
                              // function as queue in each
                              // node in the hierarchical task
                              // queue
    current->SetProcBelow((int)pow(branch,deep-a)/branch);
    if(a==deep){
       q_access[1+count++]= current;
       return count+1;
    }
    for(i=0;i<branch;i++){
        a++;
        create_hier(current,branch);
                              // recursively create queues in
                              // the hierarchical task queue
        a--;
    }
    return count+1;
}

// calculate the number of queues in the organization
// to determind how many queues have to be created
```

```
int hier_q::number_queue(int branch)
{
    int count=0,i;
    int d;
    d=dept(leaf,branch);        // determind the depth of the
                                // hierarchical task queue
    for(i=0;i<=d;i++){
        count+=(int)pow(branch,i);
    }
    return count;
}

// put the job into the Hierarchical Task Queue Organization
void hier_q::enqueue(Job *f)
{
    rootQueue* temp;
    temp=(rootQueue*)q_access[0];
    temp->AppendJob(f);
}

// remove the job from the Hierarchical Task Queue Organization
Job* hier_q::dequeue(int id)
{
    queue* temp;
    temp=(queue*)q_access[id];
    return temp->GetJob();
}

/*************************************************************************
FILE:     sem.h
FUNCTION: contains a prototype of the semaphore function to be used
          in the simulation program
*************************************************************************/
int sem_create(key_t, int, int);
                                /* create semaphore           */
int sem_open(key_t);            /* open the semaphore which is*/
                                /* already create             */
int sem_rm(int);                /* remove semaphore           */
int sem_close(int);             /* close the semaphore after  */
                                /* finishing use              */
int sem_wait(int,ushort);       /* lock semaphore             */
int sem_signal(int,ushort);     /* unlock semaphore           */
int sem_op(int   , int);


/*************************************************************************
FILE:       sem.c (adapted from [Steven 90])
FUNCTION:   Contains a semaphore implementation to used as a lock
            mechanism in the simulation program
*************************************************************************
**/
#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

#include    <errno.h>
extern int  errno;

#define     BIGCOUNT    10000 /* initial value of process counter */

/*
 * define the semaphore operation arrays for the semop() calls
 */
```

```
static struct sembuf    op_lock[2] = {
      2, 0, 0,                  /* wait for [2] (lock) to equal 0   */
      2, 1, 0                   /* then increment [2] to 1 - this   */
                                /* locks it                         */
} ;
static struct sembuf    op_endcreate[2] = {
      1, -1,0,                  /* decrement [1] (proc counter) with*/
                                /* undo on exit */
      2, -1,0                   /* then decrement [2] (lock) back to 0 */
};

static struct sembuf    op_open[1] = {
      1, -1, 0                  /* decrement [1] (proc counter) with */
                                /* undo on exit */
};

static struct sembuf    op_close[3] = {
      2, 0, 0,                  /* wait for [2] (lock) to equal 0 */
      2, 1, 0,                  /* then increment [2] to 1 - this */
                                /* locks it */
      1, 1, 0                   /* then increment [1] (proc counter) */
};

static struct sembuf    op_unlock[1] = {
      2, -1, 0                  /* decrement [2] (lock) back to 0 */
};

static struct sembuf    op_op[1] = {
      0, 99, 0                  /* decrement or increment [0] with */
                                /* undo on exit */
                                /* the 99 is set to the actual amount*/
                                /* to add or subtract (positive or */
                                /* negative) */
};

/*************************************************************************
 * Create a semaphore with a specified initial value.
 * If the semaphore already exists, we don't initialize it (of course).
 * We return the semaphore ID if all OK, else return -1.

*************************************************************************/

int
sem_create(key, initval, number)
key_t key;
int   initval;                  /* used if we create the semaphore */
int     number;
{
        int i;
      register int              id, semval;
      union semun {
            int         val;
            struct semid_ds   *buf;
            ushort            *array;
      } semctl_arg;

      if (key == IPC_PRIVATE)
            return(-1);         /* if key=IPC_PRIVATE, */
                                /* it will create a private semaphore */
      else if (key == (key_t) -1)
            return(-1);         /* value of key error */

again:
```

```
        if ( (id = semget(key, number, 0666 | IPC_CREAT)) < 0)
                return(-1);             /* permission problem or tables full */

          semctl_arg.val = initval;
          for(i=0;i<number;i++){
              if (semctl(id, i, SETVAL, semctl_arg) < 0)
                  perror("can SETVAL[0]");
          }
        return(id);
}

int
sem_open(key)
key_t key;
{
        register int            id;

        if (key == IPC_PRIVATE)
                return(-1);             /* if key=IPC_PRIVATE,                  */
                                        /* it will create a private semaphore */

        else if (key == (key_t) -1)
                return(-1);             /* value of key error */

        if ( (id = semget(key, 0, 0)) < 0)
                return(-1);             /* doesn't exist, or the system */
                                        /* resource full                */

        return(id);
}

/*************************************************************************
 * Remove a semaphore.
 * remove semaphore from the system

*************************************************************************/

sem_rm(id)
int     id;
{
        if (semctl(id, 0, IPC_RMID, 0) < 0)
                                        /* use IPC_RMID to remove */
                perror("can't IPC_RMID");
}

/*************************************************************************
 * Close a semaphore.
 * this function used after we finish using it but do not want to remove
 * it from the system because some other process still use it

*************************************************************************/

sem_close(id, number)
int     id;
int       number;
{
        register int            semval;
          int i;
        sem_rm(id);
}

/*************************************************************************
 * Wait until a semaphore's value is greater than 0, then decrement
 * it by 1 and return.
```

```
 * Dijkstra's P operation.  Tanenbaum's DOWN operation.

*************************************************************************
/

sem_wait(id, Sem_No)
int   id;
ushort  Sem_No;
{
      sem_op(id, -1,Sem_No);
}

/*************************************************************************
 * Increment a semaphore by 1.
 * Dijkstra's V operation.  Tanenbaum's UP operation.

*************************************************************************/

sem_signal(id, Sem_No)
int   id;
ushort  Sem_No;
{
      sem_op(id, 1,Sem_No);    /*add 1 to the semaphore with id=Sem_No */
}

/*************************************************************************
 * General semaphore operations.  Increment or decrement by a
 * user-specified  amount (either positive or negative;
 * the amount can't be zero).
 *************************************************************************/

sem_op(id, value, Sem_No)
int   id;
int   value;
ushort  Sem_No;
{
        int temp;
        char s[80];
      if ( (op_op[0].sem_op = value) == 0)
                              /* if the value=0 that */
                              /* means no operation  */
            perror("can't have value == 0");
        op_op[0].sem_num=Sem_No;
                              /* select the semaphore */
      if (semop(id, &op_op[0], 1) < 0)
                              /* do the operation */
                              /* by having the id as an */
                              /* index specifying whether */
                              /* wait or signal operation */
            perror(s);
}
```

VITA

WERAKUL LAOWORAKIAT

Candidate for the Degree of

Master of Science

Thesis: COMPARATIVE STUDY OF TASK QUEUE ORGANIZATIONS

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangkok, Thailand, September 24, 1968, son of Sunseng and Supawadee Laoworakiat.

Education: Graduated from Benjamabophit High School, Bangkok, Thailand in June 1986; received Bachelor of Science degree in Computer Engineering from Kasetsart University, Bangkok, Thailand. Completed the requirements for the Master of Science degree at the Computer Science Department at Oklahoma State University in July 1996.

Professional Experience: Computer Programmer, International Software Factory Co., Ltd., Bangkok, Thailand, April 1991-July 1992.