SCHEDULING THREE CLASSES OF PROGRAMS ON DISTRIBUTED SYSTEMS

G. Rommel and J. A. Stankovic

COINS Technical Report 90-96
October 20, 1990

Scheduling Three Classes of Programs on Distributed Systems *

Gary Rommel and John Stankovic
Dept. Computer and Information Science
University of Massachusetts
Amherst, Mass. 01003

October 20, 1990

Abstract

In this paper we develop an algorithm that can perform distributed scheduling for three classes of programs. The three classes of programs are simple, independent programs, and two types of parallel programs – one which has significant intertask communication and one which does not. The algorithm uses negotiation and bidding and is based on the processor sharing local scheduling discipline. This is in contrast to most current distributed scheduling algorithms that assume independent sequential tasks and a local scheduling discipline based on a first come first serve policy. Via simulation we show that the new algorithm performs well in comparison to no load balancing, to a shortest queue first algorithm, and to a perfect state information algorithm.

*This work was supported, in part, under NSF grant ECS 8[illegible]

1 INTRODUCTION

In this paper, we investigate the scheduling of three classes of programs on a local area network. We delineate the scope of our work by presenting the following taxonomy.

A **distributed computer system** is a system composed of two or more computer sites connected by a communication system referred to as a subnet. Classes of programs which may execute on a distributed system include:

- **A Simple Program:** This is a program composed of a single task. Such a program cannot take advantage of the inherent parallelism of the distributed system by separating itself into subtasks. It can take advantage of the distributed system by perhaps moving to a lightly loaded site.
- **A Parallel Program:** This is a program composed of tasks which can execute at the same time. Examples of these programs are found in many numerical programs such as matrix computation and non-numerical programs such as the partitioned traveling-salesperson problem or graphics programs such as the histographing program.
- **A Distributed Program:** This is a program composed of tasks which are physically separated. Examples of these distributed programs which need not be parallel, are pipelined tasks accessed by remote send/wait message. Of course, parallel programs might also be distributed.

It is obvious that parallel programs and distributed programs are not disjoint sets. Consequently, it is not sufficient to classify a program as parallel or distributed. An important criterion in making a distributed scheduling decision will be intertask communication. Let us continue the taxonomy based on the type and amount of communication between tasks of a program.

- **A Cluster:** A cluster is a parallel program composed of a collection of tasks which communicate either frequently, or in large amounts, or via synchronous communication. In general, distributing this program is not beneficial to its execution. In this paper we study only a restricted form of clusters, (i.e., parallel programs with asynchronous communication and a uniform, but heavy, communication pattern). Examples of clusters include fork-join programs with significant asynchronous communication.
- **A Distributed Group:** This is a program which is composed of a collection of tasks that interact, but potentially benefit from executing at different sites. Fork-join programs with minimal asynchronous communication are examples of distributed groups.

Most of the distributed scheduling literature has dealt with simple (independent) programs based on local first come first serve scheduling disciplines. This work is too limited for today's distributed systems. We have developed and evaluated an effective distributed scheduling algorithm that has the following characteristics:

- it handles three classes of programs, i.e., simple programs, clusters, and distributed groups,
- it uses negotiation to schedule clusters,
- it uses bidding to deal with simple and distributed group programs,
- it is based on a processor sharing local scheduling discipline, and
- it uses periodic re-evaluation to deal with tasks in execution.

The performance evaluation of the algorithm has shown that our algorithm is superior to the no-load balancing and the shortest queue first baselines under many conditions including over a wide range and variance of cluster sizes, over a wide range of intra-program communication and site utilization, under extremes in the workload for each class of program, and for various subnet delays. Further, even without perfect state information our algorithm performs nearly as well as a perfect state information baseline.

This study also provides the following insights for scheduling clusters on distributed systems: grouping tasks of a cluster at one site should be done when site utilizations are moderate or high, clusters should not be grouped at one site when both the site utilizations and the intra-program communication are very low (given reasonable subnet delays), large clusters (those with a significant number of tasks) should always be grouped at a single site, clusters which exhibit a large amount of intra-program communication should always be grouped at a single site, and simple algorithms such as shortest queue first cannot be used indiscriminately when parallel programs are to be scheduled on distributed systems.

The remainder of this paper is organized as follows. Section 2 briefly mentions some of the past scheduling research and describes our algorithm. Section 3 presents simulation results showing the effectiveness of the algorithm. Section 4 summarizes the paper. An appendix contains the pseudo code for the algorithm.

2 A SCHEDULING ALGORITHM FOR PARALLEL PROGRAMS ON DISTRIBUTED SYSTEMS

Distributed scheduling has been treated by graph theoretic approaches [Sto77] [Sto78] [Bok79] [Lo84] [RSH79], queueing theoretric approaches [NTT87] [NT84] [ELZ86a] [MTJ89] [Rom89a], Mathematical Programming [MLT82], and heuristic approaches [ELZ86b] [ELZ86a] [LM82] [MTJ89] [RS84] [Efe82] [Sta85] [BF81] [Sta84] [SS84]. Graphic theoretic approaches are difficult to extend to networks with large numbers of sites. Queueing theoretic techniques become very complex when issues like intra-program communication are incorporated. Mathematical programming solutions for the distributed scheduling problem are in general *NP-Complete*. Finally, heuristics have been used to solve the distributed scheduling problem if sub-optimal solutions are acceptable. However, almost all of this work deals with simple programs on distributed systems where the local scheduling discipline is FCFS. Since many current systems use round-robin (or a form of it such as a multi-level feedback queue)

for scheduling, we consider the problem of scheduling programs on a distributed system composed of processor-sharing servers.

Before we present the algorithm itself, let us examine a typical distributed system with simple, distributed group, and cluster programs. Figure 1 represents a five site distributed system. The programs are labeled using a letter for the class, the first subscript for the program, and the second subscript for the task within the program. The letters s, d, and c refer to class simple, distributed group, and cluster, respectively.

In Figure 1 we show simple programs s_1, s_2, s_3, and s_4. If, for discussion, we assume queue length as a metric to determine performance, we see that programs s_1 and s_2 would probably perform better than programs s_3 and s_4. If we transfer s_4 to site 5, both s_3 and s_4 would benefit.

Continuing with describing Figure 1, we show 3 distributed groups. The first distributed group has tasks $d_{1,1}$ and $d_{1,2}$ both at site 1. These tasks are grouped at the same site and do not benefit from the parallelism of the distributed system. We expect that these tasks will be considered for re-assignment by the scheduling algorithm. The second distributed group has tasks $d_{2,1}$ and $d_{2,2}$ which are separated and exhibit a degree of parallelism. We do expect that these tasks would exhibit good performance as long as the site at which they are located is not overloaded.

In Figure 1 there are three clusters. The first cluster is composed of tasks $c_{1,1}$, $c_{1,2}$, and $c_{1,3}$. Tasks $c_{1,1}$ and $c_{1,2}$ are at site 2. Because some members of this cluster are not at site 2, in this case $c_{1,3}$, we call the subgroup composed of $c_{1,1}$ and $c_{1,2}$ a *partial cluster*. Task $c_{1,3}$ is also a *partial cluster*. *Partial clusters* are candidates for re-assignment. The second cluster is at site 1 and is composed of tasks $c_{2,1}$, $c_{2,2}$, and $c_{2,3}$. We call this configuration a *complete cluster* since the complete program is at one site. We expect this program to be exhibiting the minimum intra-cluster overhead since the communication overheads among tasks at the same site are small compared to communication among tasks at different sites. Thus, in general we would not expect to consider any of the individual tasks of this cluster for re-assignment, although in certain situations the entire cluster might be moved.

We now examine some concerns regarding scheduling clusters. We know that intra-program communication across the subnet results in a reduction in performance of the site. This is attributed to servicing the communication requests. To prevent this reduction in performance we could group the tasks of a cluster at one site. However, grouping clusters at one site prevents taking advantage of the inherent parallelism of a distributed system. Thus, in scheduling clusters we are concerned with balancing the the intra-program overheads with the parallelism of the system.

Second, suppose some tasks of a particular cluster are performing well and other tasks of that same cluster are performing poorly. How can this situation be rectified?

A third scheduling concern deals with the question of considering sites which do not have members of a cluster. If sites of the network are lightly loaded, but do not have cluster tasks, should these sites be considered in a cluster re-assignment?

Finally, an interesting situation occurs when sites become idle. Several possible scheduling approaches could be implemented. For example, the *V* system at Stanford exploits this

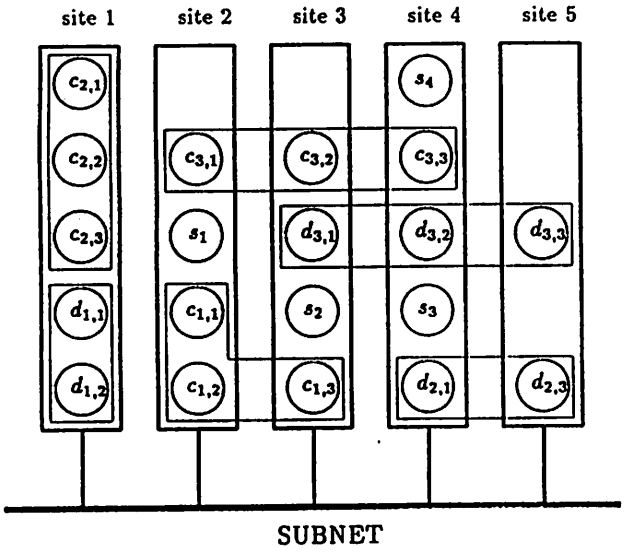

Figure 1: Newly Arriving Simple Task

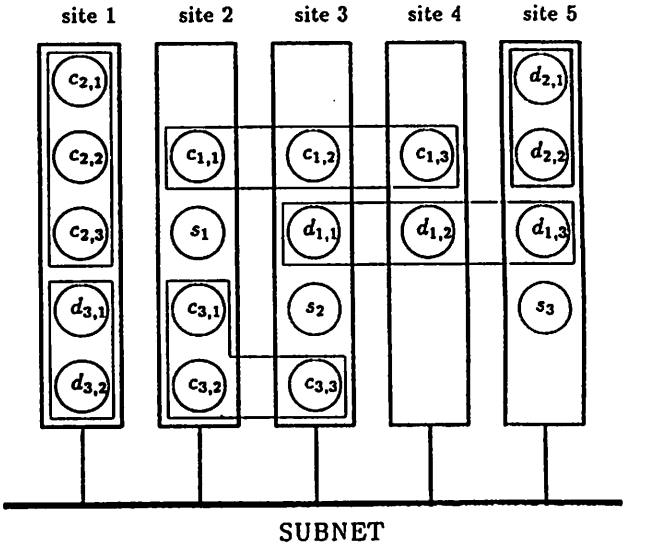

Figure 2: Newly Arriving Distributed Group

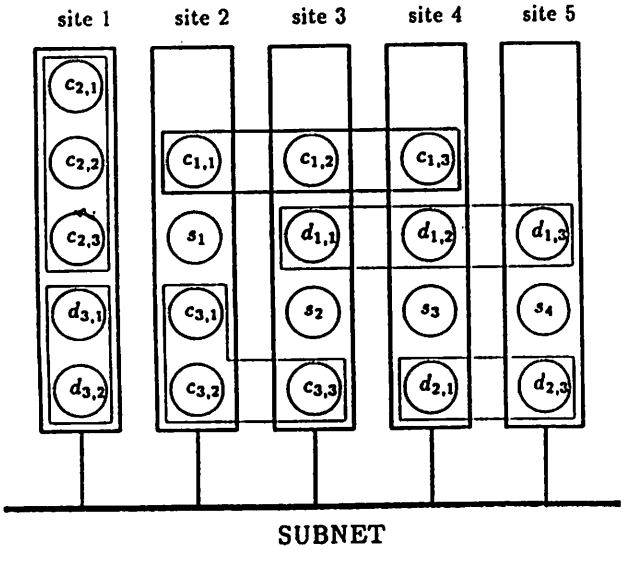

Figure 3: Newly Arriving Cluster

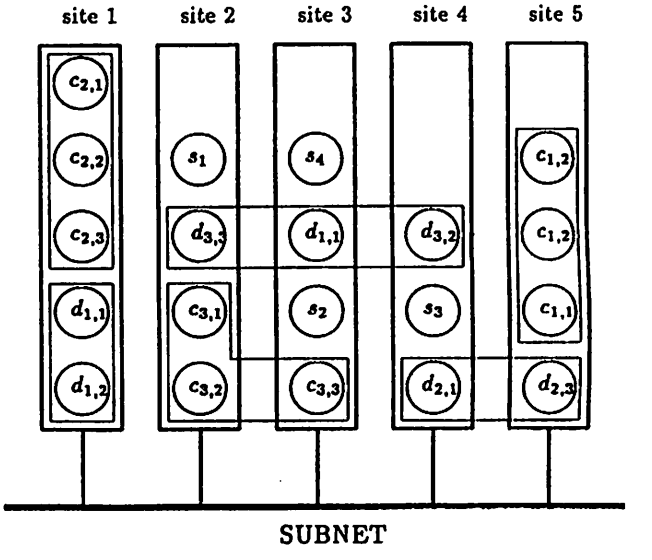

Figure 4: After First Coordination

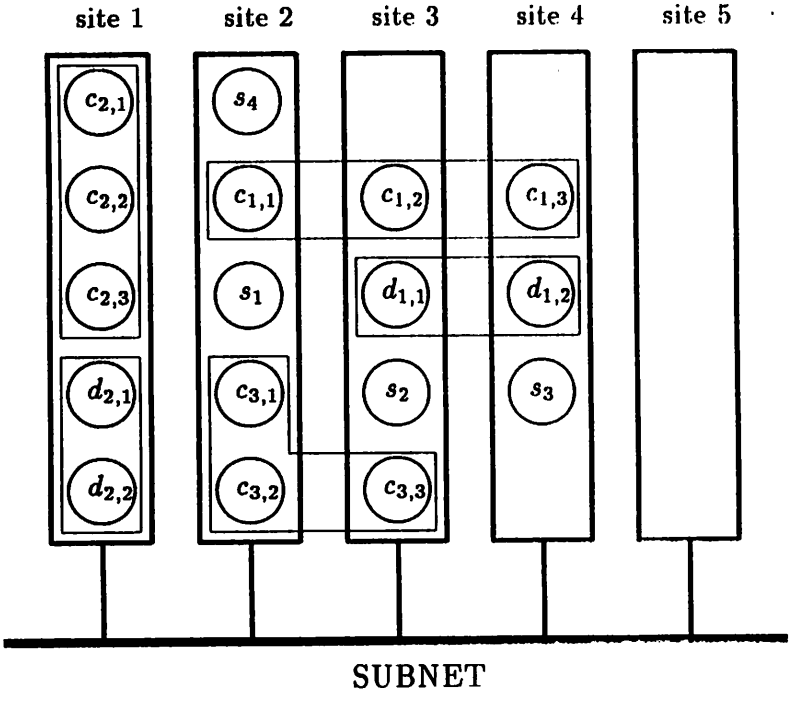

Figure 5: System with a Idle Site

situation by forwarding tasks to an idle site pool. Under a workload with clusters a reasonable extension would be to assign all the tasks of a cluster to the idle site. This eliminates the intra-cluster communication overheads.

In summary, there are four concerns in our scheduling algorithm: assigning simple programs, assigning tasks of distributed groups, assigning tasks of clusters (addressing the concerns expressed above), and taking particular care to schedule work for idle sites. Our algorithm explicitly addresses each of these issues.

Our algorithm makes several assumptions. First, our algorithm does not consider any resource requirements other than those of the site's single central processor. Second, the algorithm assumes that all communication can be treated as asynchronous communication. Third, the algorithm requires that the sites can estimate the time to complete, ETC, each class of program as well as each task. We make this assumption because we expect both future programmers and the compilers on distributed systems to know more about the service requirements of programs than those of uniprocessor systems. Fourth, the algorithm assumes that the intra-program communication can be estimated. This last assumption is supported by the fact that programs with intra-program communication would likely be pretested and analyzed for parallel execution.

2.1 The Distributed Scheduling Algorithm

In our algorithm we have a separate approach for simple, distributed group, and cluster programs. Also, the scheduler is invoked when tasks arrive at the system, periodically during execution, and when a site goes idle. We now explain the algorithm by examining its three points of invocation for each program class. We present the algorithm by example. We give its pseudo code in the Appendix.

2.2 New Program Arrivals

The scheduling algorithm is invoked upon new program arrivals. The arriving program is treated differently depending on whether it is a simple program, a distributed group, or a cluster. We assume that the system is aware of the class of the program, but if it is not then it is a simple task to assume it is a simple program and re-classify the program once it creates parallel tasks.

2.2.1 Invocation due to Simple Program Arrivals

To understand our algorithm's operation for a new simple program arrival consider Figure 1 again. Assume that s_4 is a new task in the system. To handle s_4 the algorithm first computes the estimated time to complete the simple task locally. We assume that most simple tasks are computationally short and, since processor-sharing is assumed, we know that a good estimate for the time to complete is $ETC = (N_{local} + 1)S$, where N_{local} is the local queue length and S is the service time required for the simple task. The algorithm then proceeds to compare the simple task's ETC with a fixed task threshold, Δ_{task}. If the task's ETC is below this

threshold, then we assume that the task would execute well at the local site and is assigned to the local site. In Figure 1 we see that s_4's arrival would force the task to share the central processor with four other tasks. Therefore, its $ETC = 5S$. Let's assume that $5S > \Delta_{task}$. We now consider what the algorithm does next.

If it is predicted that s_4 will perform poorly at the local site via the above formula, then an exchange of site queue lengths is initiated by the local site. First, the local site broadcasts a request for bids, RFB. After the local site has received all the bids, the algorithm assigns the task s_4 to the site with the minimum value of the ETC computed at each remote site which includes the transfer delay to reach that site. In Figure 1 we see that the site with the minimum bid would be site 5 since its queue length is 2. Assuming that the subnet delay is the same for all sites, s_4 would then be transferred to site 2. Notice that the transfer might be prevented if the transfer delay is large because the local ETC (which has no transfer delay) would be the best site.

The motivation for this feature of the algorithm is that it is simple computationally and is an approximation to the actual simple program's performance.

2.2.2 Invocation due to Distributed Group Arrivals

The next example used to describe our algorithm considers invocation due to distributed group arrivals. To use the inherent parallelism of the system, the algorithm attempts to reassign the distributed group tasks. Suppose that distributed group 2 has just arrived at site 5 as in Figure 2. At site 5 this distributed group would be unable to exhibit any parallelism. This assignment may also cause site 5 to overload. Although there are several possible reassignments, a good assignment would be to keep one of the distributed group tasks at site 5 while moving the other task to site 4, the least loaded site in the system. Before explaining how the algorithm handles distributed group tasks, we must first extend the notion of the estimated time to complete the distributed group tasks.

The Estimations in the Algorithm A basic element of the algorithm is to compute the estimated time to complete, ETC, for a task and sum of the time to complete all tasks at a site, STC. We use both of these parameters to make decisions concerning the placement of the tasks in the system. As before, the system parameter, Δ_{tasks}, is used as the threshold for the response time of tasks to determine if simple tasks are performing well. Our overall site threshold, Δ_{site}, is used as the threshold to determine if the site is performing well. That is, tasks which have an ETC over Δ_{tasks} are considered as running poorly. Sites which have an STC over Δ_{sites} are considered overloaded. From previous tests we have found that a good choice for the task threshold is the average task service time. Most of our experiments use this value.

We let $\eta_\tau(t)$ represent the remaining number of instructions of task τ at time t. We assume that at time t all tasks are ordered by ascending order of remaining instructions, $\eta_1(t) < \eta_2(t) < ... < \eta_{N_s}(t)$ where N_s is the total number of tasks at the site. The expression for ETC of the τ^{th} task at site s is

$$ETC_{s,\tau} = \nu_s \sum_{j=1}^{\tau} (N_s - j + 1)(\eta_j(t) - \eta_{j-1}(t)). \quad (1)$$

Where

N_s is the total number of tasks at site s,

ν_s is the instruction time at site s, and

$\eta_j(t)$ is the total number of instructions remaining for task j at this site at time t. The value of $\eta_0(t) = 0$.

Note: this expression is valid only for the processor-sharing discipline.

We assume that we can estimate the time to complete a task under the assumption that no arrivals occur between the time we compute the estimate and the time we use the estimate in our algorithm. This is because our subnet is assumed to be fast *w.r.t.* the inter-arrival time of groups. The $ETC_{s,1}$ is the approximate time to complete the first task. The value $ETC_{s,2}$ is the approximate time to complete the second task following the completion of the first task. The value of $ETC_{s,\tau}$ is the approximate time to complete the τ^{th} task.

The expression for STC at site, s, is

$$STC_s = \sum_{\tau=1}^{N_s} ETC_{s,\tau}. \quad (2)$$

The STC_s is the sum of all expected completion times. The motivation behind using this metric rather than the average of the expected times to complete is because the average might be dominated by the simple tasks. This occurs when we have a large number of small simple programs at a site. If the average was used as a metric for site performance, a site with a large number of simple programs would appear as lightly loaded. If a task would be placed at that site, it would not receive good performance. This is because it would share the central processor with all the simple tasks throughout the task's sojourn at that site.

We now return to the discussion of the new distributed group arrival. A loaded or busy site is define by the site's STC being greater than the site threshold, Δ_{site}. For convenience of the discussion, we will use queue length as a vehicle for depicting loading rather than STC except where stated otherwise.

The algorithm is invoked after the distributed group has arrived. It proceeds by first computing the ETC for each distributed group task based on equation (1). Each task below the task threshold, Δ_{task}, is assigned locally. For each task above the task threshold a RFB is broadcast. The algorithm then waits for the returning bids. After receiving all bids (which again account for transfer delays), the scheduler then sends out the most computationally bound task of the program to the site with the lowest bid (that site's, STC). If there are more than two distributed group tasks above the task threshold, then the scheduler sends the most computationally bound task (highest ETC) of the program to the site with the lowest bid (lowest STC) and the next most computationally bound task to the site with the next

lowest bid and so on. This procedure is repeated until all tasks above threshold are assigned to sites.

Notice that this procedure does not allow two or more tasks of the same distributed group at the same remote site unless the number of tasks exceed the number of sites. If the number of tasks exceed the number of sites, then the scheduler proceeds by assigning the tasks again by STC. The motivation for this procedure is that it is computationally simple. The distributed group scheduling can produce an assignment which exhibits a large amount of parallelism.

2.2.3 Invocation Due to Cluster Program Arrivals

For clusters, as in the case of distributed group programs, we again use the computation of ETC and STC developed in equations (1) and (2). For purposes of this example we use queue length as a vehicle for discussion rather than the full ETC and STC calculations. Now consider the case of a new arriving cluster. Figure 3 illustrates such a situation. Assume that cluster 1 has just arrived with tasks $c_{1,1}$, $c_{1,2}$, and $c_{1,3}$ at sites 2, 3, and 4, respectively. If we seek to assign the cluster to one site, then we are confronted with a dilemma. If we assign the cluster to site 2, then site 2 would have tasks $c_{1,1}$, $c_{1,2}$, $c_{1,3}$ $c_{3,1}$, $c_{3,2}$, and s_1. The load would be 6 tasks. If we assign the cluster to site 3, then this site would have tasks $c_{1,1}$, $c_{1,2}$, $c_{1,3}$ $c_{3,3}$, $d_{1,1}$, and s_2. Again the load is 6 tasks. If we assign the cluster to site 4 or 5, then again the load is 6 tasks. If a load of 6 tasks is unacceptable, *i.e.*, $STC > \Delta_{site}$, then we must do something else such as *negotiating* for a reassignment of tasks.

Let us assume that the loading of site 5 with 6 tasks is lower than that of sites 2, 3, and 4. Site 5 is selected for the cluster. But a load of six might be greater than the site threshold. The algorithm now proceeds to reduce the load at site 5 by sending tasks to sites 2, 3, and 4. This is logical since the algorithm has just before determined that the cluster tasks would be removed from these very same sites.

The algorithm may be able to reduce site 5's loading if the algorithm moves the simple task, s_4 to site 3. If the algorithm moves $d_{1,3}$ to site 2, the algorithm would drop the load at site 5 to 4 tasks. We expect that site 2 could handle this transfer since it would be losing $c_{1,1}$. This exchange would be preferable to moving the distributed group task, $d_{1,3}$ to sites 3 or 4. The reason for moving the task to site 2 rather than sites 3 or 4 is that this assignment keeps all the distributed group tasks at different site. Thus, the algorithm maintains the maximum level of parallelism for the distributed group as before.

There are other good assignments such as moving the new cluster to site 3, task $c_{3,3}$ to site 2, and task s_2 to site 4. This latter assignment is the situation when all sites in the exchange had cluster members. The example of the previous paragraph was a situation when the cluster was grouped at a site without any cluster members. Our algorithm allows for both such assignments. Let us now proceed to explain how the algorithm implements these assignments.

The algorithm for cluster arrivals begins by having each site compute the ETC based on equation (1) for a cluster task arrival. If all computed task ETC values are below the task threshold, Δ_{task}, then the cluster tasks are accepted locally. If any one of the cluster task

ETC values are above this threshold, then the site having the largest ETC value becomes the cluster coordinator and is responsible for a re-asssignment of the cluster.

The coordinator begins by broadcasting a RFB for its most computationally bound cluster task. These bids are the ETC value at the remote site for the cluster task. The coordinator then waits for the return of the bids. When all bids are returned, the coordinator forms a team of sites for the re-assignment of the cluster. The team is determined from the set of sites with the cluster members and one randomly chosen site with a bid less than the site threshold, Δ_{site} if any. For systems composed of many sites our algorithm would select more than one lightly loaded site.

Only the team sites are used in the re-assignment of the cluster. The motivation behind limiting the team is threefold. First, this procedure reduces the scope of the sites to a subset of the sites of the network. This reduces the algorithmic overheads and reduces the amount and frequency of the information required for the re-assignment. Second, the algorithm selects a re-assignment which has a good chance of minimizing task re-assignment due to the fact that some team sites have cluster members already. Third, the algorithm allows a lightly loaded or idle site to be considered for the cluster.

Once the team sites are chosen, the coordinator selects a site for the complete cluster based on the minimum bid. This site is referred to as the selected site or *s-site*. In Figure 3 site 3 might be the coordinator and site 5 might be considered the *s-site*. Next, the coordinator enters the first of possibly four phases to find a good re-assignment of tasks for the cluster. We now proceed to explain each of these phases.

Phase One attempts to re-assign only the cluster tasks to the *s-site* without additional task transfers. If the *s-site* can accept its current workload and the complete cluster, then all cluster tasks are transferred to the *s-site* and the algorithm stops. The decision is made based on the current workload and the complete cluster. To do this the algorithm estimates STC from the current tasks at the site and the arriving cluster. If the STC due to this workload is less than the site threshold, Δ_{site}, we say that the *s-site* can accept this workload. The motivation behind this phase being considered first, is that it is the simplest and prevents additional transfers.

If *Phase One* is not adequate, then and only then does the algorithm proceed to *Phase Two*. *Phase Two* seeks to transfer simple tasks of the *s-site* to other team members. This phase begins with coordinator selecting a simple task at the *s-site*. The coordinator chooses simple tasks based on shortest time to complete first. The selected simple task is now considered for transfer. The coordinator then selects a team site. If the team site can accept its current assignments and the simple task under consideration, then the simple task is marked for transfer to the team site, is now considered part of the remote site's workload, and is temporarily removed from the *s-site*'s workload. Now, if the *s-site*'s workload is lower than the site threshold, Δ_{site}, then all marked tasks are transferred to their respective sites. Otherwise, another team site is examined for the simple task under consideration. This procedure is repeated for all team sites. If no team site can be found for the simple task under consideration then the simple task is removed from consideration and another simple task is considered. Only if all the s-site's simple tasks have been considered and no acceptable

assignment occurs does the algorithm proceed to the next phase. If we consider Figure 3 again, we see that this phase might move s_4 to site 3 given that the cluster is marked for site 5.

Phase Three is designed to move partial clusters at the *s-site* to team members. The motivation behind this is the double benefit of completing the cluster at the *s-site* and completing the partial cluster at the team site. This phase proceeds as follows. First, a partial cluster is chosen from the *s-site* current workload. This decision is based on the highest ETC value for a partial cluster task. Next, a team site with members of the partial cluster's program is selected. If the team site can accept this partial cluster, then all tasks of the partial cluster are marked for transfer to the team site, all tasks of the partial cluster are removed from consideration, all tasks of the partial cluster becomes part of the team site's workload, and all tasks of the partial cluster are removed from the *s-site*'s workload. Subsequent workloads of both the team site and the *s-site* assume that these task transfers have taken place. If the new workload at the *s-site* is below threshold, then all marked tasks are transferred to their respective sites and the algorithm stops. Otherwise, another team site is chosen for the partial cluster under consideration. If all team sites have been exhausted, then the next partial cluster is chosen. Only if all partial clusters are exhausted and no acceptable assignment can be found does the algorithm proceed to the next phase.

Phase Four transfers distributed group tasks from the *s-site* to the team sites. The procedure is similar to the *Phase Two* and *Phase Three*. Distributed group tasks are selected from the *s-site* workload and are placed under consideration for transfer. A team site is chosen from those team sites without a member of the distributed group now under consideration. If team site can accept its current workload and the distributed group task, then the distributed group task is marked and treated in a similar fashion to that of the simple task. If the *s-site*'s new workload is below site threshold, then the marked tasks are transferred and the procedure stops. Otherwise, another team site is selected for the distributed group task. If all distributed group tasks have been considered and still an acceptable assignment has not been found, then the algorithm transfers all the marked tasks to their respective sites. At this point the algorithm stops.

2.2.4 Incorporation of Communication Costs

Just as for simple programs, communication cost for clusters is included in the algorithm in the ETC and STC calculations. The communication cost is translated into an increase in the number of instructions. The increase is a product the current number of instruction, the constant ψ_C, and the number of remote tasks of the cluster *w.r.t.* the local site. We assume that ψ_C has been estimated either by the programmer or the system.

2.3 Periodic Invocation

We now explain what occurs during the periodic invocation of the algorithm. The algorithm is invoked *periodically* at each site. This invocation begins by a computation of the ETC for each local task. ETC is computed for all classes of programs. The motivation behind this

action rather than simply using $(N_{local}+1)S$ is that to compute ETC for clusters we need the ETC for simple programs. We use equation 1 for this computation. Next, the algorithm forms a list of tasks in descending order by their ETC values. The algorithm proceeds by removing all tasks with the ETC values less than the task threshold, Δ_{task}. Next, the list is truncated to the top β_{number}. This is done to limit the overhead of the algorithm. Finally, for each task in this list the algorithm proceeds as in the invocation of the algorithm for a new program.

Figure 4 displays a possible result of the algorithm when applied to the tasks given by Figure 3. We see that such an assignment does not form a complete cluster 3 at one site. However, over time, the state of the system may change. Suppose that a change takes place in which s_1 terminates but $c_{3,3}$ is performing poorly. Such a situation would be addressed by the periodic invocation of the algorithm. Since $c_{3,3}$ is communicating with two task on site 2, then the partial cluster $c_{3,3}$ would be detected as executing poorly. $RFBs$ would be issued. Site 2 would respond with a low bid. It would become the *s-site*. After coordination at site 3, the partial cluster $c_{3,3}$ would be transferred to site 2.

We can consider another scenario for the periodic invocation. Again consider Figure 4. Now suppose the distributed group 3 at site 1 is executing poorly. This could be due to the fact that the site was previously an *s-site* for cluster 2. Here the site is performing well but the tasks, *i.e.*, $d_{3,1}$ and $d_{3,2}$ are not executing well. If the algorithm is invoked it might select task $d_{3,2}$ for re-assignment. Site 1 would issue an RFB. Upon evaluating the bids it would probably select site 4 for task $d_{3,2}$ due to the load of 3 tasks.

2.4 Idle Site Invocation

The *idle site* procedure is invoked when any site goes idle and periodically while the site is idle. After the algorithm is invoked, it requests state information from each site. Each site then returns a bid. The *idle site* then waits for the bids. The bids are the remote STC values. After receiving all bids, the algorithm randomly selects one site with an STC greater than the site threshold, Δ_{site}. The motivation behind this procedure is to select a highly loaded site while selecting a site which is not considered by another idle site. We refer to this task as the *c-site*. Once the *c-cite* is chosen, then the algorithm obtains the *c-site*'s task lists. Now, the *i-site* forms two lists: the candidate list, CL, and the marked list, ML. The candidate list contains all tasks at the *c-site*. The marked list contains those tasks marked for transfer from the *c-site* to the *i-site*. The *i-site* computes an *i-site* threshold, Δ_{i-site}. This is the minimum of either the site threshold, Δ_{site} or half the STC of the *c-site*.

The *i-site* procedure now orders the CL based on a descending order of task ETC values. Choosing the first task for consideration for transfer, algorithm determines the task's program class. If the program class is a simple program, then the algorithm examines the *i-site*'s ability to accept it's current workload and the simple task. If it can accept the workload, i.e., the *i-site*'s STC is less than Δ_{i-site}, then the task is marked for transfer to the *i-site*, removed from the *c-site*'s workload, assigned to the *i-site*'s workload, and the algorithm selects another task. If the *i-site* cannot accept this or there are no more additional tasks, then algorithm transfers all marked tasks and stops.

If the program class is a distributed group program, then the algorithm checks the marked list for tasks which are from the same program. If there are such members, the algorithm selects another task. If there are no such members, then the algorithm examines the *i-site*'s ability to accept it's current workload and the distributed group task. If it can accept the workload, i.e., the *i-site*'s STC is less than Δ_{i-site}, then the distributed group task is marked for transfer to the *i-site*, removed from the *c-site*'s workload, assigned to the *i-site*'s workload, and the algorithm selects another task. If the *i-site* cannot accept this or there are no more additional task, then algorithm transfers all marked tasks and stops.

If the program class is a cluster, then the algorithm determines all sites with tasks of the cluster, then the algorithm examines the *i-site*'s ability to accept it's current workload and the cluster. If it can accept the workload, i.e, the *i-site*'s STC is less than Δ_{i-site}, then the complete cluster is marked for transfer to the *i-site*, the cluster tasks are removed from the *c-site*'s workload, the cluster tasks are assigned to the *i-site*'s workload, and the algorithm selects another task. If the *i-site* cannot accept this cluster or there are no more additional tasks in the CL, then algorithm transfers all marked tasks and stops.

Consider Figure 5. We see that site 5 is idle. Sites 1 and 2 are busy while sites 3 and 4 are performing well. The algorithm would be invoked when site 5 became idle. If site 2 was chosen as the *c-site*, then tasks $c_{1,1}$, $c_{3,1}$, $c_{3,2}$, $c_{1,1}$, s_1 and s_4 would be candidates for transfer from site 2 to site 5. In addition, task $c_{3,3}$ would be a candidate for transfer to site 5 if cluster 3 was transferred to site 5. Likewise, tasks $c_{1,2}$, and $c_{1,3}$ would be candidates for transfer to site 5 if cluster 1 is transferred to site 5. Let us for convenience again use queue length as our metric. Let us assume that $\Delta_{site} = 4$, then $\Delta_{i-site} = 3$. If we assume that the cluster task ETC values are greater than simple programs, then the two clusters would be considered first for transfer. The algorithm might find that the ETC value for the cluster 3 to be greater than cluster 1. In which case tasks $c_{3,1}$, $c_{3,2}$, and $c_{3,3}$ would be marked for transfer to site 5. No additional tasks would be considered since the Δ_{i-site} has been reached. Now the re-assignment would have the *i-site* with $c_{3,1}$, $c_{3,2}$, and $c_{3,3}$, and the *c-site* would have $c_{1,1}$, s_1, and s_4. This re-assignment results in a load of 3 for site 2, 3 for site 3, 3 for site 4, and 3 for site 5. The re-assignment has resulted in a completed cluster 3 at site 5. It is a good assignment in the sense that the communication overhead is reduced due the completion of cluster 3 and the system is nearly balanced due to the approximate equalization of loads. We see that site 1 was excluded from the re-assignment since site 1 did have any tasks of any of the clusters at site 2. See the Appendix for the pseudo code for all parts of our algorithm.

3 PERFORMANCE EVALUATION

In this section we evaluate our algorithm's performance considering the following issues: cluster size, extremes in workloads of different classes of programs, and a wide range of utilizations. Although not reported here due to space limitations, we also illustrated the usefulness of the four phases of the algorithm, the idle site procedure, the periodic procedure, and the selection of the best site for an arriving cluster. In analyzing the results we used two methods of comparison: a simulation baseline using no load balancing and a simulation

baseline using scheduling based on the shortest queue length. For a complete set of results see [Rom89b].

We use the *method of replicated* runs with a 9 degree of freedom *student t distribution* to simulate our model and obtain confidence intervals. To estimate the transient phase contribution to our sample data we use Kobayashi's *transient time equation*[Kob76]. Our algorithm tuning parameter selection may be found in [Rom89b]. Here we simply note that these parameters are robust and that the parameter settings become more robust as the number of sites in the system increases.

To attempt to be realistic, we use actual service time parameters measured and reported in the literature as well as from measurements on our own system at the University of Massachusetts. We take our task sizes and task transfer times from measurements in the literature. We have computed the actual overhead of our algorithm from empirical results. The simulation model is composed of five sites.

For simplicity we assume task times are exponential with a default mean of 30 seconds for the simple task service times. For distributed group tasks and cluster tasks we assume a range of service times from 30 to 300 seconds. We base this assumption on the fact that large programs have been reported to be between an order of magnitude to two orders of magnitude greater than that of simple programs [LO86].

We assume that programs enter the system according to a Poisson process with rate λ. The probability of a simple program is α_1, a cluster α_2, and a distributed group program, α_3 such that $\alpha_1 + \alpha_2 + \alpha_3 = 1$. The $\alpha_i's$ are considered as input parameters which we vary in our testing.

The independent parameter of the model is site utilization. We determine the interarrival time of the program by the utilization of the sites. A simple program is always assumed to have only one task. The number of tasks per cluster, $m_{cluster}$, is an arbitrary system parameter. We vary this parameter from $1 \leq m_{cluster} \leq 15$. The number of tasks per distributed group program, $m_{distributed}$, is an arbitrary system parameter. We set the default for this parameter at two unless otherwise stated. We assumed that the time to move a task from one site to another is given by

$$\chi^{\tau}_{delay} = \chi_{size}\chi_{time}, \tag{3}$$

where χ_{size} is the size of the task in kilobytes, and χ_{time} is the time per second for the subnet to move a kilobyte task. To obtain practical results we use the V system data cited in [TLC85], where $\chi_{time} = 0.003 seconds/Kilobyte$, and the range of χ_{size} is 0.5K byte to 70K bytes. Since no distribution was cited in the [TLC85], we assume an exponential distribution for χ^{τ}_{delay}. We use 0.01 seconds as a default mean for χ^{τ}_{delay} in our runs.

Cheriton and Zwaenepoel [CZ83] recorded for a 10 MB LAN that the CPU overhead was 3.80 milliseconds per packet. We model this overhead by including in each task an amount of additional service time. The model for the overhead assumes a constant, ψ_C such that the additional amount of service time is equal to the product of the number of remote tasks of the cluster and ψ_C. We assume that the overhead influence changes as tasks arrive, depart, or complete. Finally, it should be pointed out that except for the obvious dependence on the

speed of the subnet, ψ_C does not depend on χ^T_{delay}.

3.1 Number of Tasks in a Cluster

To gain some insight into our algorithm we vary the number of tasks in a cluster, $m_{cluster}$, while maintaining the same average service time per cluster program, $\chi_{program}$. Each task service time is an independent exponentially distributed random variable with mean $\chi_{program}/m_{cluster}$. We have chosen $\chi_{program} = 150$ seconds. In our experiments the cluster sizes are 1, 3, 5, 10, and 15. The cluster average task service times are 150 seconds, 50 seconds, 30 seconds, 15 seconds, and 10 seconds for $m_{cluster}$ at 1, 3, 5, 10, 15, respectively. The task threshold, Δ_{task}, is set 3.0. We assume that the value of δ_{alg} (the cost of executing the algorithm) was 0.0. We also performed a second experiment with all the same parameter settings, but with the value of δ_{alg} set to 0.01 (the cost measured on a microVAX).

The distributed groups are composed of 2 tasks. There are 10 subruns each of 10000 seconds. The probability of a simple program, α_1, is 0.588, of a cluster program, α_2, is 0.118, and of a distributed group program, α_3, is 0.294. This configuration is designed such that each class program service time is identical. The site threshold, Δ_{site}, is 300.00. We assume that the local STC's are updated every 10 seconds. The delay to transfer a task across the subnet is 0.3 seconds. The periodic procedure and the idle site procedure are both activated. We set $\beta_{period} = 250$ seconds.

In the experiments the average program interarrival time to the system is 17.66 seconds. Given our subrun simulation time of 10000 seconds we have approximately 1000 tasks per subrun with $m_{cluster} = 5$. The average throughputs are 0.0333, 0.0067, and 0.0166 programs/second for simple programs, clusters, and distributed groups, respectively.

Figure 6 presents the results of the first experiment. Figures 6(a),(b),(c), and (d) display the simple, cluster, distributed group, and average program response times, respectively. We show the confidence intervals in Figure 6 (d).

From Figure 6 several observations may be determined.

1. First, our algorithm outperforms the no-load balancing, *nlb*, and *shortest queue first*, *sqf*, algorithm as the number of tasks in clusters increases.

   To explain the effect the number of cluster tasks has on performance, we consider two aspects of the problem. First, if we increase the number of tasks while maintaining a constant program service time, we are in fact producing an Erlang service time distribution with lower coefficients of variance as we increase the number of tasks. As we increase the value of $m_{cluster}$, we approach a constant service time per task in each program. A constant task service time maximizes the intra-cluster communication. This, in turn, increases the communication in the system when the cluster tasks are distributed.

   The second aspect of the problem is associated with both the *nlb* and *sqf* algorithms. Using these algorithms as we increase the number of tasks in a cluster while maintaining a constant number of sites, we increase the likelihood of producing an assignment with

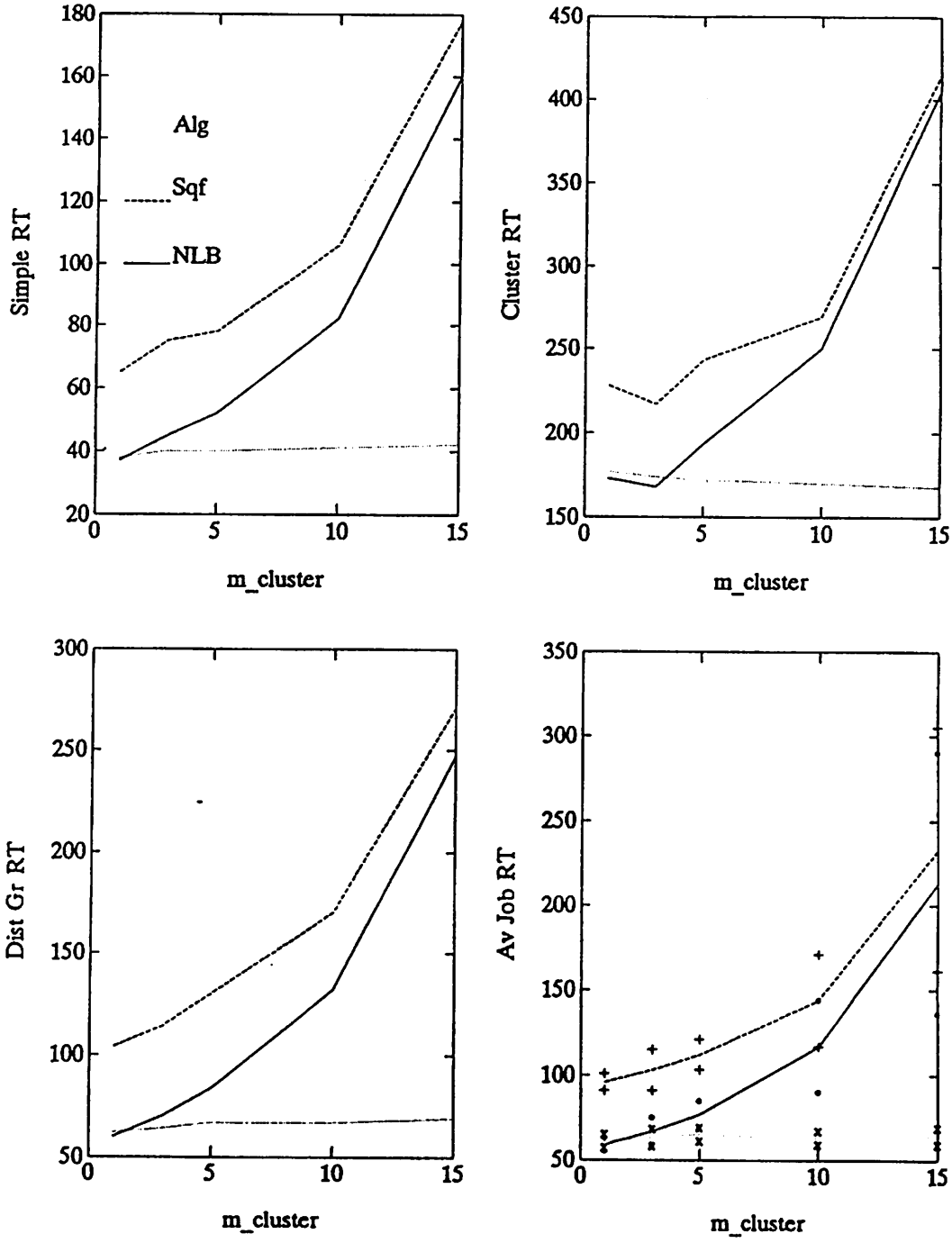

Figure 6: Performance Versus Cluster Size

distributed cluster members. (This result is supported theoretically by [Rom89a].) This results in an increase in communication overhead. If we observe Figure 6 with $m_{cluster} = 1$ and $m_{cluster} = 3$ we see that all three algorithms have nearly equal response times. It is only at values of $m_{cluster} \geq 5$ that the *nlb* and *sqf* algorithms produce much worse results than our algorithm.

2. Second, our algorithm is relatively insensitive to the number of tasks in a cluster at least up to 15.

   Both *sqf* and *nlb* approaches are sensitive to the cluster size because there is no explicit consideration of either cluster size or the overhead of cluster communication. On the other hand our algorithm explicitly considers both through the use of *ETC*. This is evident by the nearly flat curves of found in Figure 6 associated with our algorithm.

3. Third, our algorithm does not exhibit the processor-sharing phenomena of an increased response time due to an increase in the number of tasks per program.

   Processor-sharing was shown in [Rom89b] to exhibit extremely poor performance for large numbers of tasks per cluster. However, because our algorithm uses both *ETC* and *STC* the performance is nearly independent of this phenomena.

We do not present the results of the second experiment due to space considerations. Briefly the data from the second experiment verifies the first set of results and indicates the following:

1. Our algorithm's performance does not depend significantly on the overhead of execution of the algorithm. In most cases there was little decrease in performance due to the overhead of the algorithm. We saw that the greatest percentage increase in response time is only 8% for distributed groups when the cluster size is one (not really a cluster, but used to show the effect of assuming a cluster when the program is not a cluster). This is somewhat expected since there are no benefits to our algorithm a cluster size of one.

2. When the cluster size exceeds 5, we saw the benefits of the algorithm outweigh the overheads.

3.2 Extremes in Workloads of Different Classes of Programs

As we have seen, our algorithm has performed well under conditions in which clusters play an important role. A question of concern is how well does our algorithm perform as one class of program becomes dominant in terms of computational requirements. For example, suppose a system is composed of all clusters. Would our algorithm perform well or would the baselines perform better? The same questions can be asked for simple programs and distributed group programs. This is an important issue since good performance under wide range of workloads establishes the robustness of the algorithm itself.

To establish the robustness of our algorithm we set up five extreme workloads.

- Our first experiment assumes a workload in which 100 % of the computational requirements are due to clusters. We assume all group arrivals are Poisson. In the results we denote this experiment as *100 c*.

- The second experiment assumes a workload in which 80 % of the computational requirements are due to distributed group programs and 20 % of the computational requirements are due to clusters. We assume all group arrivals are Poisson. In the results we denote this experiment as *80 d*.

- The next experiment is the extreme workload of 100 % distributed group programs. In the results we denote this experiment as *100 d*.

- To gain some insight into the algorithm under extremes in simple programs, we consider the case in which the simple program computational requirements represent 90 % of the workload while the cluster programs represent only 10 % of the computational workload. In the results we denote this experiment as *90 s*.

- The final experiment considers a workload in only simple programs. In the results we denote this experiment as *100 s*.

Each experiment compares our algorithm to both the no-balancing and the shortest queue first algorithm. In the shortest queue first algorithm we consider subnet delays. The mean of the subnet delay, χ^{τ}_{delay} is set at 0.3 seconds for all experiments. The actual subnet delays for each task is then drawn from an exponential distribution with the corresponding mean. The number of tasks in a cluster is fixed at 5. The number of tasks in a distributed group is 2. The subrun time is 10000.0. Site utilization without intra-program communication is 0.6. We use $\psi_C = 0.2$ where applicable.

The site threshold is 300.0 seconds. The task threshold is set at 3.00. The time between periodic invocations is 250.00 seconds The task service times are determined from an exponential distribution with a mean of 30.0 seconds. The periodic and idle site procedures are activated. We assume that $\delta_{alg} = 0.0$.

Finally, we use two forms of the the shortest queue first algorithm. The first assumes no transfer delays and is denoted as *sqf*, while the second assumes transfer delays and is denoted as *sqf'*.

Figure 7 presents the average program response times under extremes in workloads per class of program.

The results support the fact that our algorithm is superior to all baselines under the *100 c*. Our algorithm, is nearly 50 % better than the best baseline, *sqf*.

The data clearly supports the fact that our algorithm is superior to no-load balancing under extremes. A good illustration of this is the results from *90 s* and *100 s*. Using our algorithm with the workload of *90 s* the average program response time is nearly 50 seconds. Under that same workload the programs experience nearly a 75 second response time for *nbl*. In the extreme case of *100 s* we see that our algorithm is better than all baselines.

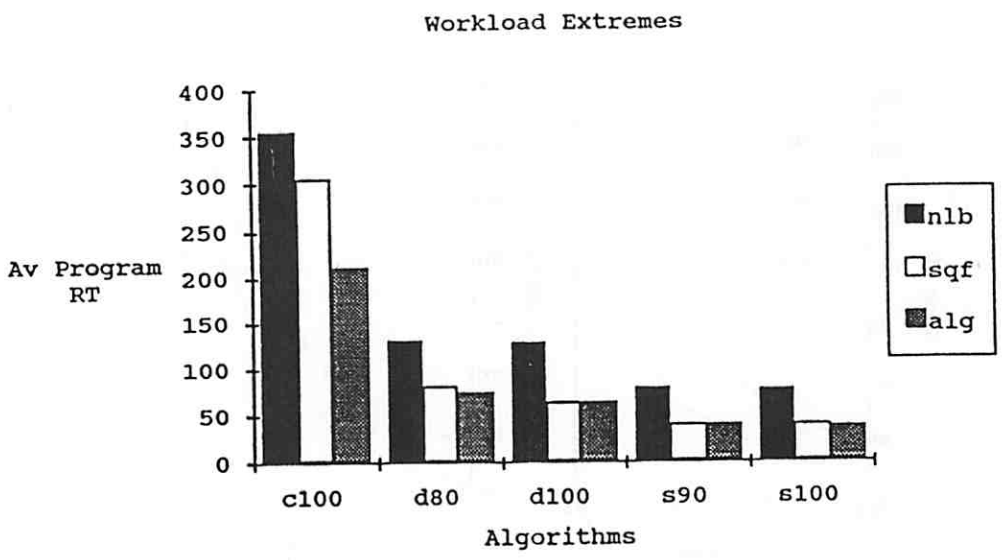

Figure 7: Av. RT Under Extremes in Workloads

Under extreme workloads our algorithm performs better than either baseline based on average program response time. The reason for this performance improvement under extremes in simple programs is that our algorithm schedules simple programs based on time to complete, $(N+1)S$. This has been shown to be optimal. Likewise, under extremes in distributed group workloads our algorithm considers both the workload at the sites and the nature of the distributed group. The result of our algorithm is to schedule distributed groups at separate sites in general. If we consider the *80 d* workload we see that the average program response time under our algorithm is 65 seconds while under shortest queue first algorithm it is 73 seconds and nearly double that value of the *nlb*. Finally, because our algorithm groups all tasks of a cluster, we do see at *90 s* the shortest queue first algorithm does have better cluster response times. If the intra-program communication was higher, this would not be the case. Whether or not our algorithm outperforms the shortest queue first algorithm is a function of several parameters. Among the most significant are the values of $m_{cluster}$, ψ_C, and ρ. Finally, at nearly all extremes our algorithm outperforms the other baselines. This is because we selectively address the scheduling needs of each class of programs.

We may conclude the following about our algorithm.

- Under heavy simple and cluster program workloads our algorithm performs much better than either baseline.
- Under heavy distributed group workloads our algorithm performs as well as *sqf* but much better than either *sqf*' or *nlb*.
- Finally, since the algorithm performance well under these extremes and under workloads dominated by clusters, we conclude that our algorithm is a robust under many various workloads.

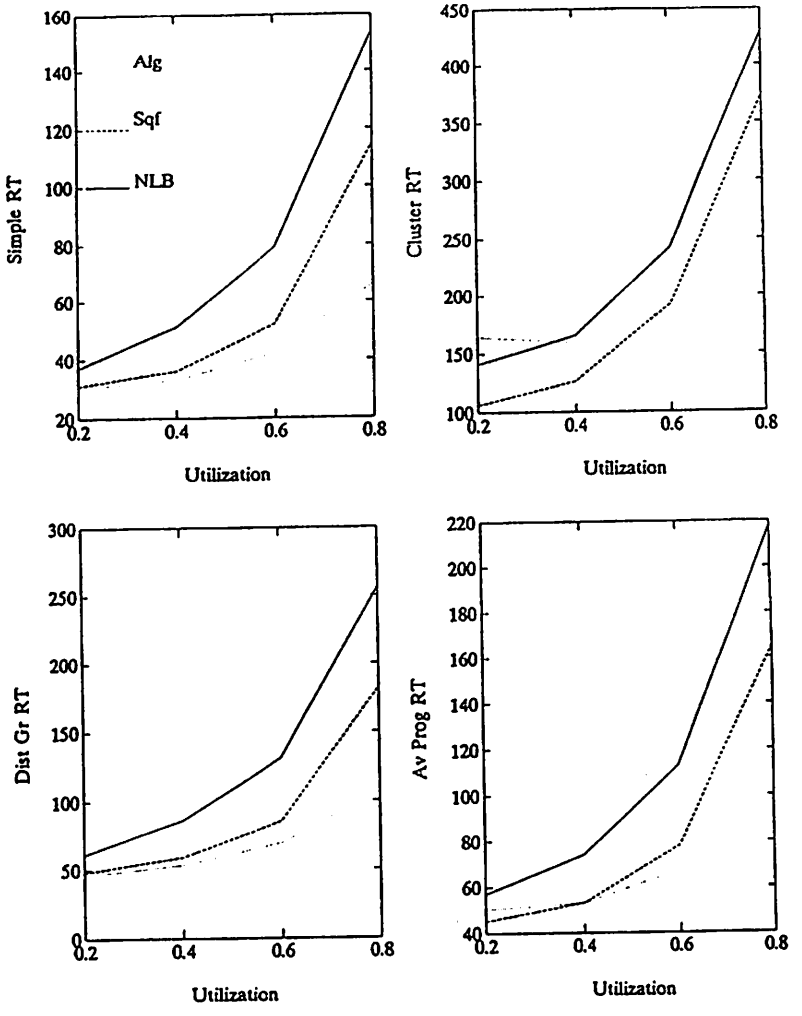

Figure 8: Vary Utilizations

3.3 Vary Site Utilizations

A major concern for any scheduling algorithm is its ability to perform well under light, moderate and heavy loads. For most of the previous experiments we have maintained a moderate load of 0.60 utilization. In this section we examine the system performance as a function of the site utilization. Due to space limitations we give the results in graphs (Figure 8) and then present the high level conclusions of our experiments.

1. Under utilizations less than 0.40 our algorithm does not perform as well as the *sqf* algorithm.

   The justification for the larger cluster response times under our algorithm as opposed to the shortest queue first algorithm is strictly due to the fact that under very low utilizations communication overheads are not important. When all sites are lightly

loaded, assigning tasks of a cluster to different sites allows the program to proceed in parallel. Thus, the reduction in cluster response time and the conclusion that at very low utilizations it is preferable to use the *sqf* algorithm rather than our algorithm.

2. Under a slightly higher utilization, $\rho = 0.4$, our algorithm begins to perform at a superior level to both simple and distributed programs. For example, for utilization $\rho = 0.4$, the simple response time is 33 seconds, 36 seconds, and 51 seconds for our algorithm, *sqf* and *nlb*, respectively.

   The reason that our algorithm is much superior to the baselines at utilizations > 0.4 is for the following reasons:

   (a) At these utilization the communication overheads dominate the performance of the system. A small increase in utilization due to communication shifts a moderately utilized system into a heavily utilized system, and a heavily utilized system into an overloaded one.

   (b) A moderate or heavily loaded system gives the periodic procedure an opportunity to be involved in scheduling more than under lightly loaded conditions. This reduces the average response times under our algorithm.

   (c) Any benefits of parallelism of clusters are overshadowed by the disadvantages of increasing the loading. We see this when we contrast *sqf* to our algorithm at high utilizations.

3. At high utilizations, our algorithm is superior for all program classes. For example, at $\rho = 0.8$ the average response times are 98 seconds, 165 seconds and 218 seconds for our algorithm, *sqf* and *nlb*, respectively.

   The reason for this very poor performance of the *sqf* algorithm can be explained from the results found in [Rom89a]. Here it was demonstrated that even at high utilizations (≈ 0.9) the *sqf* algorithm would still attempt to load balance for systems composed of five or more sites when bulk arrivals occurred. In fact, the probability of load balancing is nearly ≈ 1 for high average site utilizations. If we add the cluster overhead into the problem, we see that the *sqf* algorithm still separates the cluster tasks which in turn causes the very poor performance.

4 Summary

In summary, the algorithm that we have developed handles simple programs, distributed groups and clusters. It combines negotiation (which seems to be necessary for large parallel programs), and bidding. It is based on the processor sharing discipline. The main performance results indicate that our algorithm is superior to both no-load balancing and the shortest queue first baselines in all the following situations:

- as the number of tasks in a cluster increases,

- as the variance of the number of tasks in a cluster increases,
- as the amount of intra-program communication increases,
- as site utilization increases,
- under extremes in workloads as a function of program class,
- as subnet delays increase, and
- even without perfect state information our algorithm performs nearly as well as the perfect state information baseline.

While not shown in this paper, regarding the performance of various components of our algorithm (see [Rom89b]):

- The idle site procedure in general improved performance for all classes of programs over a wide range of workloads.
- The periodic procedure assisted the performance over a range of program workloads.
- *Phase One* was found to be extremely important when intra-program communication is considered.
- Phases Two, Three and Four each improved the performance over a limited, but different range of program workloads.

5 Appendix Containing Pseudo Code

This appendix contains the pseudo code for our algorithm. Figure 9 gives the algorithm's procedure for handling simple programs. Figure 10 gives the algorithm for new distributed group programs. Newly arriving clusters are handled by the pseudo code explained by Figure 11. Periodic invocation of the algorithm is described in Figure 12. Finally, Figure 13 describes the algorithm for idle site invocations.

Invocation due to Newly Arriving Simple Program

```
begin
 compute ETC = (N_local + 1)S
 if ETC < Δ_task
  then
   keep the task locally
  else
   For large networks Broadcast Request for Bids
   Wait for the Bids
   Transfer the task to the site with the lowest ETC
 end_if
end
```

Figure 9: Algorithm Invocation Due to Newly Arriving Simple Program

Invocation due to Newly Arriving Distributed Program

```
begin
 compute STC
 if STC < Δ_site
  then
   keep the task locally
  else
   For large networks Broadcast Request for Bids
   Wait for the Bids
   Transfer the task to the site
    with the lowest STC and without tasks of the same program
 end_if
end
```

Figure 10: Algorithm Invocation Due to Newly Arriving Distributed Group

Invocation due to Newly Arriving Cluster

begin
find the largest task ETC of the cluster
if $ETC < \Delta_{task}$
then keep the task locally
else
begin coordinator
For large networks Broadcast Request for Bids
Wait for the Bids
Form a team from the sites with task members of this cluster
and some number of randomly chosen sites with $bid < \Delta_{site}$
Select one site for the entire cluster based on the minimum Bid as the s-site
mark all remote cluster tasks for transfer to the s-site

Phase One

begin Phase One Cluster assignments without additional exchanges
if the s-site with its current assignments plus
all cluster tasks is below Δ_{site}
then
transfer all marked cluster tasks to the s-site
stop
end_if
end Phase One

Phase Two

begin Phase Two-Cluster assignments with Simple Task transfers
for each simple task at s-site
while task is under consideration
repeat
pick a team site not previously examined
if the team can accept its current assignments plus this simple task
then
mark this task for transfer
remove this task from the s-site's assignments
remove this task from further consideration
add this task to the team site's assignments
if the s-site with the new assignments is below Δ_{site}
then
transfer all marked tasks
stop
end_if
end_if
until all sites have been examined
end_while
end_for
end Phase Two

Invocation due to Newly Arriving Cluster(continued)

```
Phase Three
begin Phase Three-Cluster assignments with Partial Cluster transfers
  for each partial cluster at s-site
    while the partial cluster is under consideration
     repeat
      pick a team site not previously chosen but
        with a member of the partial cluster's group
      if the team can accept its current assignments and
        the partial cluster's task
        then
          mark the tasks of the partial cluster for transfer
          remove the partial cluster from further consideration
          remove all tasks of this partial cluster from the s-site's assignments
          add these tasks to the team site's assignments
          if the s-site with the new assignments is below Δ_site
            then
              transfer all marked tasks
              stop
          end_if
      end_if
     until all sites have been examined
    end_while
  end_for
end Phase Three

Phase Four

begin Phase Four-Cluster assignments with Distributed Group Task transfers
  for each distributed group task at s-site
    while task is under consideration
     repeat
      pick a team site not previously examined and
        without a member of the distributed group under consideration
      if the team can accept its current assignments plus
        this distributed group task
        then
          mark this task for transfer
          remove this task from further consideration
          remove this task from the s-site's assignments
          add this task to the team site's assignments
          if the s-site with the new assignments is below Δ_site
            then
              transfer all marked tasks
              stop
          end_if
      end_if
     until all sites have been examined
    end_while
  end_for
end Phase Four

  Transfer all marked tasks
 end coordinator
end cluster invocation
```

Figure 11: Algorithm Invocation Due to Newly Arriving Cluster

Periodic Invocation

begin
For all tasks at this site, compute $ETC's$ based on the task program class
Form a list of tasks in descending order by ETC value
Remove from this list all tasks with $ETC < \Delta_{task}$
Limit the list to β_{number} tasks
for each task in the list proceed as in the
the invocation due to newly arriving tasks
end

Figure 12: Periodic Algorithm Invocation

Invocation due to Idle Sites

begin
Call the idle site, i-site
Broadcast request for bids
Wait for Bids
Randomly select one site with
$STC > \Delta_{site}$.
Call this site the c-site
Using c-site's task information form two list:
One of candidates, CL
the other for marked tasks, ML, for transfer
Compute the idle site threshold,
$\Delta_{i-site} = min(\Delta_{site}, STC_{c-site})$

for each task in the candidate list select a task
case task of type

simple:
if the i-site can accept its current workload and
the simple task
then
mark this simple task for the i-site
end_if simple

distributed:
if the i-site can accept its current workload and
the distributed group task and
no other members of the distributed group are present
then
mark this distributed group task for the i-site
end_if distributed

cluster:
if the i-site can accept its current workload and
the complete cluster
then
mark all remote cluster tasks for the i-site
end_if cluster

end_case
end_for
transfer all marked tasks
stop
end idle

Figure 13: Invocation of Algorithm for Idle Sites

References

[BF81] R. Byrant and R. Finkel. *A Stable Distributed Scheduling Algorithm.* IEEE Computer Society, New York, 1981.

[Bok79] S. H. Bokhari. Dual processor scheduling with dynamic reassignment. *IEEE Transactions on Software Engineering*, 5, 1979.

[CZ83] D. R. Cheriton and W. Zwaenepoel. Distributed v kernal and its performance for diskless workstations. *Proceed. of the Ninth ACM Sysp. of Oper. Sys. Priniciples*, 5, 1983.

[Efe82] K. Efe. Heuristic models of task assignment scheduling in distributed systems. *IEEE Computers*, 15, 1982.

[ELZ86a] D. L. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12, 1986.

[ELZ86b] D. L. Eager, E. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load balancing. *Performance Evaluation*, 6, 1986.

[Kob76] H. Kobayshi. *Modeling and Analysis: An Introduction to System Performance Evaluation Methodogy.* Addison Wesley, Reading, Massachusetts, 1976.

[LM82] M. Livny and M. Melmen. Load balancing in homogeneous broadcast distributed systems. *Proceeding of the Computer Network Performance Symposium*, 1982.

[Lo84] V. Lo. *Heuristic Algorithms for Task Assignments in Distributed Systems.* IEEE Computer Society, New York, 1984.

[LO86] Leland and Ott. Load balancing heuristics abd process behavior. *ACM Performance Evaluation Review*, 14, 1986.

[MLT82] P. Ma, E. Lee, and M. Tsuchiya. A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, 31, 1982.

[MTJ89] R. Mirchandaney, D. Towsley, and J.Stankovic. Analysis of the effects of delays on load sharing. *IEEE Transactions on Computers*, 38, 1989.

[NT84] R. Nelson and D. Towsley. Approximating the mean delay of a multiple server queue using threshold scheduling. *IBM Report RC10912*, 1984.

[NTT87] R. Nelson, D. Towsley, and A. Tantawi. Performance analysis of parallel processing systems. *IEEE Transactions on Software Engineering*, 14, 1987.

[Rom89a] C. Gary Rommel. Probability of load balancing success in distributed operating systems. *ISMM International Conference, Ft. Lauderdale*, 1989.

[Rom89b] C. Gary Rommel. *Scheduling Parallel Programs in Distributed Systems.* PhD thesis, Massachusetts University, Amherst, Massachusetts, 1989.

[RS84] K. Ramamritham and J. Stankovic. Dynamic task scheduling in hard real-time distributed systems. *IEEE Computers*, 1, 1984.

[RSH79] G. Rao, H. Stone, and T. Hu. Assignments of tasks in a distributed processor system with limited memory. *IEEE Transactions on Computers*, 28, 1979.

[SS84] J. Stankovic and I. Sidhu. An adaptive bidding algorithm for processes, clusters, and distributed groups. *Proceedings of the Fourth International Conference on Distributed Computing*, 1984.

[Sta84] J. Stankovic. Simulation of three adaptive, decentralized controlled job scheduling algorithms. *Computer Networks*, 8, 1984.

[Sta85] J. Stankovic. An application of bayesian decision theory to decentral control of scheduling. *IEEE Transactions on Computers*, 43, 1985.

[Sto77] H. Stone. Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, 3, 1977.

[Sto78] H. Stone. Critical load factors in two-processor distributed systems. *IEEE Transactions on Software Engineering*, 4, 1978.

[TLC85] M. Theimer, K. Lantz, and D. Cheriton. Preemptable remote execution facilities for the v-system. *Proceed. of the Tenth ACM Symp. on Oper. Sys. Principles*, 19, 1985.