Distributed Dynamic Scheduling of Composite Tasks on Grid Computing System

by

Hongtu Chen

A thesis

Submitted to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

Department of Electrical & Computer Engineering

University of Manitoba

Winnipeg, Manitoba, Canada

To My Parents…

Abstract

Network computing attracts the attention of many computer researchers and scientists because it can better utilize existing computing resources. The key challenge of network computing is the search for the best method to distribute computing resources to submitted tasks. This thesis demonstrates a distributed dynamic scheduling of composite tasks on a grid computing system. It describes how a computer program was written to simulate a real world computer network.

Submitted tasks consist of subtasks represented by DAGs. The adopted scheduling and mapping include two steps: one external and the other internal. External scheduling and mapping are performed on the task level, and internal scheduling and mapping are done on the subtask level. A task and its subtask must go through these two steps to be allocated computing resources.

This research analyzes different factors on the distributed dynamic scheduling algorithm. The factors include Subtask Waiting Queue size, submitted task number, task submission interval, and network infrastructure. The percentage of tasks completed before deadline and average response times are used as indexes of network computing performance.

Acknowledgement

I would like to use this opportunity to thank my advisors, Dr. Muthucumaru Maheswaran and Dr. Robert McLeod, for their great guidance through my academic studies and this research. It would have been impossible to finish this project without their generous help. Great appreciation to my family, for their unconditional love and always being behind me.

Table of Content

List of Figure

List of Table

1 Introduction

Heterogeneous computing changes a network of heterogeneous computers into a single computing resource entity. The central theme of heterogeneous computing is to utilize computing resources of different machine architectures. On one hand, many users find that the computers they use are not powerful enough to meet their purposes; on the other hand, many of the computers in a typical network are idle, having no job to process. This situation happens within a LAN, or even WAN-wide. Ideally, if computing resources can be shared, it could dramatically increase our work efficiency.

The greatest challenge to network computing is to obtain a near-optimal algorithm to solve the mapping and scheduling problem. Several characteristics should be considered: the dynamic nature of computer traffic loading, the intensity of task sub missions, the infrastructure of the computer network, and the fair competition for utilizing computational resources.

This research paper presents an analysis of distributed dynamic scheduling of composite tasks on a grid computing system environment. The examined network infrastructure consists of a WAN composed of many LANs. Multiple tasks are submitted. Each task has a Directed Acyclic Graph (DAG) structure representing subtasks and data dependences between them. A task is scheduled based first on an immediate mode external mapping scheduling algorithm, and then on a batch mode internal mapping and scheduling algorithm.

External scheduling determines in which LAN the task is to be executed. It views a LAN as a signal computing entity, and views each task as independent and undividable (no subtasks). After a task arrives in a LAN, subtasks are mapped to different computer nodes for parallel computing. The mapping and scheduling of subtasks are managed by internal scheduling.

This thesis is organized as follows: In Chapter 2, an overview of the related literature is presented, and recent technological advances that justify our research are explained. In Chapter 3, details of scheduling and mapping algorithms adopted in this research are discussed. In Chapter 4, the design and implementation of the simulation program is presented. Chapter 5 shows the experimental results of the simulation and discusses them. The conclusion is stated in Chapter 6.

2 Literature Review

2.1 Heterogeneous Network Environments

Recent advances in software and hardware technology have greatly improved the performance of a Network of Workstations (NOW). Very often in a NOW environment, machines are owned by individual users whose typical processing needs rarely require the full capacity of their workstation. Conversely, some users may have computationally intensive tasks that are beyond the capacity of the workstation he or she owns. Consequently, if each user were restricted to running tasks within the boundaries of a single workstation, precious computational resources would be wasted. This raises the challenge of developing a load-balancing environment to utilize available computational resource more efficiently.

The nature of a connected workstation network is heterogeneous. Heterogeneity takes a number of forms:

1) Heterogeneity of a configuration, whereby hosts may have different processing power, memory space, disk storage, and so on;
2) Architectural heterogeneity, that makes it impossible to execute the same code on different hosts;
3) Operating system heterogeneity, where hosts have different operating systems running and may be incompatible [ZhW92].

However, for this research paper, only the heterogeneity of a configuration was considered, in which we assume that a task can be executed on any computer node in the NOW.

Besides heterogeneity, a NOW system has three other unique features in comparison with a multiprocessor or a multicomputer system:

1) Low bandwidth communication: Even when high-speed networks are used, the inter-node communication still causes bottleneck problems. Therefore, only coarse-grained or medium-grained parallel tasks are suitable for running on a NOW.
2) Random network topology: A NOW system connects workstations in a random way, and its topology may change from time to time in practice.
3) Multidirectional scaling: A NOW system can be scaled in three directions: by increasing the number of workstations, by upgrading the power of the workstations, and by a combination of the two [Du].

2.2 DAG Model

In this thesis, we define a task as an independent, computationally intensive application sent by different users. A parallel task can be divided into subtasks with data dependence between them. By the loop-unraveling technique, computational loops can be subdivided into a number of subtasks. Usually a large class of data-flow computation problems and many numerical algorithms (such as matrix multiplication) do not have conditional

branches or indeterminism in the program, thereby making them suitable candidates for subdivision. In addition, in many numerical tasks, such as Gaussian elimination or fast Fourier transforms (FFT), the loop bounds are known during compile-time. As such, one or more iterations of a loop can be deterministically encapsulated in a subtask. [KwA99]. These techniques made parallel processing of a task possible.

Based on the discussion above, a parallel task can be represented by a Directed Acyclic Graph (DAG), which is illustrated in Figure 1. In a DAG, *V* is a set of *v* nodes and *E* is a set of *e* directed edges. *G=(V, E)*, where the set of vertices $V=\{v_1, v_2, \ldots v_n\}$ represents the set of subtasks to be executed, and the set of weighted, directed edges E represents communication between subtasks. A node in the DAG represents a subtask that is a set of instructions that must be executed sequentially without preemption in the same processor. The weight of a node is computation cost. The edges in the DAG correspond to the communication messages and precedence constraints among the nodes. The weight of an edge is referred as communication cost. Thus $e_{ij} = (v_i, v_j) \in E$ indicates communication from subtask v_i to v_j, and $|e_{ij}|$ represents the volume of data sent between these subtasks. The node- and edge-weights are usually obtained by estimation using profiling information of operations such as numerical operations, memory access operations, and message-passing primitives. In a DAG, the source node of an edge is called the parent node while the sink node is called the child node. A node with no parent is called an entry node and a node with no child is called an exit node. As shown in Figure 1, *N2* is the parent of *N4* and *N5*, *N4* and *N5* are the child nodes of *N2*. *N1* is the entry node, and *N8*

and *N8* are exit nodes, and the line in bold is the crucial path of the task. [KwA99][IvÖ98].

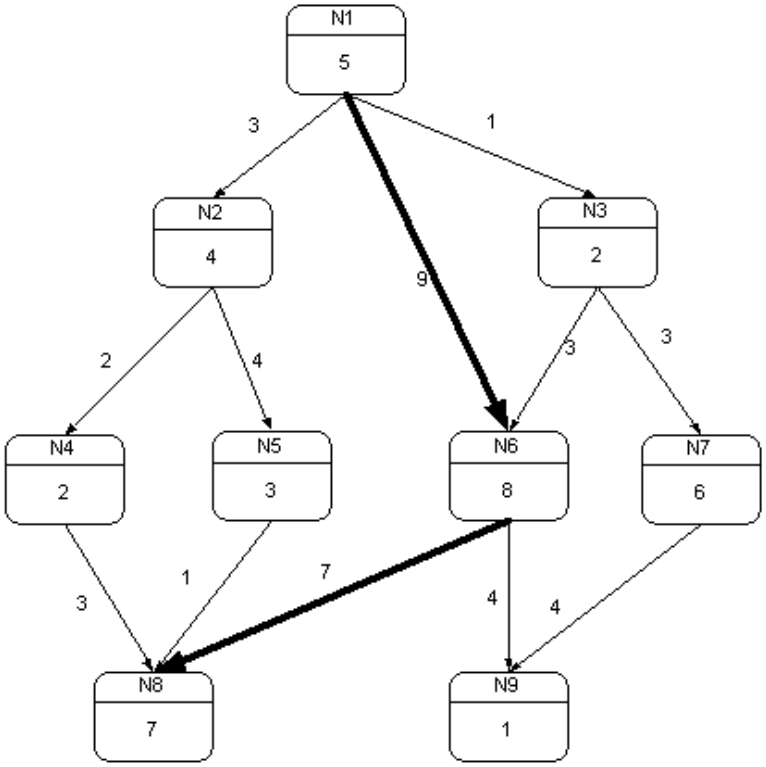


Figure 1. A Task DAG Graph

Subtask processing can either be preemptive or non-preemptive. After a node has been selected for execution, non-preemptive subtask processing dictates that the subtask cannot be moved even if a more suitable node is available. In contrast, preemptive processing entails stopping the process, moving the subtask to the new node, and resuming its execution. Preemptive processing is much more costly than a non-preemptive transfer in two senses: First, the implementation and maintenance of the

mechanisms necessary to encapsulate, transfer, and resume execution from this complex state are expensive. Second, since preemptive processing causes an overhead that is much greater than that of the non-preemptive variety, it is not obvious what performance improvement might result beyond non-preemptive processing [KrB93]. For this thesis, a non-preemptive DAG represents a subtask structure that assumes that once a subtask starts on a machine, it cannot be stopped. If it is stopped for some unexpected reason, like machine failure, it has to be restarted again.

2.3 Mapping and Scheduling Algorithm

The problem of mapping and scheduling multiple tasks can be divided into two categories: task mapping and scheduling, and subtask mapping and scheduling. In task mapping and scheduling, independent tasks are scheduled among the network of workstations to optimize overall system performance. In contrast, the subtask scheduling and mapping problem requires the allocation of multiple interacting subtasks of a single parallel task in order to minimize the completion time. Task scheduling usually requires dynamic run-time scheduling because it is not a priori decidable, the subtask mapping and scheduling problem can be addressed both statically and dynamically [KwA99]. In this thesis, a multiple task computing simulation in a heterogeneous environment is used, therefore both task and subtask mapping and scheduling are addressed.

2.3.1 Task Level Mapping and Scheduling

Task level mapping and scheduling considers a scenario where each task is independent, and there is no communication between them. Those independent tasks compete for computational resources, and the task level mapping and scheduling heuristics attempt to match these tasks with available computational entities.

The task mapping heuristics can be grouped into two categories: dynamic heuristics and static heuristics. Dynamic heuristics can be further grouped into two categories: immediate mode and batch mode heuristics. In the immediate mode, a task is mapped as soon as it arrives. In the batch mode, tasks are not mapped as they arrive; instead they are collected into a set that is examined for mapping at prescheduled times called mapping events. The independent set of tasks that are considered for mapping at the mapping events is called a meta-task. A meta-task can include newly arrived tasks (i.e., the ones arriving after the last mapping event) and the ones that were mapped in earlier mapping events but did not begin execution. While immediate mode heuristics consider a task for mapping only once, batch mode heuristics consider a task for mapping at each mapping event until the task begins execution. For immediate mode there is no mapping delay between mapping events, the tasks are mapped right after they arrive. However, as a tradeoff, since immediate mode can only map tasks once, its performance is not as good as batch mode when the arrival of tasks is very intensive [MaA99] [SiS00].

There are five different types of immediate mode heuristics. These are 1) minimum completion time (MCT); 2) minimum execution time (MET); 3) switching algorithm (SA); 4) k-percent best (KPB); and 5) opportunistic load balancing (OLB). The MCT heuristic assigns each task to the machine that results in that task's earliest completion time in order to balance the load. The MET heuristic assigns each task to the machine that performs that task's computation in the least amount of execution time. The MET heuristic can potentially create load imbalance across machines by assigning many more tasks to some machines than to others. The SA heuristic is a combination of MCT and MET. The idea behind it is that when the tasks are arriving in a random mix, it is possible to use the MET, at the expense of load balancing until a given threshold, and then use the MCT to smooth the load across the machines. SA uses the MCT and MET heuristics in a cyclical fashion depending on the load distribution across the machines. The purpose is to have a heuristic with the desirable properties of both the MCT and the MET. The KPB heuristic is another form of a combination of MET and MCT. The heuristic considers only a subset of machines while mapping a task. The subset is formed by picking the $m \times (k/100)$ best machines based on the execution times for the task, where $100/m \leq k \leq 100$. The task is assigned to a machine that provides the earliest completion time in the subset. If $k=100$, then the KPB heuristic is reduced to the MCT heuristic. If $k=100/m$, then the KPB heuristic is reduced to the MET heuristic. The OLB heuristic is very simple; it assigns a task to the machine that becomes ready next, without considering the execution time of the task onto that machine. If multiple machines become ready at the same time, then one machine is arbitrarily chosen. [MaA99] [SiS00].

Three batch mode heuristics are presented here: (i) the Min-min heuristic, (ii) the Max-min heuristic, and (iii) the Sufferage heuristic. The Min-min heuristic is archived by executing following step:

1) For each task find the earliest completion time and the machine that obtains it.
2) Within these earliest completion times, find the minimum, map the task to the machine.
3) Update computational entity free time.
4) Repeat step 1, 2, and 3 until all tasks are mapped.

The Max-min heuristic is similar to the Min-min heuristic. It differs from the Min-min heuristic in step 2, which instead of finding the minimum the Max-min heuristic is to find the maximum. The Max-min is likely to do better than the Min-min heuristic in cases where there are many more shorter tasks than longer tasks. The Sufferage heuristic is based on the idea that better mappings can be generated by assigning a machine to a task that would "suffer" most in terms of expected completion time if that particular machine is not assigned to it [MaA99][SiS00].

In contrast to dynamic task mapping heuristics, static heuristics perform task mapping in a statically (i.e., off-line, or a predictive manner). Static heuristics assume all tasks are known before they are mapped. The static OLB (opportunistic load balancing) heuristic is similar to its dynamic counterpart except that it assigns tasks in an arbitrary order, instead of order of arrival. The UDA (user directed assignment) heuristic works in the same way as the MET heuristic except that it maps tasks in an arbitrary order instead of order of arrival. The fast greedy heuristic is the same as the MCT, except that it maps

tasks in an arbitrary order instead of their order of arrival. The static Min-min heuristic works in the same way as the dynamic Min-min, except a meta-task contains all the tasks in the system. The static Max-min heuristic works in the same way as the dynamic Max-min, except a meta-task has all the tasks in the system. The greedy heuristic performs both the static Min-min and static Max-min heuristics, and uses the better of the two solutions [BrS01].

2.3.2 Subtask Level Mapping and Scheduling

Subtask level mapping and scheduling, also referred as DAG mapping and scheduling, considers a scenario where each subtask is related, and there is data dependence between them. These related subtasks compete for computational resources, and the subtask level mapping and scheduling heuristics are to match these tasks with available computational entities and increase overall system performance and computational usage.

In DAG scheduling, the target system is assumed to be a network of workstations, each of which is composed of a processor and a local memory unit; they do not share memory and communication between them relies solely on message-passing. The processors may be heterogeneous or homogeneous. However, DAG scheduling assumes every module of a parallel program can be executed on any workstation even though the completion times on different processors may be different. The workstations are connected by an interconnection network with a certain topology. The topology may be fully-connected or of a particular structure such as a hypercube or mesh [Y. Kwok 99].

Subtask mapping and scheduling algorithms exist in two forms: static and dynamic. As mentioned, a parallel task can be represented by a DAG. In static scheduling, which is usually done at compile time, the characteristics of a task (such as subtask processing times, communication, data dependencies, and synchronization requirements) are known before program execution. In dynamic scheduling, a few assumptions about the task can be made before execution, and thus, scheduling decisions have to be made on-the-fly. Dynamic schedulers usually can offer better performance, but the goal of a scheduling algorithm includes not only the minimization of the program completion time but also the minimization of the scheduling overhead. A dynamic approach, in contrast to static scheduling, can increase the time-complexity of the scheduling algorithm [Y. Kwok 99].

Most scheduling algorithms are based on the list scheduling techniques. The basic idea of list scheduling is to make a scheduling list (a sequence of subtasks for scheduling) by assigning them some priorities, and then schedule those subtasks according to their priorities [Y. Kwok 99].

Two frequently used attributes for assigning priority are the *t-level* (top level), *b-level* (bottom level), and p-level (partial level). The *t-level* of a node is the length of a longest path (there can be more than one longest path) from an entry node to the node itself (excluding itself). Here, the length of a path is the sum of all the node and edge weights along the path. The *b-level* of a node is the length of the longest path (there can be more than one longest path) to an exit node. Some scheduling algorithms do not take into

account the edge weights in computing the b-level. In such a case, the b-level does not change throughout the scheduling process. This algorithm is referred to as the static b-level. The p-level of a node is simply the computation cost of that given node; also, the p-level does not change throughout the scheduling process. [KwA99][MaS99]. As it is illustrated in Figure 1, a list of t-levels and b-levels are shown in Table 1.

Table 1. t-levels, b-levels, and p-levels for the DAG of Figure 1

Node	t-level	b-level	p-level
N 1	0	36	5
N 2	8	19	4
N 3	6	18	2
N 4	14	12	2
N 5	16	11	3
N 6	14	22	8
N 7	11	11	6
N 8	26	7	7
N 9	29	1	1

Different algorithms use the t-level and b-level in different ways. Some algorithms assign a higher priority to a node with a smaller t-level while some algorithms assign a higher priority to a node with a larger b-level, or a larger p-level. Still some algorithms assign a higher priority to a node with a larger (b-level – t-level). In general, scheduling in a descending order of *b-level* tends to schedule critical path nodes first, while scheduling in an ascending order of *t-level* tends to schedule nodes in a topological order. The composite attribute *(b-level – t-leve*l) is a compromise between the previous two cases. The notion behind the p-level was that by executing higher computationally intensive subtasks first, the overall completion time of the task may be minimized [KwA99][MaS99].

List scheduling includes both static list scheduling and dynamic list scheduling. In static list scheduling, the scheduling list is statically constructed before node allocation begins, and most importantly, the sequencing in the list is not modified. A task is usually scheduled on the processor that gives the earliest start time for the given task. Thus, at each scheduling step, the task is selected first, then its destination processor. The procedure of static list scheduling entails repeatedly executing the following two steps until all the nodes in the graph are scheduled: 1) removing the first node from the scheduling list; 2) allocating the node to a processor which allows the earliest start-time. Dynamic list scheduling takes a different approach. After each allocation, the priorities of all unscheduled nodes are re-computed, and consequently the scheduling list is then rearranged. In this case, the tasks do not have a pre-computed priority. At each scheduling step, each ready task is tentatively scheduled to each processor, and the best task-processor pair is selected. Both the task and its destination processor are selected at the same time. Thus, these algorithms essentially employ the following three-step approaches: 1) determining new priorities of all unscheduled nodes; 2) selecting the node with the highest priority for scheduling; 3) allocating the node to the processor that allows the earliest start-time or earliest finish-time. Scheduling algorithms that employ this three-step approach can potentially generate better schedules, but the tradeoff is the scheduling time is increased [KwA99][RaG00].

Both static and dynamic approaches of list scheduling have their advantages and drawbacks in terms of the schedule quality they produce. Static approaches are better

suited for communication-intensive and irregular problems, where selecting important tasks first is more crucial. Dynamic approaches are better suited for computationally intensive applications with a high degree of parallelism, because these algorithms focus on obtaining good processor utilization [RaG00].

.

2.3.3 Multiple Task Mapping and Scheduling

So far, we have discussed task mapping and scheduling, and signal DAG mapping and scheduling. In this thesis, we analyze the behavior of multiple task (multiple DAG) computing in a heterogeneous environment, therefore the objective of this research is to study multiple DAG scheduling. However, there is little literature in this area.

Iverson presents a dynamic, competitive scheduling of multiple DAGs [IvÖ98]. In his framework, each task is responsible for scheduling its own tasks. Thus, there is no centralized scheduling authority. A task is scheduled without the knowledge of other tasks; the task scheduler only knows the current work loading of the network. Iverson's algorithm is based on the expectation that if each task had the best mapping and scheduling possible, the overall parallel computing performance would be optimal.

3 Scheduling and Matching Algorithm

This chapter presents a detailed discussion of the scheduling and mapping algorithm. We start with the definition and assumptions of the problem, and then we move on to a detailed discussion of the algorithms under consideration.

3.1 Problem Definition and Assumption

Consider a WAN consisting of several LANs. All LANs in the WAN reach an agreement to share their computing resources. In addition, the computer network nodes of each LAN are also resource sharing. This makes it is possible that a subtask *i* of task *j* submitted from computer network node *w* in LAN *x* is executed on computer network node *y* of LAN *z*.

A task consists of a various numbers of subtasks, which are organized as a DAG. Within the DAG, a DAG node represents a subtask and an edge represents the data dependency between two subtasks. If there is no data dependency between two subtasks, they can be executed on different workstations concurrently. However, in order to limit network traffic, all task execution should be within a LAN. This means that all subtasks of a task should be executed on machines within a specific LAN. When multiple tasks are submitted for execution, they compete with each other for the available computational resources. A mapping and scheduling algorithm is needed to fairly and efficiently utilize available computational resources to execute these tasks.

The simulation of a distributed heterogeneous computer network is a difficult endeavor due to the complexity of a WAN environment. In order to simplify the problem, the simulation model was designed based on following assumptions.

A machine can only execute one subtask at a time (single programming), and a subtask cannot coexist in memory with other subtasks on one computer node. Subtask execution is based on a First In First Served (FIFS) basis. Once a subtask starts running, it competes with other local tasks for resources and is scheduled by the operating system of that computer network node. Therefore, the actual execution time of a subtask can vary from the pre-estimated execution time due to the CPU loading.

Once a subtask starts, other subtasks have to wait until the running subtask is finished. Subtask execution is non-preemptive, which signifies that there are no checkpoints for subtask processes and that a subtask cannot be moved to another node after it starts. A subtask only starts after all input data are available. It is assumed that no additional data are needed during subtask execution. The output of a subtask, if any, is available only after the subtask is completed.

The DAG of a task is known when it is submitted. Task information includes the DAG, and deadline. Each task is represented by a set of communicating subtasks. These tasks are organized using a DAG, $G=(V, E)$, where the set of vertices $V=\{v_1, v_2, \ldots v_n\}$ represents the set of tasks to be executed, and the set of weighted, directed edges E represents communication between tasks. Thus $e_{ij} = (v_i, v_j) \in E$ indicates

communication from task v_i to v_j, and $|e_{ij}|$ represents the volume of data sent between these tasks. However, in this research we assume that the communication cost of a task is identical, therefore the communication cost $|e_{ij}|$ between any subtask of a task are identical. Nevertheless, the communication cost of different tasks can be different, which means that $|e_{ij}|$ of task A can be different from $|e_{ij}|$ of task B. As mentioned, the execution environment consists of a set of heterogeneous machine, which can be represented by the set of $M=\{m_1, m_2, \ldots m_q\}$. The computation cost function, $C(v_i, m_j)$, represents the execution time of a individual subtask on each available machine. The task deadline is the time line before when the user expects the task to be completed. It is assumed that all users have the same priorities. Therefore the scheduler needs to meet as many task deadlines as possible.

In the simulation model, it is assumed that network bandwidth and delay time are static, which means they do not change with time. It is also assumed that no new workstations join during the simulation, but some workstations can crash and be temporarily out of service. However, it is assumed that a computer node failure will not affect network communication, a computer node failure solely affects that machine is not able to execute subtasks, and the external scheduler and internal scheduler are never crushed.

3.2 Mapping and Scheduling Algorithm

Mapping and scheduling in this research simulation model consists of two steps: external mapping and scheduling, and internal mapping and scheduling. The external scheduling

involves WAN-wide, task-level, and distributed mapping and scheduling, whereas internal scheduling involves LAN-wide, subtask-level, and centralized mapping and scheduling. An external scheduler and internal scheduler reside in each LAN. An external scheduler carries out external mapping and scheduling, and an internal scheduler carries out internal mapping and scheduling.

External mapping and scheduling is task-level, thereby the external scheduler has no knowledge of the associated DAGs and it views each LAN as a signal computational entity. The responsibilities of external scheduler are: 1) receiving a task submitted by a computer node of LAN; 2) sending a bidding request to an internal scheduler; 3) based on bidding replies, received from internal schedulers, selecting a LAN that is best suited for the task (external scheduling); and 4) receiving results from internal scheduler.

Internal mapping and scheduling is done on the subtask level, therefore the internal scheduler has full awareness of subtasks. However, the internal scheduler has no knowledge of other LANs. The computational entities for the internal scheduler are the computer nodes within its LAN. The responsibilities of an internal scheduler are: 1) replying to bidding requests according to current local LAN task loading; 2) receiving tasks from an external scheduler; 3) conducting internal scheduling by sending subtasks to the computer nodes of the local LAN for execution; 4) receiving subtask results from computer nodes; and 5) sending task results back to the external scheduler.

3.2.1 External Mapping and Scheduling

External scheduling is based on the MCT algorithm (which was discussed in Chapter 2) to share the computing load among the LANs. The External scheduler archives the completion time for a task through "bidding". After an external scheduler receives a task, it sends its bidding request to the internal schedulers, including the internal scheduler of a local LAN, for "task auction". The bidders (internal schedulers) reply to the request with an Estimated Task Execution Time (ETET) of the task on their LAN. The algorithm for calculating ETET will be explained in the next section of internal scheduling. After the external scheduler receives all replies from the internal schedulers, it chooses a LAN based on the MCT algorithm. The external scheduler determines the best suited LAN based on the Estimated Task Response Time (ETRT). A Task Response Time (TRT) is defined as the difference between the time the task would be returned and the time the task would be sent. The ETRT is determined by considering three issues: ETET, Network Transfer Rate (NTR), and Average LAN Credibility (ALC).

LAN Credibility (LC) represents the computing reliability of that LAN. After a task is completed and sent back to the external scheduler, we get the LC of the LAN in which the completed task was executed. It is the result of the Actual Task Response Time (ATRT), which is the ATRT achieved after the task result has come back, divided by ETRT claimed by the internal scheduler, which can be represented by Equation 1.

$$LC_{i,j} = \frac{ATRT_{i,j}}{ETRT_{i,j}} \qquad \textbf{Equation 1}$$

Where: $LC_{i,j}$ = *LAN credibility of LAN i for task j*

$ATRT_{i,j}$ = *Actual task response time of task j on LAN i.*

$ETRT_{i,j}$ = *Estimated task Response execution time of task j on LAN i.*

If the LC is higher than 1, it demonstrates that the internal scheduler overestimates the computing ability of the LAN it represents. Conversely, if the LC is less than 1, it means that the internal scheduler underestimates the computing ability of the LAN. The Average LC (ALC) is a weighted average that is shown in Equation 2. The initial ALC is set to be 1.

$$ALC_j = old\ ALC_j \cdot 0.99 + LC_{i,j} \cdot 0.01 \qquad \textbf{Equation 2}$$

Where: ALC_j = *Average LAN credibility of LAN j*

$old\ ALC_j$ = *Previous Average LAN credibility of LAN j*

$LC_{i,j}$ = *LAN Credibility of LAN j of task i*

The external scheduler decides which LAN the task is sent to by the ETRT. This is defined using Equation 3.

$$ETRT_{i,j} = (ETET_{i,j} + \frac{Task\ Data\ Size_i}{NTR_{j,k}} + \frac{Task\ Outcome\ Size_i}{NTR_{j,k}}) \cdot ALC_{j,k} \qquad \textbf{Equation 3}$$

Where: $ETRT_{i,j}$ = *Expected task response time of job i in LAN j*

$ETET_{i,j}$ = *Expected task execution time of job i in LAN j*

$ALC_{j,k}$ = *Average LAN credibility of LAN j in the credibility table of LAN k*

Task Data Size $_i$ = *Size of task data of task i*

Task Outcome Size$_i$ =*Size of outcome of task i*

Network Trans Rate $_{j,k}$ = *Network transfer rate between LAN j and LAN k*

Since every LAN has only one internal scheduler and external scheduler, the internal scheduler and external scheduler are assumed to be located on the gateway of the LAN. Therefore, if a task is assigned to a local LAN, no network data needs to be transferred. In this case the second part of equation 3 is 0. After an external scheduler receives all replies, it selects the LAN that offers the minimum ETRT and sends the job to it for execution. The pseudo-code for external scheduling is represented in Figure 2.

```
external scheduling {
        if a task submitted {;
                for every internal scheduler participating in bidding {
                        send bidding request;
                }
        }
        if a bidding reply comes in {
                Store the bidding replay;
                if all bidding replies for this task are received {
                        for all bidding replies{
                                select the minimum  ETRT;
                                select LAN = LAN offers minimum ETRT;
                        }
                send task to selected LAN;
        }
        if a task completion message comes in {
                get the  task result;
                update ALC;
        }
}
```

Figure 2. Pseudo-Code for External Scheduling

3.2.2 Internal Mapping and Scheduling

As mentioned, subtasks of a task can run concurrently on different computer nodes within a LAN if there is no data dependence among them. After a task arrives at a LAN for execution, the subtasks are distributed to workstations of the local LAN according to the internal mapping and scheduling algorithm. The internal scheduler is responsible for carrying out internal mapping and scheduling.

Once a task arrives, all subtasks of that task are placed into an Arrive Subtask Set (ASS), which also holds subtasks from other tasks. Those subtasks of which all the necessary input data are available are further moved into a Ready Subtask Queue (RSQ). Thus there

is no data dependence among the subtasks in an RSQ. Every time before an internal scheduler conducts an internal scheduling, it will check its ASS and move any subtasks that are ready to an RSQ. An internal scheduler only schedules those subtasks within the RSQ.

An internal scheduler uses the static b-level to assign priorities to subtasks. Subtasks are assigned a priority before they are mapped and scheduled. A Subtask Priority (SP) is determined using following equation.

$$SP_i = Deadline_i - blevel \qquad \textbf{Equation 4}$$

Where: SP_i = *Priority of subtask i*

$Deadline_i$ = *Deadline of* job_i

$blevel_i$ = *b-level of* $subtask_i$

Once subtasks are assigned priorities, they are queued up in an RSQ with lower SP values in the front and higher SP values in the back. Subtasks with lower SP values have higher priority in mapping to a machine for execution. An internal scheduler picks up the subtask on the top of the RSQ and assigns it to the machine that can offer minimum Expected Subtask Response Time (ESRT). ESRT is defined as the difference between the time the subtask would be returned and the time the subtask would be sent, which is shown in Equation 5.

$$ESRT_{i,j} = EMFT_j + \frac{SET_i}{Computing\ Power_j} + \frac{Subtask\ Size_i}{NRT_j} + \frac{Result\ Size_i}{NRT_j}$$

Equation 5

Where: $ESRT_i$ *=Expected subtask response time of subtask i on computer node j.*

$EMFT_j$*= Expected machine free time of computer node j for new task*

SET_i *=Subtask execution time of subtask i*

$Computing\ Power_j$ *= Computing Power of computer node j*

$Subtask\ Size_i$ *= Subtask data size of subtask i*

$Result\ Size_i$*= Result size of subtask i*

NTR_i *= Network transfer rate to node j*

Machine Free Time (MFT) is defined as the time that a machine is free to execute new tasks. EMFT is an Estimated MFT predicted by an internal scheduler, and AMFT is the Actual MFT. The second part of Equation 5 is defined as Estimated Subtask Execution Time (ESET), which is defined as the result of SET divided by Computing Power. However, the Actual Subtask Execution Time (ASET), which represents the actual running time of the subtask on that workstation, could be different because of the dynamic nature of CPU loading. SET is calculated on a workstation of which the computing power is 1; ESET is calculated by assuming the machine is 100% free, therefore the ASET of a subtask on a computer node is equal to or less than ESET; and ASET is the real running time of that subtask on a machine. A system crash, however,

can cause the ASET to be infinite, which in turn causes the Actual Subtask Response Time (ASRT) to be infinite.

Internal schedulers make decisions based on the ESRT. If we knew the ASET of subtasks, then we could further know the ASRT. From this, we could make a better mapping and scheduling algorithm, but in practice, this is impossible. Therefore, we set up a threshold for the ASRT. When a subtask is executed for too long and passes the late threshold, we consider that computer to have crashed. In this situation, all the subtasks inside the SWQ, including the subtask currently running, are pulled back for rescheduling. The internal scheduler will claim that the computer in question has crashed and erase its computer ID from the participating workstation list.

Threshold determination is difficult. Under some circumstances, internal schedulers will make an incorrect mapping, but to improve on this would require more computational cost. On the other hand, sometimes an internal scheduler has to correct the wrong mapping because if it continued, it would make things worse and cause more damage. Most of the time, the ASRT is different from ESRT. But if the difference is slight, it is better to keep the old scheduling because to rearrange the SWQ is a big job and it would not be worth it. But if the difference is beyond some range, it is better for the internal scheduler to redo the scheduling because as a tradeoff of the effort spent on rescheduling, the efficiency of the computing resource usage increases.

After an internal scheduler sends a subtask, it erases the subtask entry from the RSQ, moves the elements in the RSQ one space forward, and then updates the EMFT of that machine. The EMFT is updated by using Equation 6.

$$EMFT_j = old\ EMFT_j + \frac{SET_i}{Computing\ Power_j} \qquad \textbf{Equation 6}$$

Where: $EMFT_j$= *Expected machine free time of computer node j*

$old\ EMFT_j$=*old Expected machine free time of computer node j*

SET_i =*Subtask execution time of subtask i, which was just sent*

Computing Power $_j$ = *Computing Power of computer node j*

Then internal scheduler repeats the same procedure until either the RSQ is empty or the Subtask Waiting Queue (SWQ) of that assigned computer is full. Figure 2 illustrates the subtask movement between subtask sets during internal scheduling.

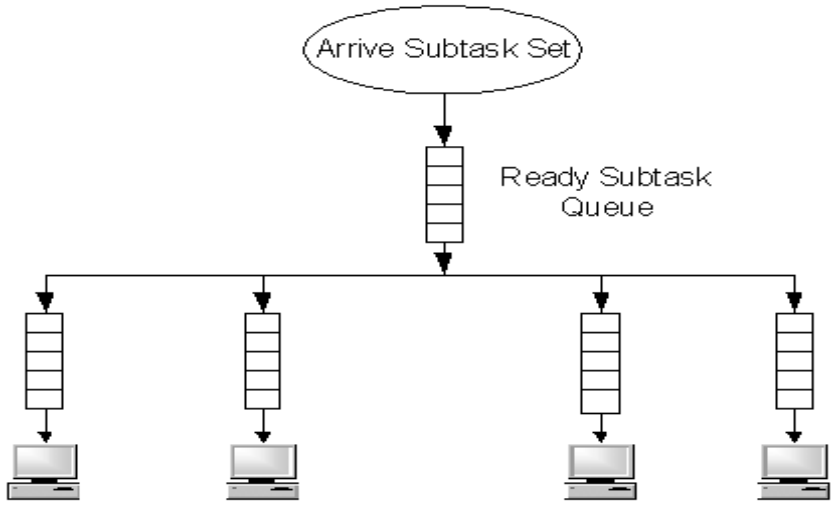


Figure 3. Internal Scheduler Subtask Movement

An SWQ is the subtask waiting queue held by each workstation to insure subtasks are executed by following FIFS. Subtasks sent to workstations are queued up in an SWQ where they wait for execution. If a SWQ of a machine is full, the internal scheduler cannot send a subtask to that workstation. The length of an SWQ is another major issue in internal scheduling. Internal schedulers depend on EMFT to conduct internal scheduling. Nevertheless, the actual network computing performance depends on Actual Machine Free Time. The longer the SWQ, the earlier the subtasks within the SWQ are sent out. On one hand, we want to keep the SWQ as short as possible because: 1) the EMFT is different from the AMFT because of the variation of CPU loading, therefore the later the subtasks are sent, the more actual run time information the internal scheduler can use to do more precise scheduling; 2) in case of machine failure, an internal scheduler needs to pull back subtasks in the SWQ and resend the subtask data, which will increase network traffic and delay subtask execution. On the other hand, if the subtask-waiting

queue is too short, some machines will find the SWQ is empty after they finish their current subtask and sit idle waiting for the internal scheduler to send another one, in which case wasting precious CPU time. Based on the information above, the choice of size of an SWQ is a balance between these effects.

Internal scheduling is an event-driven procedure. The firing events, which are named "mapping events," include the arrival of new tasks, the completion of subtasks, and system traps. A system trap occurs when a subtask is running on a machine for too long and passes the threshold (ASRT > threshold). Upon receiving a subtask completion message, an internal scheduler checks if it is the last subtask of that task. If it is, the internal scheduler will send the task result back to the external scheduler. The pseudo-code of the internal scheduling is listed in Figure 4.

```
internal scheduling {
        if receive a bidding request {
                check current subtasks in ASS and RSQ;
                check EMFT of each machine in local LAN;
                get simulated ETET;
                send ETET back to external scheduler;
        }
        if receive a task execution request {
                put subtasks of the task into ASS;
                move ready subtasks to RSQ;
                while (RSQ ≠ empty ) {
                        for computer ∈ local LAN {
                                pick a machine offering minimum ESRT;
                                best machine = machine offering minimum ESRT;
                        }
                        if SWQ of best machine has position {
                                send the subtask on top of RSQ to the best machine;
                        }else
                                break;
                        rearrange RSQ;
                }
        }
```

```
        if receive a subtask completion message {
                if that subtask is the last subtask of a task
                        send the task result back to external scheduler;
                move ready subtasks to RSQ;
                while (RSQ ≠ empty ) {
                        for every computer node in local LAN {
                                pick a machine offering minimum ESRT;
                                best machine = machine offering minimum ESRT;
                        }
                        if SQW of best machine has position {
                                send the subtask on top of RSQ to the best machine;
                        }else
                                break;
                        rearrange SWQ;
                }
        }
        if (ASRT > threshold) {
                remove all subtasks inside SWQ of that node back to RSQ;
                rearrange RSQ;
                while (RSQ ≠ empty ) {
                        for every computer node in local LAN {
                                pick a machine offering minimum ESRT;
                                best machine = machine offering minimum ESRT;;
                        }
                        if SQW ≠ full {
                                send the subtask on top of RSQ to the best machine;
                        }else
                                break;
                        rearrange RSQ;
                }
        }
}
```

Figure 4. Pseudo-Code for Internal Scheduling

4 Simulation Program

In this chapter, the details of the simulation program implementation are given. The specification of the simulation system is also listed.

4.1 Simulation Language and Design

The simulation software was written in the PARSEC parallel simulation language [MeB98]. As explained in the PARSEC User Manual: "PARSEC (for PARallel Simulation Environment for Complex systems) is a C-based discrete-event simulation language. It adopts the process interaction approach to discrete-event simulation. An object (also referred to as a physical process) or set of objects in the physical system is represented by a logical process. Interactions among physical processes (events) are modeled by timestamped message exchanges among the corresponding logical processes."

The overview of the simulation program layout is illustrated in Figure 5. A simple approach to designing a network simulation model is to create each computer network node as an entity. Although it is easy to understand and implement, it has a scalability problem [BaT99]. By increasing the number of computer network nodes, the number of entities that the simulation program generates will increase. This amplifies both memory requirements and the number of messages passed among entities, which significantly deteriorates the performance of the simulation.

A different approach was adopted for this thesis. Instead, each computer network node is represented by an entity. A genetic entity was created to represent all computer network nodes. Messages are passed with a tag showing the destination of the message. After the machine entity receives the message, first it examines the tag of the message. According to the tag, it changes its local variable to represent the computer network nodes the message is supposed to arrive at. The same approach was used for the external schedulers and internal schedulers. This approach decreased the overhead of message passing and dramatically increased the performance of the simulation. In the simulation program, only four entities were created: the job maker entity, the external scheduler entity, the internal scheduler entity, and the machine entity. Tiers Random Network Generator, Tiers–1.1, generated the computer network topology [Do96]. Tiers-1.1 is a random network topology generator used to generate the simulation test bed. With the Tiers package, the user can configure the complexity of the network (i.e. specify number of nodes and connectivity) and it will generate a random network with the specified properties. The link propagation delays are derived from the length of the links (i.e. the distance between the two nodes being connected). Also, a bandwidth is suggested. The user can interactively change those values after topology generation.

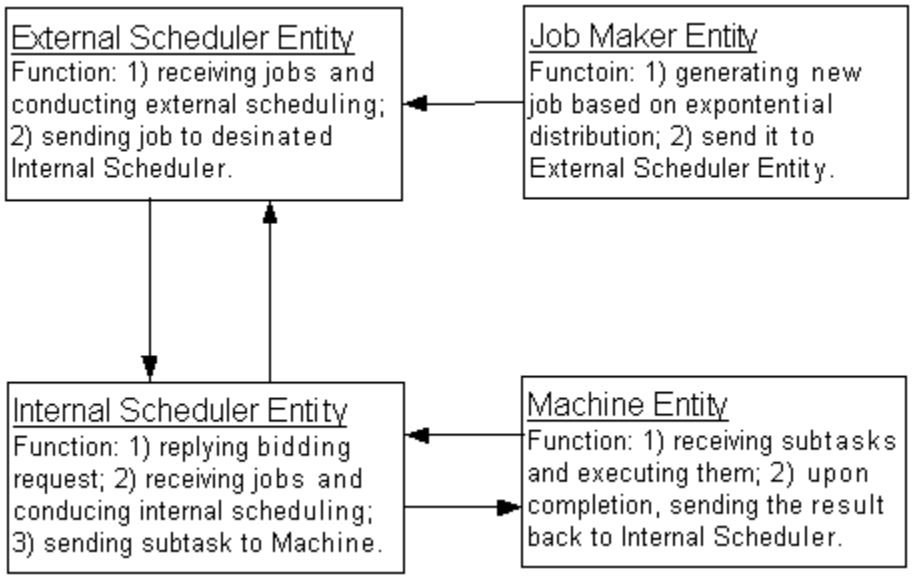


Figure 5. Simulation Program Layout

4.2 System Specifications

To write simulation software, it is necessary to set system variables to define the simulation environment. The overview of the system environmental specifications used in this research is listed on Table 1. Simulation Environmental Specifications are grouped into four categories: network specification, machine specification, task specification, and run time specification. The details of these specifications are explained in following subsections.

Table 2. Simulation Environmental Specification

Simulation Environmental Parameters	
Network Specification	LAN number
	Node number of each LAN
	Total number WAN level node
	WAN bandwidth
	LAN bandwidth
	Node propaganda delay
Workstation Specification	Machine recover time
	Machine failure rate
	Machine computing power
Task Specification	Task size
	Task result size
	Subtask execution time
	Max number of subtask in each task
	DAG degree
Run Time Specification	External scheduling execution time
	Internal scheduling execution time
	Total task number
	SWQ size
	Task interval
	Total number of bidding LAN

4.2.1 Workstation Specification

The simulation program was defined to simulate a WAN consisting of 100 LANs. Each LAN consisted of 10 nodes. In addition, 10 nodes belonged to the WAN itself. Therefore, there are a total of 1100 nodes in the simulation. The bandwidth between WAN nodes was defined to be 10 Kbytes/sec, and the bandwidth between LAN nodes was set to be 100 Kbytes/sec. The node delay, which presents the message passing processing time, was ranged from 51 ms to 5060 ms. Messages passing through a node were delayed by the node delay time of that computer node. A network topology generated by Tiers1.1 is

displayed on Figure 7. Based on the above information, the data transfer time can be determined by equation 7.

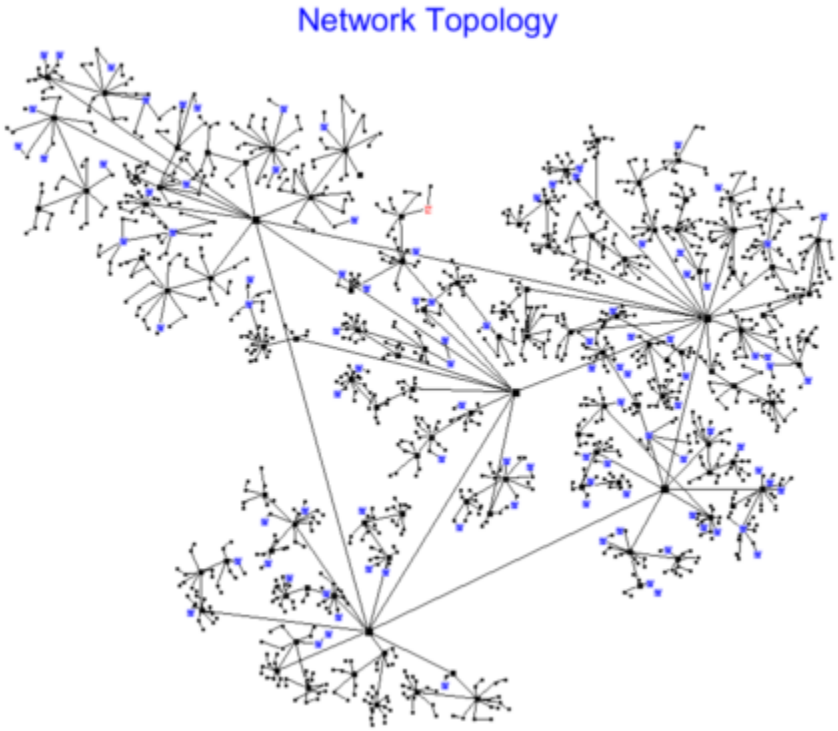


Figure 6. Nework topology generated by Tiers1.1[YaJ].

$$T_{l,z} = \begin{cases} \dfrac{DS}{Bandwidth_{l,z}} & if\,(z = l+1) \\ \displaystyle\sum_{i=l}^{z} \frac{DS}{Bandwidth_{i,i+1}} + \sum_{i=l+1}^{z-1} Delay_i & if\,(z = l+1) \end{cases} \qquad \textbf{Equation 7}$$

Where:

DS = Data size transferred, Kbytes;

$Bandwidth_{l,z}$ = Network bandwidth between node l and z, Kbytes/sec.

$Delay_i$=Network traffic delay of node i; sec.

l+1…z-1 = network node numbers along the path of l to z;

4.2.2 Workstation Specifications

The simulation is based on a heterogeneous workstation network environment. As discussed in the literature, heterogeneity takes the form of the heterogeneity of a configuration, architectural heterogeneity, and operating system heterogeneity. In this simulation, however, only the heterogeneity of a configuration is considered. The computing power that is pre-assigned to each node reflects the heterogeneity of a configuration. The computing power ranged from 0.3 to 1. We assume that the subtask execution time is inversely proportional with the computing power of the machine it is executed on, and consists of the computational cost divided by the computing power. For example, if a computer node *i* has a computing power of *x* and another computer node *j* has a computing power of *y*, therefore the computer node *i* is *x/y* times as fast as node *j*. Subtask execution time on node *i* is equal to *y/x* of the execution time on node j.

A dynamic environment brings dynamic CPU loading. As mentioned before, workstations can crash during subtask execution. However, if a workstation crashed, there would be no means to notify the internal scheduler. Therefore, a threshold was set up as 2 × ESRT of the task-workstation pair. After a subtask is sent to a machine for execution, a threshold was set up to trace that subtask and workstation. If the subtask ASRT was greater than 2 × ESRT, the internal scheduler would assume that the machine has failed.

The computer failure rate was defined as the possibility of the ESRT being higher than 2 × ASRT during a subtask's execution. The machine failure rate ranged from 0% ~ 0.2%. For example, if the failure rate was 0.1%, it indicated the possibility that the ESRT would be higher than 2 × ASRT once every 1000 subtask executions. However, the simulation ignored other causes of workstation failure such as power off, or system maintenance shutdown. If a workstation never executed a subtask, it would never crash. After a computer node crashed, it would take from 1 sec to 1 hr for the computer node to actually come back to service. It was assumed that a workstation crash could only affect its ability to execute subtasks, but it did not affect other abilities like scheduling (if the scheduler is located on that computer node) or transferring data.

4.2.3 Task Specification

Network computing requires task data and result transferring around the WAN and LAN. It was assumed that the task size ranged from 10 Kbytes and 2 Mbytes, and the size of a task's result ranged from 10 Kbytes to 5 Mbytes. Each task had at most 50 and at least 10 subtasks. All subtasks in a task were organized by a tree structure. The maximum degree of the tree was 5, which required that each subtask would have at most 5 child subtasks.

Subtasks had different execution times. The subtask execution time was determined as the running time of that subtask on a computer node with a computing power of 1. In the simulation program, the subtask execution time ranged from 1 min to 10 min. The EST of

that subtask was determined by the computing power of the computer node that the subtask was executed on using following equation.

$$EST_{i,j} = \frac{SET_i}{Computing\ Power_j} \qquad \textbf{Equation 8}$$

Where:

$EST_{i,j}$= *Estimated subtask execution time of task i on machine j;*

SET_i = *Subtask execution time of task i on a machine with computing power of 1.*

$Computing\ Power_j$ = *Computing ability of machine j.*

4.2.4 Run Time Specifications

The time of external and internal scheduling were fixed to be 12 ms and 28 ms, respectively. Task arrival for interval scheduling followed an exponential distribution. Once a task was submitted, the external scheduler asked the internal schedulers to bid for the task. The external scheduler could broadcast its task-bidding request to every internal scheduler within the WAN, which had the benefit of evenly distributing task loading across the WAN. But as a tradeoff, broadcast bidding requests increased network traffic. In this simulation, the number of internal schedulers participating in task bidding was set to be 10, including the local internal scheduler. The participating internal schedulers were selected randomly.

As the number of submitted tasks increased, the computing load on the entire network increased also. In the simulation, different numbers of tasks were tested in order to the observe impact on the performance of network computing.

5 Experiment Result and Discussion

Based on the discussion above, the simulation was carried out in four phases: varying the task numbers; varying the task interval; changing of machine failure rate; and varying the network infrastructure. In order to analyze the impact of the SWQ size, each variation was carried out with different waiting queue sizes. Every simulation was repeated 100 times with different random seeds to get an average. The simulation was conducted on a computer with 8 Pentium III 550 processors, with 4 GB memory, and running on a Linux Red Had 6.1 platform.

In the simulation, we used two major key indexes to measure network-computing performance: the percentage of tasks meeting their deadlines and the average response time. The percentage of tasks meeting their deadlines was the percentage of tasks submitted to the simulation that were returned to the external scheduler before their deadline. The average response time was the average time difference between the time a task is returned to the external scheduler and the time the task is sent out by the external scheduler.

Network traffic was also monitored. Network loading was divided into two components: the network traffic caused by external scheduling and that caused by internal scheduling. Networking loading caused by task bidding, tasks sent to a LAN for execution, and results sent back to the external scheduler belonged to external network traffic. Network loading caused by subtasks distributed to computer network nodes within a LAN

belonged to internal network traffic. Both internal and external network traffic were recorded during the simulation. Network traffic was measured by the sum of the size of the packet transferred multiplied and the number nodes the packet had to travel through. For example, if the size of a packet was 2 MB, and it traveled from node A to node B, to node C, and then to its destination of node D, and the total traveling time is 10 sec, the network traffic caused by that packet is deemed to be 0.8 MB/sec during that 10 sec period. This is to record to the network loading on entire network.

5.1 Variation of Task Number

The first experimental phase was to examine the influence of the number of tasks on network computing performance. The simulation environmental parameter setting is represented in Table 3. The bold and italic entries are the specifications that were variables in this simulation phase. As demonstrated in Table 3, the simulation was conducted on various numbers of tasks and sizes of SWQs. All other parameter were kept constant throughout this simulation phase.

Table 3. Simulation Phase I Environmental Parameter Setting

Simulation Environmental Parameter		**Value**
Network Specifications	Number of LANs	100
	Number of nodes in each LAN	10
	Total number WAN level node	10
	WAN bandwidth	10 Kbytes/sec.
	LAN bandwidth	100 Kbytes/sec.
	Node propaganda delay	[51 ms, 5060 ms]
Machine Specifications	Machine recover time	[1 sec, 1 hr]
	Machine failure rate	0.2 %
	Machine computing power	[30%, 100%]
Task Specifications	Task size	[10 Kbytes, 2,000 Kbytes]
	Task result size	[10 Kbytes, 5,000 Kbytes]
	Subtask execution time	[1 min., 10 min.]
	Max number of subtask in each task	[10, 50]
	DAG degree	[1, 5]
Run Time Specifications	External scheduling execution time	12 ms
	Internal scheduling execution time	28 ms
	Total number of tasks	***{1000, 1500, 2000, 2500, 3000}***
	SWQ size	***{1, 2, 3, 4, 5, 10, 15, 20}***
	Task interval	4000
	Total number of bidding LAN	10

The impact of the number of tasks on the percentage of tasks meeting their deadlines and the average response time were illustrated in chart 1 and 2 of Figure 7, respectively. It can be observed from chart 1 and 2 that as the number of tasks increased, the network performance worsened. This is easily understood. As more tasks were submitted, the less computational resources were available for the tasks to share. This was reflected by a decrease in the percentage of tasks that met their deadlines and by an increase in average response time.

The impact of changing the SWQ size was also apparent. It is observed from Figure 7 that network computing performance increased dramatically as SWQ size increased from

1 to 2. Nevertheless, as the SWQ size continued to increase from there, only small changes in computing performance could be observed. The network computing performance reached its peak as the SWQ size was increased. Depending on the number of submitted tasks, where the SWQ size reached its peak performance differed. As hypothesized before, choosing the best SWQ size was a balance between the negative effects of having a size that was too long or too short. The results confirmed this theory. As the SWQ size was continually increased, the network performance began to decrease.

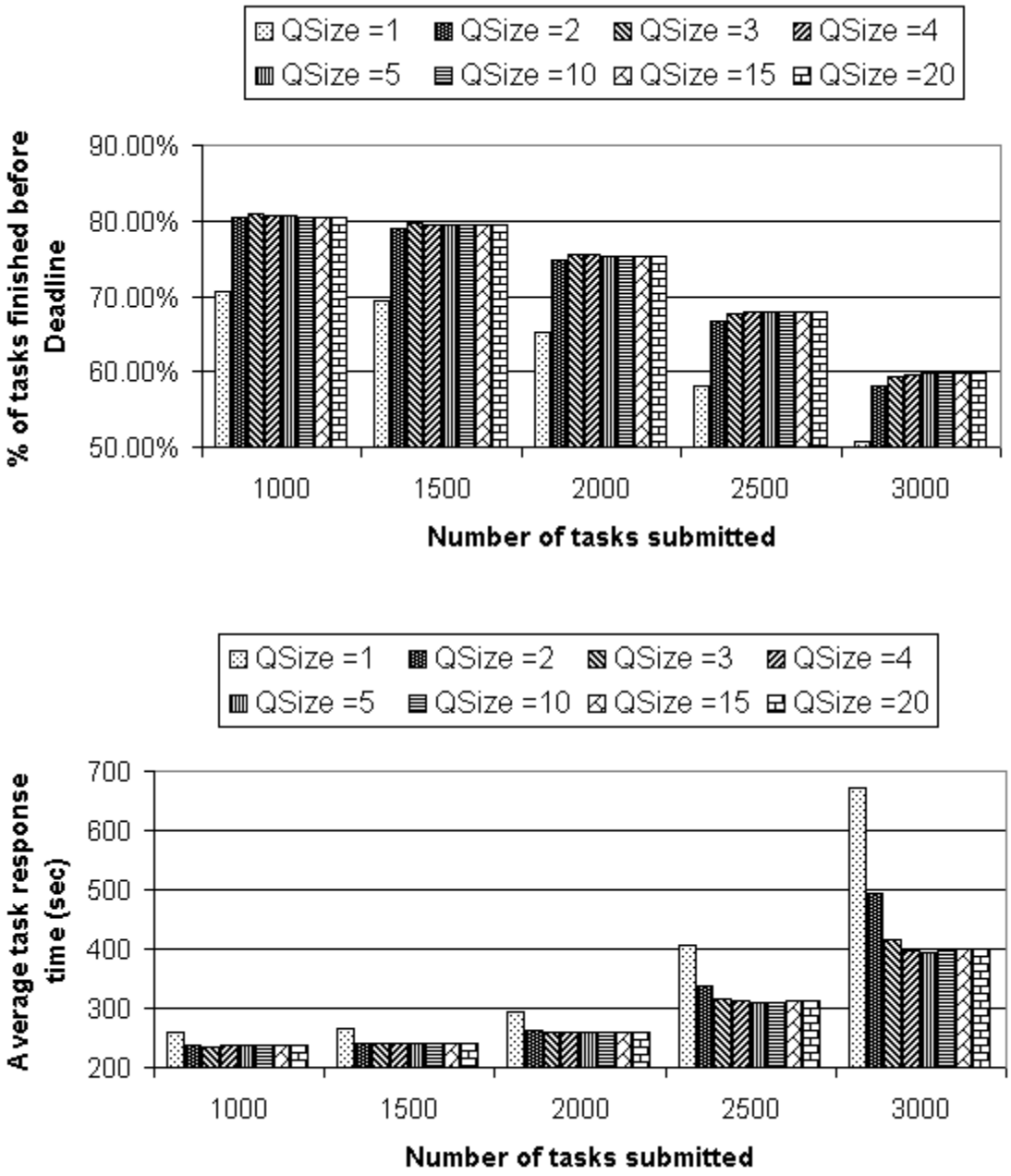


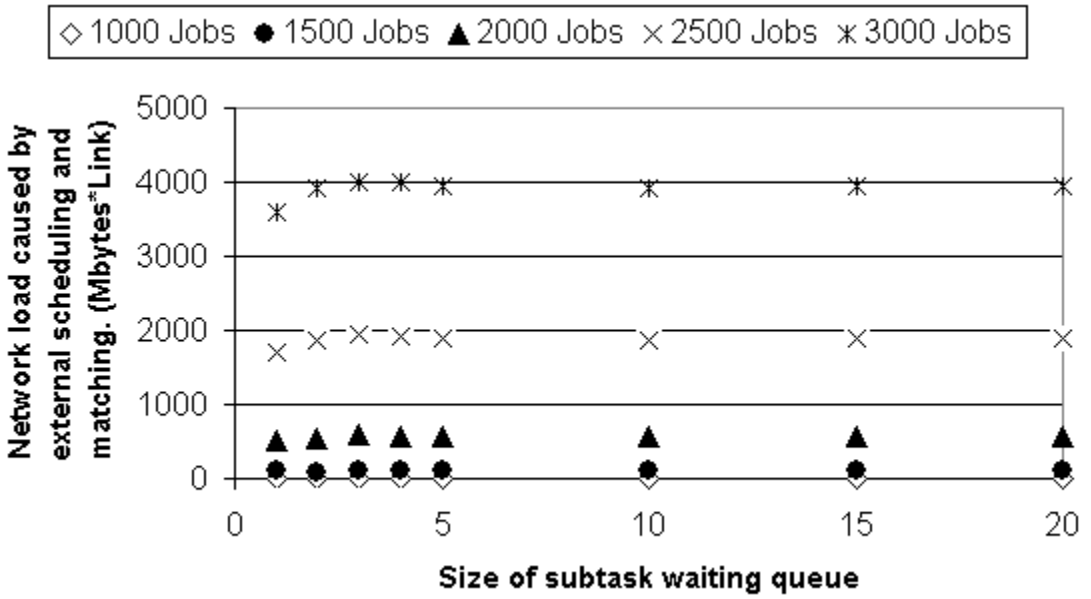
Figure 7. The impact of submitted task number on network computing performance during simulation Phase I.

Chart 1 and chart 2 of Figure 8 illustrated the impact of the number of tasks and SWQ size on external and internal network traffic. Network traffic was caused by sending bidding requests, sending tasks to other LANs for execution, and sending subtasks within a LAN. External and internal network traffic increased as the number of tasks increased. As more tasks were submitted, more network traffic was generated because of internal and external scheduling.

It was interesting to find that as the size of the SWQ increased from 1 to 3, the external network traffic increased. As the SWQ size continued to increase, the external network traffic dropped. An increase in external network traffic indicates that more tasks were sent to the LANs than executed locally. The change of external network traffic matched the change of network computing performance, and as more tasks were distributed in the WAN, the computing resources could be better shared, which produced a better computing result. However, the change of the external network traffic caused by the change of SWQ size could only be observed in a high task number experiment. As the number of tasks decreased from 3000 to 1000, the change became negligible. The explanation for this is that as the number of tasks decreased, tasks were most likely executed on the local LAN, and there was no need to transfer them to other LANs. Therefore, external network traffic did not change very much.

As the SWQ size increased, the internal network traffic also increased. As discussed before, an increase in SWQ size resulted in more subtasks being pulled back in the case of a machine crash. The results matched the assumptions we made. The internal network

traffic increase caused by the changes of SWQ size was higher in high task number experiment runs. As fewer tasks were submitted, the SWQs, which stored the subtasks assigned to be run on a given machine, stored only a few subtasks any way. The change of SWQ size had little impact.



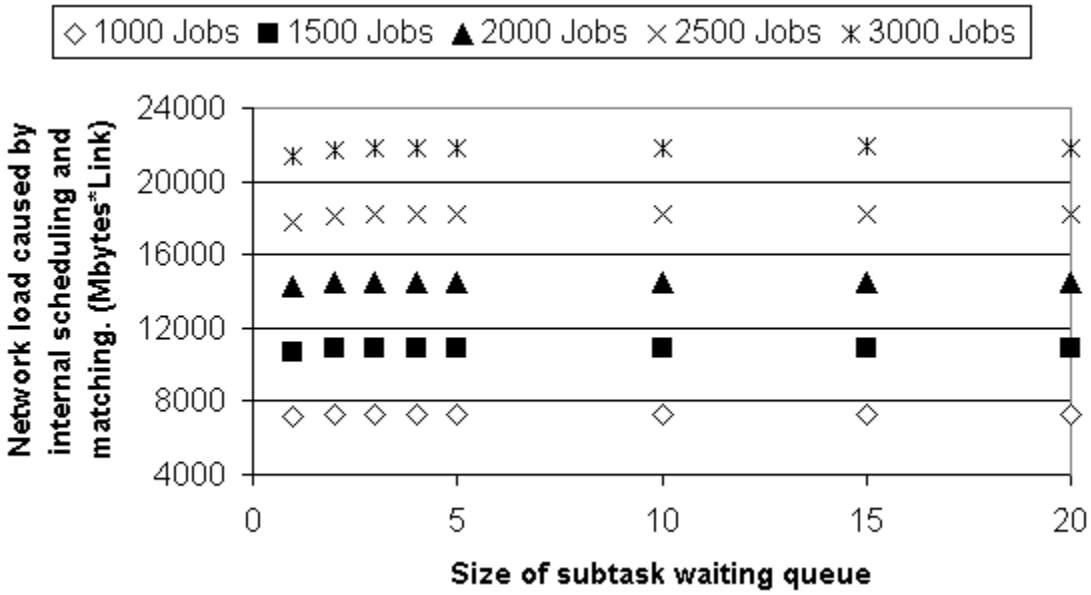


Figure 8. The impact of submitted task number on network traffic loading during simulation Phase I.

5.2 Variation of Task Interval

Of interest in the second simulation experimental phase was to observe how task intervals affected the network computing performance. The simulation's environmental specifications were listed in Table 3. The bold and italic entries were the parameters that were changed in this simulation phase. Simulation phase II was conducted on various task intervals and different sizes of SWQs. All other parameters were kept constant through this simulation phase.

Table 4. Simulation Phase II Environmental Parameter Setting

Simulation Environmental Parameter		**Value**
Network Spc.	Number of LANs	100
	Number of nodes in each LAN	10
	Total number WAN level node	10
	WAN bandwidth	10 Kbytes/sec.
	LAN bandwidth	100 Kbytes/sec.
	Node propaganda delay	[51 ms, 5060 ms]
Workstation Spc.	Machine recover time	[1 sec, 1 hr]
	Machine failure rate	0.2 %
	Machine computing power	[30%, 100%]
Task Spc.	Task size	[10 Kbytes, 2,000 Kbytes]
	Task result size	[10 Kbytes, 5,000 Kbytes]
	Subtask execution time	[1 min., 10 min.]
	Max number of subtask in each task	[10, 50]
	DAG degree	[1, 5]
Run Time Spc.	External scheduling execution time	12 ms
	Internal scheduling execution time	28 ms
	Total number of tasks	2000
	SWQ size	***{1, 2, 3, 4, 5, 10, 15, 20}***
	Task interval	***{5000 sec., 4000 sec, 3000 sec., 2000 sec., 1000 sec.}***
	Total number of bidding LAN	10

The impact of task intervals on the percentage of tasks meeting their deadlines and the average response time was illustrated in chart 1 and chart 2 of Figure 9, respectively. The simulation with 1000 ms task intervals had very poor network computing performance compared to other simulations with higher task intervals. As task intervals continued to increase, the computing performance dropped. The possible reasons for these phenomena are local bidding errors and remote bidding errors.

The local networks were superior to the remote networks in bidding tasks because the local networks did not need task and result transferring, therefore the time needed to transfer data was 0. Thus, tasks were tended to execute on the local network if the computing load was low. The bidding replies of the internal scheduler were based on the current network computing load. There was a time interval that occurred between when a task would begin to be bid upon and when the task finally arrived at its destination network. In the case when a submitted task interval was very intense, many tasks could be sent to a local network during the time interval of task bidding. For example, consider the situation of task 1 being submitted to a network. The external scheduler would send a bidding request out, and wait for the bidding replies. In the meantime, task 2 would also be submitted to the same network, so the external scheduler sends the bidding request of task 2 out. Because the local internal scheduler would use the current network computing load to make a bid, and if task 1 had not arrived yet, it would send a “good” bid for task 2 as well as for task 1. This could result in both task 1 and 2 being sent to the local LAN and cause it to be overloaded.

As the task submission interval increased, more tasks were sent to remote LANs, by which the local bidding error problem was solved. But another remote bidding error problem arose. Some remote LANs were superior to others because of better bandwidth and faster computer systems. As mentioned, there was a time interval between when task bidding started and when the task finally arrived at the destination LAN for execution. In the case of a task being sent to a remote LAN, the time interval was greater than for a local submission because the task's data needed to be sent to the remote LAN. Even when a task was assigned to execute on a LAN and the task data was on its way, the internal scheduler would not consider the incoming task when it conducted the ETRT calculation. Since the bidding algorithm was based on the current network computing load, a task sent to the current "best LAN" may become overloaded very soon. Thus many tasks can be sent to the same remote LAN, in which case it caused the remote LAN to become overloaded which in turn deteriorated the overall network computing performance. As more tasks were sent remotely, the situation became worse.

SWQ size continued to have an influence on network computing performance regardless of changes in the task submission interval. The results showed that a size of 3 or 4 for the SWQ was the best overall; it gave the highest percentage of completed tasks before their deadline and the lowest average response time.

The impact of the task submission interval on external and internal network traffic was illustrated in charts 1 and 2 of Figure 10, respectively. Chart 1 confirmed the statement above, that as the task submission interval increased, external network traffic also increased, which indicated that more tasks were run remotely. However, from chart 2 of

Figure 10, it seemed that the task submission interval had no obvious impact on internal network traffic.

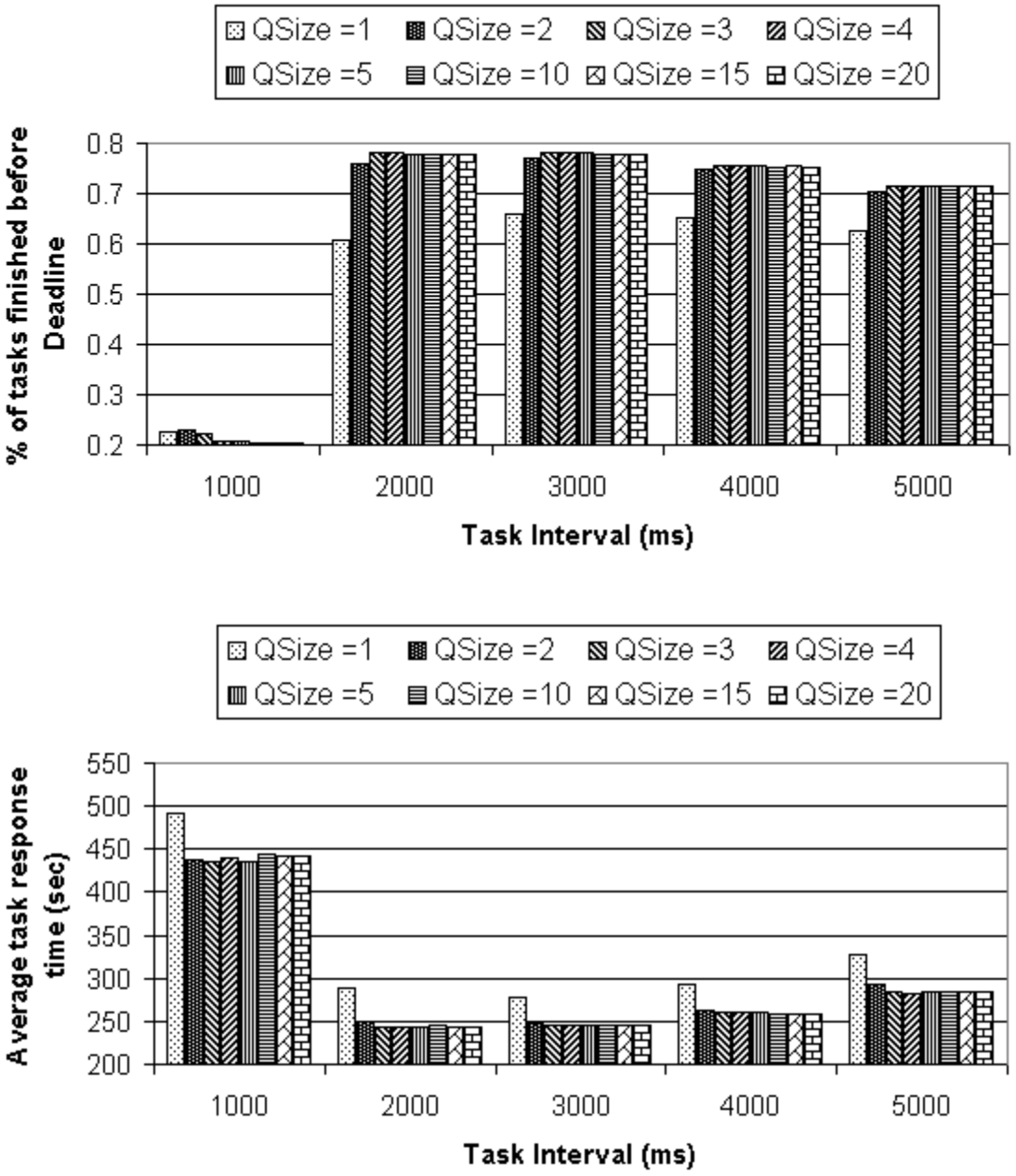


Figure 9. The impact of submitted task interval on network computing performance during simulation Phase II.

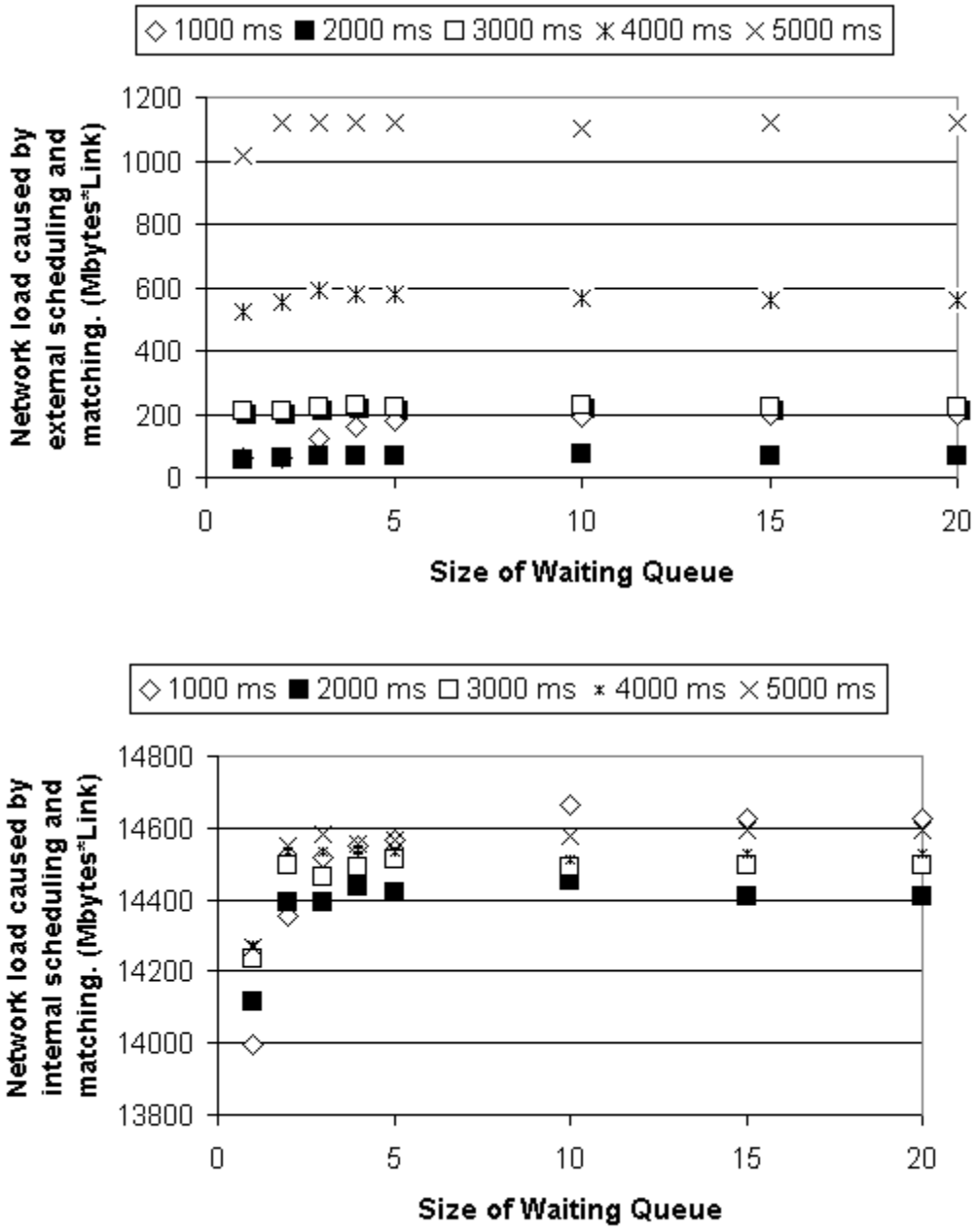


Figure 10. The impact of submitted task interval on network traffic loading during simulation Phase II.

5.3 Variation of Machine Failure Rate

Of interest in the third simulation phase was the observation of how the machine failure rate impacted computing performance. The simulation's environmental parameter settings were represented in Table 4. The bold and italic entries were the variables in this simulation phase. Simulation phase three was conducted on various machine failure rates and SWQ sizes. All other parameters were kept constant through this simulation phase.

Table 5. Simulation Phase III Environmental Parameter Setting

Simulation Environmental Parameter		**Value**
Network Specification	Number of LANs	100
	Number of Nodes in each LAN	10
	Total number WAN level node	10
	WAN bandwidth	10 Kbytes/sec.
	LAN bandwidth	100 Kbytes/sec.
	Node propaganda delay	[51 ms, 5060 ms]
Workstation Specification	Machine recover time	[1 sec, 1 hr]
	Machine failure rate	***{0.2 %, 0.15%, 0.1%, 0.05, 0}***
	Machine computing power	[30%, 100%]
Task Specification	Task size	[10 Kbytes, 2,000 Kbytes]
	Task result size	[10 Kbytes, 5,000 Kbytes]
	Subtask execution time	[1 min., 10 min.]
	Max number of subtask in each task	[10, 50]
	DAG degree	[1, 5]
Run Time Specification	External scheduling execution time	12 ms
	Internal scheduling execution time	28 ms
	Total number of tasks	2000
	SWQ size	***{1, 2, 3, 4, 5, 10, 15, 20}***
	Task interval	4000 sec
	Total number of bidding LAN	10

The impact of the machine failure rate on the percentage of tasks completed before their deadlines and the average response time was illustrated in chart 1 and chart 2 of Figure

11, respectively. From Figure 11, it can be observed that as the machine failure rate increased, the network computing performance degraded. A sudden decrease in network computing performance occurred after the machine failure rate increased from 0 to 0.5%. As the machine failure continued to increase, the decrease in computing performance became slight. Both charts 1 and 2 confirmed our assumptions that the decrease in the machine failure rate helped computing performance. All of the subtasks in the SWQ were pulled back for rescheduling and remapping when a node failure occurred, and consequently this delayed subtask execution.

Figure 11 also illustrated that while SWQ size increased from 1 to 3 or 4, the computing performance increased regardless of the impact of the change in the machine failure rate. As discussed before, if the SWQ size is too short, some machines will sit idle, waiting for internal scheduler to send them subtasks. This would result in a degradation of computing performance. However, as the SWQ size continued to increase, there was a negative influence on performance. If the SWQ size was too long, it affected computing performance in two ways: 1) The internal scheduler used EST to conduct scheduling--sending subtasks earlier resulted in using less actual run time information, which consequently resulted in less optimal mapping and scheduling; 2) In case of machine failures, the internal scheduler needed to pull back subtasks in the SWQ and resend the subtask data, which increased network traffic and delayed subtask execution. However, as the machine failure rate dropped to zero, the second effect was eliminated.

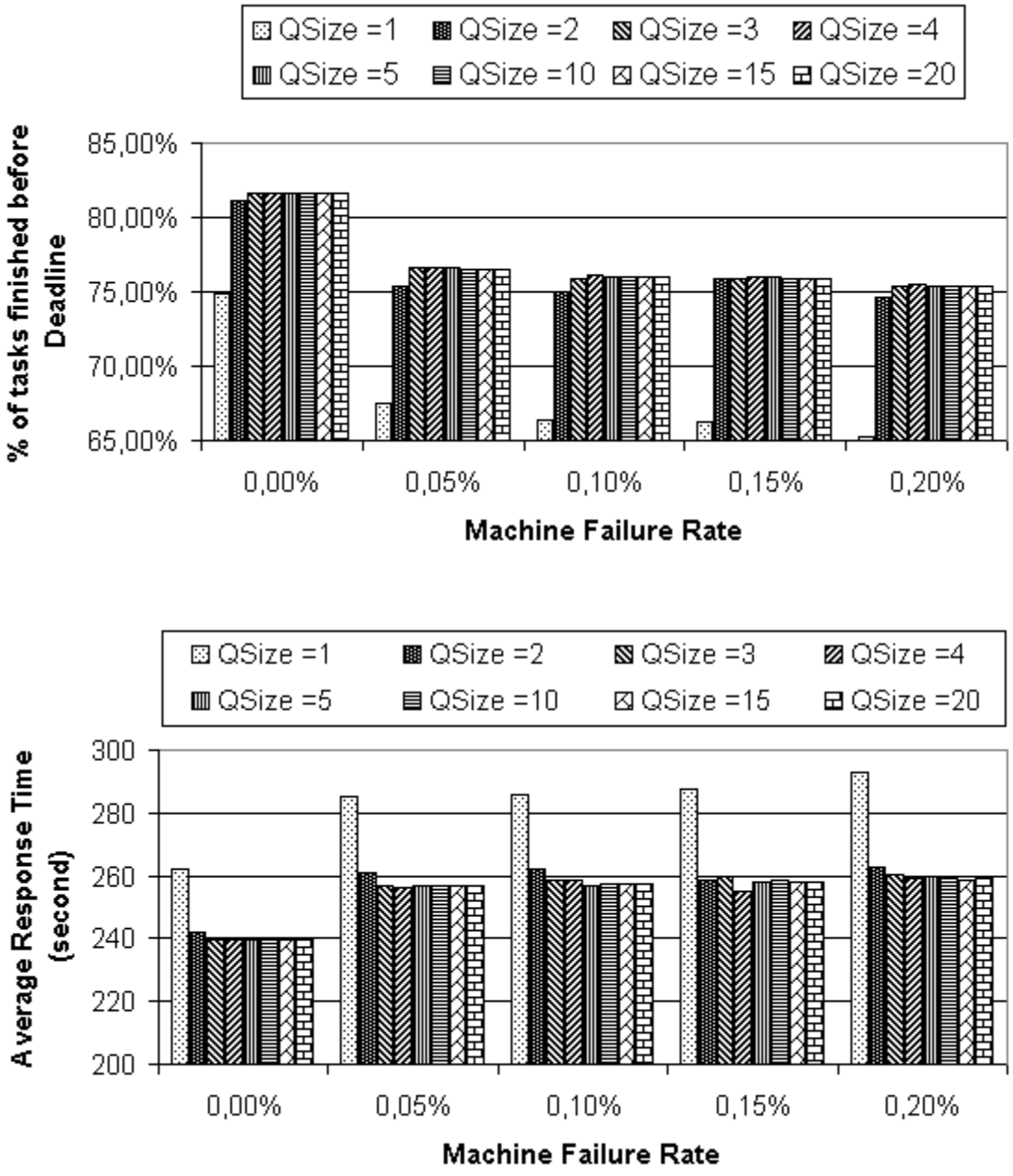


Figure 11. The impact of machine failure rate on network computing performance during simulation Phase III.

Figure 12 illustrated the impact of the machine failure rate and SWQ size on internal and external network traffic, respectively. It can be observed that as the machine failure rate decreased, the external scheduler tended to send tasks to non-local LANs for execution, which reflected in higher external network traffic. Another interesting observation was that as the machine failure rate decreased, there was a tendency for a decrease in internal network traffic as well. Since fewer machines failed, the internal network traffic caused by subtask resending also decreased, thus the overall internal network traffic dropped.

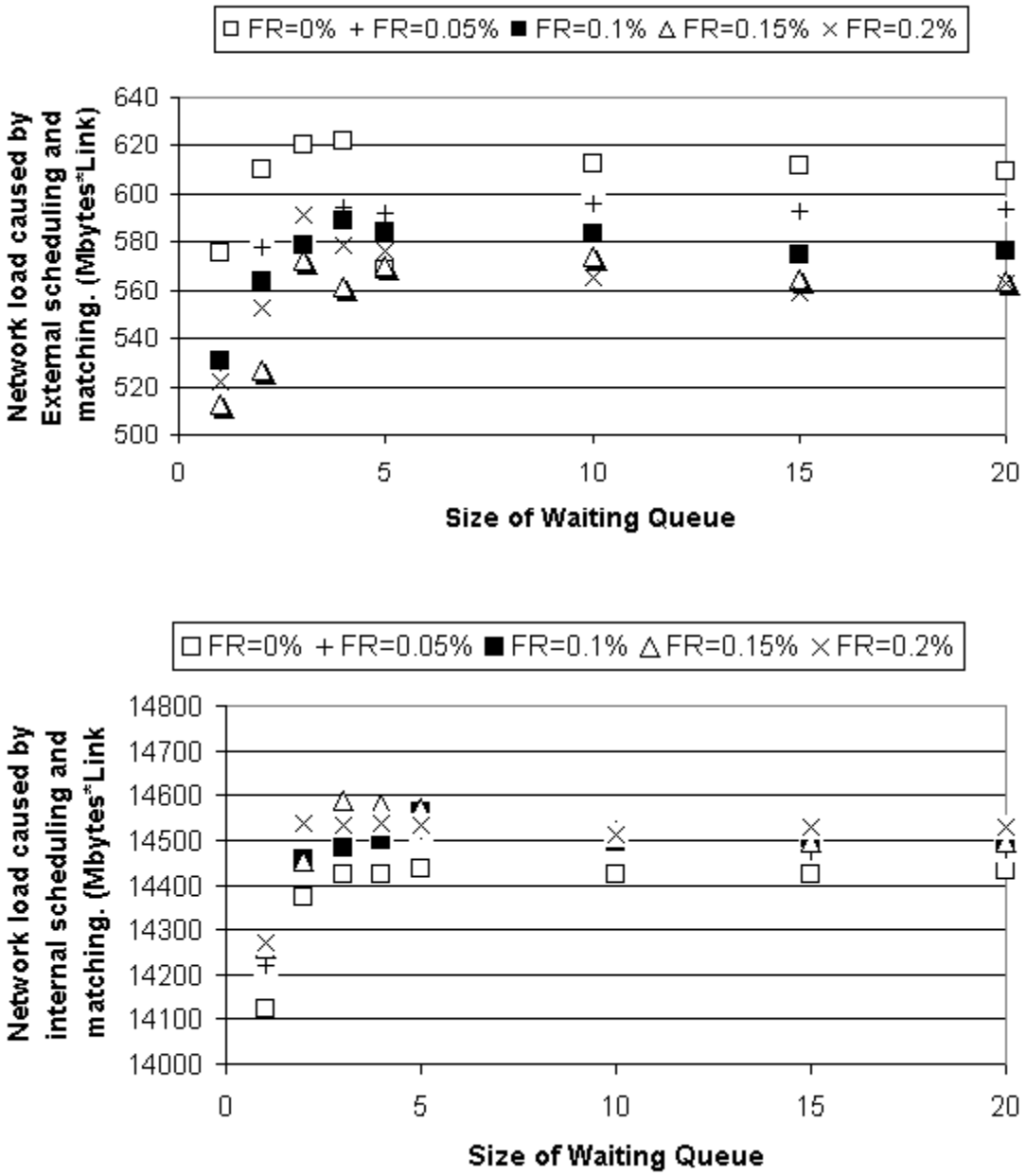


Figure 12. The impact of machine failure rate on network traffic loading during simulation Phase III.

5.4 Variation of Network Infrastructure

The fourth experimental phase was to observe how network infrastructure affects network computing performance. The simulation's environmental parameter settings were represented in Table 6. The bold and italic entries were the parameters that changed in this simulation phase. As shown in Table 5, the simulation was conducted on two different network infrastructures and various SWQ sizes. All other parameters were kept constant through this simulation phase.

Simulation phase four was different from the previous three simulation phases since this used two different network layouts. Two different network infrastructures were compared, one had 100 LANs with each LAN consisting of 10 nodes, and the other one had one LAN consisting of 1000 nodes. In order to achieve an ideal condition and eliminate the effects of network throughput and scheduling overhead, network transferring time was set to be 2000 ms regardless of task size and nodes. All other effects caused by the change in network infrastructure were not considered and were assumed identical.

Table 6. Simulation Phase IV Environmental Parameter Setting

Simulation Environmental Parameter		**Value**
Network Specification	***Number of LANs***	***{1, 100}***
	Number of nodes in each LAN	***{1000, 10}***
	Number of WAN level node	*10*
	Network transferring time	2000 ms
Machine Specification	Machine recover time	[1 sec, 1 hr]
	Machine failure rate	0
	Machine computing power	[30%, 100%]
Task Specification	Task size	[10 Kbytes, 2,000 Kbytes]
	Task result size	[10 Kbytes, 5,000 Kbytes]
	Estimated subtask execution time	[1 min., 10 min.]
	Number of subtask in a task	[10, 50]
	Max degree of DAG	[1, 5]
Run Time Specification	External scheduling time	12 ms
	Internal scheduling time	28 ms
	Total number of tasks	2000
	Size of Subtask Waiting Queue	***{1, 2, 3, 4, 5, 10, 15, 20}***
	Task interval	4000 sec
	Total number of bidding LAN	10

The impact of network infrastructure on the percentage of tasks meeting their deadlines and the average response time was illustrated in chart 1 and chart 2 of Figure 13, respectively. From Figure 13, it could be observed that the 1-LAN structure had a better network-computing performance than the 100-LAN structure. That is because in both scenarios it was the pre-requirement that a task had to be executed within a LAN. Thus, in the 100-LAN structure, a submitted task had to go through an external scheduler to locate the best LAN, and then the task could be sent. In a 100-LAN structure, an internal scheduler only had 10 nodes to assign subtasks to. But in a 1-LAN structure, the internal scheduler had 1000 nodes. Thus, in a 1-LAN network structure, the computational resources were distributed better and were utilized across the entire network, which resulted in a better network-computing performance.

However, the simulation was conducted without considering other effects. For example, managing 1000 nodes instead of 10 nodes would cause a dramatic difference in computing load on internal scheduling, which would increase the scheduling and mapping time. In the experiment, it was noticed that the execution time of a 1-LAN network is much longer than that of a 100-LAN network, which should be the result of an increase in scheduling and mapping time for each task. Therefore, scheduling time in a 1-LAN structure should be much greater than the scheduling time of a 100-LAN structure. But how big is it? How do the number of nodes affect scheduling and mapping time? This aspect is beyond the scope of this research. Notwithstanding, in this experimental phase, we ignored this effect and set the mapping and scheduling time for both network structures to be identical.

Another interesting point that needed to be mentioned was that the waiting queue size had no impact on a 1-LAN structure. The reason is that with a 1000 node computing resource, subtasks can be better distributed; therefore the waiting queue of each computer system was short enough that the SWQ size had no effect on network performance.

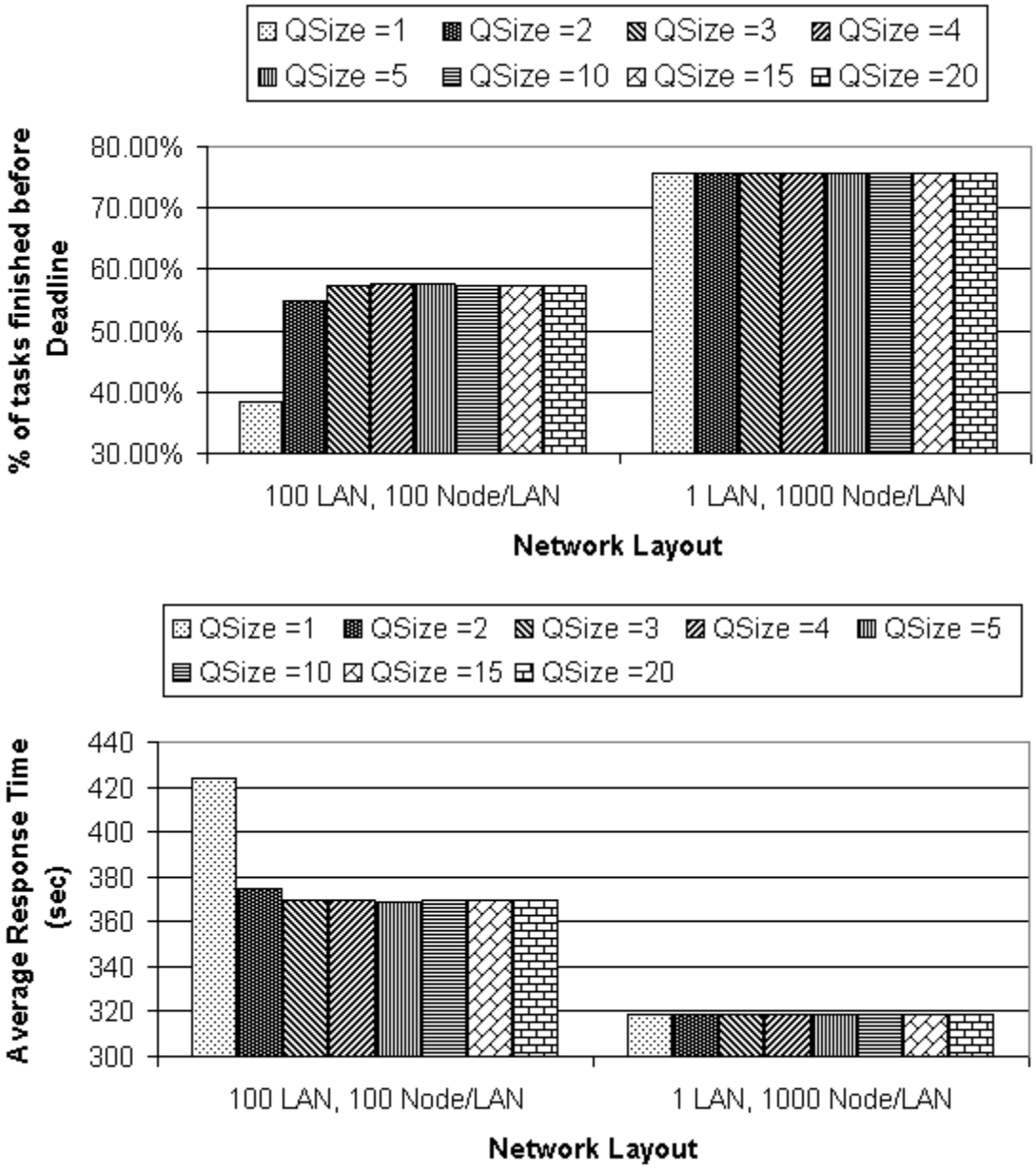


Figure 13. Impact of Network Infrastructure on Computing Performance.

6 Conclusion and Future Studies

This thesis describes a distributed dynamic scheduling algorithm of composite tasks on a grid computing system. The algorithms include two steps: external scheduling and internal scheduling. Five factors are analyzed in the simulation experiments. The factors include the number of task submissions, the task submission interval, the machine failure rate, the network infrastructure, and SWQ size.

The number of task submissions definitely affects network-computing performance. As more tasks are submitted, the network-computing performance drops because more tasks are competing for a fixed number of computing resources. Task submission intervals have two effects on network-computing performance. If a task submission interval is too short, it will cause local bidding errors; conversely, as the task submission interval increases, the number of remote bidding errors rises. A decrease in the machine failure rate helped augment computing performance because it gave network computing more computing resources and minimized the impact that machine failure has on rescheduling and re-mapping. Ideally, if the network infrastructure does not affect other environmental parameters, like scheduling and mapping time, a 1-LAN structure is superior to a 100-LAN structure. SWQ size had impacts network computing performance, choosing the best SWQ size was a balance between the negative effects of having a size that was too long or too short. Change of other factors had an impact on the SWQ size when network computing reaches its best performance.

In this research, the experimental result suggested that when conducting a design of task and subtask scheduling and mapping algorithm, the impact of number of submitted tasks, task submission intensity, reliability of computer system, and network infrastructure should be considered. The size of SWQ in where network performance reaches the best varies with the variation of these parameters. As a rule of thumb, the size of SWQ is at lease to be 2.

The task submission interval has an interesting impact on network computing performance. If a near-future network computing load can be used instead of the current network computing load during external scheduling and mapping, it will help to eliminate the problems associated with local and remote bidding errors. The impact of a network's infrastructure is another topic for future studies, such as how the nodes of a LAN can affect the scheduling and mapping time. In addition, different network specifications can affect overall network computing performance. In this thesis only a few factors were considered. Other factors include scheduling and mapping time, network bandwidth, task data size, etc. Future studies can focus on these parameters.

Acronyms

ALC	Average LAN Credibility
AMFT	Actual Machine Free Time
ASET	Actual Subtask Execution Time
ASRT	Actual Subtask Response Time
ASS	Arrive Subtask Set
FIFS	First Come First Serve
MFT	Machine Free Time
AMFT	Actual Machine Free Time
ATET	Actual Task Execution Time
ATRT	Actual Task Response Time
EMFT	Expected Machine Free Time
ESET	Estimated Subtask Execution Time
ESRT	Expected Subtask Response Time
ETET	Estimated Task Execution Time
ETRT	Task Response Time
KPB	K-Percent Best
LAN	Local Area Network
LC	LAN Credibility
MCT	Minimum Completion Time
MET	Minimum Execution Time
NOW	Network of Workstations

NOW	Network of Workstation
NTR	Network Transfer Rate
OLB	Opportunistic Load Balancing
RSQ	Ready Subtask Queue
SA	Switching Algorithm
SET	Subtask Execution Time
SP	Subtask Priority
TRT	Task Response Time
SWQ	Subtask Waiting Queue
WAN	Wide Area Network

Reference

[BaT99] L. Bajaj, M. Takai, R. Ahuja, and R. Bagrodia, "Simulation of Large-Scale Heterogeneous Communication Systems," *Proceedings of MILCOM'99*, November 1999.

[BrS01] T. D. Braun, H. J. Siegel, N. Beck, L. L. Boloni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, `` A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," Journal of Parallel and Distributed Computing, accepted and scheduled to appear 2001.

[Do96] M. Doar, "A Better Model for Generating Test Networks," IEEE Global Telecommunications Conference/GLOBECOM'96, London, November 1996. Tier source code is available at http://www.geocities.com/ResearchTriangle/3867/sourcecode.html.

[Du] X. Du, Y. Dong, and X. Zhang, Efficient Local Scheduling of Parallel Tasks and Communications on Heterogeneous Networks of Workstations. http://citeseer.nj.nec.com/246837.html

[IvÖ98] M. Iverson and F. Özgüner, Dynamic, Competitive Scheduling of Multiple DAGs in a Distributed Heterogeneous Environment. Seventh Heterogeneous Computing

Workshop. 1998.

[KrB93] P. Krueger and D. Babbar, “Stealth: A Liberal Approach to Distributed Scheduling for Networks of Workstations.” http://citeseer.nj.nec.com/ krueger93stealth.html. 1993

[KwA99] Y Kwok and I. Ahmad, " Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors ,", ACM Computing Surveys, vol. 31, no. 4, pp. 406-471, December 1999.

[MaA99]. M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, R. F. Freund, ``Dynamic Mapping of A Class of Independent Tasks onto Heterogeneous Computing *Systems,"* Journal of Parallel and Distributed Computing, Vol. 59, No. 2, Nov 1999, pp. 107-131.

[MaS99] M. Maheswaran and H. J. Siegel, “A Dynamic Matching and Scheduling Algorithm for Heterogeneous Computing Systems.” Proceedings of the 7th Heterogeneous Computing Workshop (HCW ’98), March 30, 1998 in Orlando, Florida.

[MeB98] R. A. Meyer and R. Bagrodia; “The PARSEC User's Manual,” http://pcl.cs.ucla.edu/projects/parsec/manual. 1998.

[RaG00] A. Radulescu and A.J.C. van Gemund, “Fast and Effective Task Scheduling in Heterogeneous Systems,” The 9th Heterogeneous Computing Workshop (HCW), pp.229-

238, Cancun, Mexico; May, 2000

[SiS00] H. J. Siegel and S. Ali. “Techniques for Mapping Tasks to Machines in Heterogeneous Computing Systems.” Journal of Systems Architecture, Special Issue on Heterogeneous Distributed and Parallel Architectures: Hardware, Software and Design Tools, 2000.

[YaJ] Jason Yao, “Performance Study of Multicast Protocols using the network simulator ns2,” http://www.depaul.edu/~jyao/poster/PosterAbstract.html

[ZhW92] S. Zhou, J. Wang, X. Zheng, and P. Delisle, “Utopia: A Load Sharing Facility for Large, Heterogeneous Distribited Computer Systems. ” Technical Report CSRI-257. April 1992.