# Co-Scheduling Compute-Intensive Tasks on a Network of Workstations: Model and Algorithms

*Mikhail J. Atallah*[*], *Christina Lock*, and *Dan C. Marinescu*[†]
Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA

*Howard Jay Siegel*[‡]
Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN, 47907, USA

*Thomas L. Casavant*[§]
Parallel Processing Laboratory
Department of Electrical and Computer Engineering
University of Iowa
Iowa City, IA, 52242, USA

## Abstract

The problem of using the idle cycles of a number of high performance workstations, interconnected by a high speed network, for solving computationally intensive tasks is discussed. The classes of distributed applications examined require some form of synchronization among the sub-tasks, hence the need for co-scheduling to guarantee that sub-tasks start at the same time and execute at the same pace on a group of workstations. A model of the system is presented that allows the definition of an objective function to be maximized. Then a quadratic time and linear space algorithm is derived for computing the optimal co-scheduling, given the model and the class of problems addressed.

[*]Research supported by the Office of Naval Research under contracts N00014-84-K-0502 and N00014-86-K-0689, the Air Force Office of Scientific Research under grant AFOSR-90-0107, the National Science Foundation under grant DCR-8451393, and the National Library of Medicine under Grant R01-LM05118

[†]Research supported in part by the Strategic Defense Initiation under ARO grants DAAG03-86-K-0106, DAAL03-90-0107, by NATO under grant 891107, and by the NSF Software Engineering Research Center (SERC).

[‡] Research supported by the Naval Ocean Systems Center under the High Performance Computing Block, ONT, and by the Office of Naval Research under grant N00014-90-J-1483.

[§]Research supported by the National Science Foundation under grants CCR-8809600, ECS-8800910 and by the NSF Software Engineering Research Center (SERC).

**Index terms:** workstations, scheduling, idle cycles, resource allocation, load balancing.

## 1 Introduction

The cost/performance ratio of workstations has shown a dramatic improvement over the past few years. This trend will probably continue in the near future and it is expected that large capacity memory chips (64-256 Mbits) and more advanced RISC processors capable of delivering hundreds of MIPS and/or MFLOPS will be available at a low price.

The peak performance of such workstations is needed for computationally intensive tasks, but the computing power offered by a high performance workstation of the future will considerably exceed the sustained needs for personal computing of an average user. A large fraction of the machine cycles will generally be unused by local tasks and many cycles will be available for other uses. Because high speed networks (with speed in the 80-100 Mbits/sec range) and high performance network interfaces are already emerging, efforts to use efficiently this excess computing capacity are currently being undertaken and commercial products are emerging [12]. Clearly, sharing of these resources poses challenging problems in a variety of areas, such as computer security, network management, and resource management in a distributed environment.

Several papers have presented and analyzed various distributed computing systems and have addressed different schemes for scheduling distributed resources [6], [9]. Some of the systems proposed in the literature require the users to initiate the allocation of remote resources; others embed mechanisms to determine where available resources are located [7]. This paper focuses on a particular problem of resource management called "co-scheduling" or "gang scheduling" [1]. This involves dividing a large task into subtasks that are then scheduled to execute concurrently on a set of workstations [8]. The sub-tasks need to coordinate their execution, to start at essentially the same time, and compute at the same pace. Though there may be other classes of applications that require co-scheduling, the present discussion is confined to a particular class of applications, namely solving large numerical problems using iterative methods that require some form of synchronization among the subtasks [3].

Various parameters affect the efficiency of the co-scheduling and the resulting load on the system. These parameters include the number of workstations used for the task, the percentage of free cycles of the workstations in the system (which varies from one workstation to another), and the possible start-up time of the task.

It is assumed that the high performance workstations are interconnected by a high speed network and share one or more file servers. The goal of this work is to develop a strategy to allow utilization of the idle cycles of a set of workstations to solve the type of computationally intensive tasks mentioned above.

The contributions of this paper are a model of the system that allows a definition of an objective function to be maximized, and algorithms for optimal co-scheduling. The paper is organized in the following manner. The problem formulation and the model of the system are described in Section 2. The algorithm for optimal co-scheduling assuming equal load distribution is introduced and analyzed in Section 3. Section 4 extends the results of Section 3 for unequal load distribution.

## 2 Problem Formulation

In this section, the parameters to be considered in Section 3 for the co-scheduling of a large task on a set of workstations are discussed and quantified. The problem can be formulated as follows: the user submits a computationally intensive task, there are $Q$ workstations in the system, and the system has to choose which subset of these workstations to assign to the user's task. For example, assume that the user needs to solve a set of $N$ linear equations with $N$ unknowns. The user may solve the system of linear equations using one workstation and the time to solve the problem will be denoted by $T(1)$. Assume that $T(1) = 10$ hours. If $P$ workstations are available, a parallel algorithm would allow each workstation to work on a data sub-domain of size $\left(\frac{N}{\sqrt{P}} \times \frac{N}{\sqrt{P}}\right)$ and then the workstations working on neighboring subdomains will exchange boundary values at each iteration [3].

Let $T(P)$ be the parallel execution time (the time required to solve the problem) after the task has been distributed to $P$ workstations, ***assuming each of these workstations is entirely dedicated to solving this task*** (i.e., it has no other local jobs of its own, a rather unlikely situation). The ratio $S(P) = \frac{T(1)}{T(P)}$ will be called the ***speedup*** [11]. Assuming a linear speedup with $T(P) = \frac{T(1)}{P}$, and a number of workstations $P = 100$, the parallel execution time for the previous example would be $T(100) = 6$ minutes. However, the use of $P$ workstations may lead to a better than linear speedup, $T(P) < \frac{T(1)}{P}$, due to the fact that for large problems one workstation may not have enough memory to hold the entire data domain of size $N \times N$ in main memory, and the intense paging activity that may be contributing to a large $T(1)$ would be avoided by using $P$ workstations. For some problems, less than linear speedup may result due to the overhead for communication and control of the parallel execution.

The class of applications considered here exhibits coarse grain parallelism. The communication delays are substantial even in a high speed network, and computations can be distributed to the set of workstations in an effective way only if the ratio of the computation time to communication delays is sufficiently high. It is further assumed that the expected speedup for this class of problems is a monotonically increasing function of the number of workstations assigned to the application (at least within some bounds $P_{low} \leq P \leq P_{high}$).

Often the solution of a problem in the class discussed here requires some form of synchronization. In the example above, all workstations need to complete one iteration and then exchange boundary values to guarantee the convergence of the solutions. It follows that the mechanism for resource management should allow the selection of a subset $G$ of the set of all workstations $\{W_1, \ldots, W_Q\}$, where $|G| = P \leq Q$, such that the following two conditions hold.

(a) Each workstation $W_i$ in the set $G$ has a "duty cycle" $\eta_i$, which is defined as the ratio of cycles the workstation commits to local tasks to the number of cycles available for the compute-intensive task. The duty cycle is a non-negative real number. Local tasks are non-CPU-intensive activities that are generated by a local user, e.g., text editing, mail processing. These local tasks are unrelated to the solution of the CPU-intensive numerical problems. Recall all workstations receive identically sized sub-tasks (sub-domains) of the compute-intensive remote task (the "unequal" load case is considered in Section 4). If a workstation $W_i$ has a duty cycle $\eta_i$, then a remote sub-task that would complete in $T_r$ units of time when it uses all of $W_i$'s cycles, i.e. when $\eta_i = 0$, would require $T_r' = (1 + \eta_i)T_r$ units of time when $\eta_i > 0$. For example, if $\eta_i = 0.1$ then $T_r' = 1.1T_r$. Thus within $G$, a workstation with a lower $\eta_i$ value will complete an iteration faster than a workstation with a higher $\eta_i$ value, because the cycles needed for one iteration are the same for all workstations in $G$. Because a workstation of $G$ that finishes its iteration early has to wait for the others before communicating with them and then proceeding to the next iteration, the largest $\eta_i$ is the effective bottleneck for the group $G$, which is denoted by $\eta(G)$ (i.e., $\eta(G) = \max_{W_i \in G} \eta_i$). The time taken by the sub-task is then $(1 + \max_{W_i \in G} \eta_i)T_r$. The effect of allowing the workstations to have different $\eta_i$ values is the same as assuming that some of them are faster than others.

(b) All $P$ workstations should be capable of *starting the parallel computation at the same time.* Call $T_S(G)$ the time that elapses from the moment the request to solve the task is made to the moment all the workstations of $G$ can start processing it. Let $T_{S,i}$ be the time that elapses between the moment the request to solve the task is made and the moment workstation $W_i$ can begin solving it. Then $T_S(G) = \max_{W_i \in G} T_{S,i}$. It is assumed that a workstation processes at most one compute-intensive sub-task at a time. If the workstation currently has no compute-intensive sub-task, then $T_{S,i} = 0$, otherwise $T_{S,i}$ is the remaining computation time of that workstation's current sub-task (i.e., the startup time of the next possible compute-intensive sub-task).

With these conditions the *effective speedup*, $S_e(G)$, as seen by the user, is

$$S_e(G) = \frac{T(1)}{T_S(G) + (1 + \eta(G))T(|G|)}$$

where $T(|G|)$ is the execution time with $P = |G|$ workstations if all workstations were idle prior to the request, i.e., $\eta(G) = 0$, and immediately available, i.e., $T_S(G) = 0$. Given a high speed network interconnecting $Q$ workstations $\{W_1, \ldots, W_Q\}$, the task of locating $G$ workstations that maximize $S_e(G)$ subject to conditions (a) and (b) will be called *co-scheduling* or *gang-scheduling*.

The actual architecture and the organization of the software necessary to support the distributed application described above are beyond the scope of this paper. The focus of this paper is a high level model of the system that reveals the main agents involved and the flow of information among them to allow optimal decision making. The following classes of agents can be identified: *application managers*, agents that coordinate the execution of an application, *decision making agents*, involved in establishing resource allocation policies, and *scheduling agents*, that enforce resource allocation policies. An application manager requests resources on behalf of an application from a decision making agent that, in cooperation with other decision making agents, locates available resources. A decision making agent, called in the following a *broker*, requests *bids* from other brokers and then selects a subset of workstations that maximizes an objective function. For the particular application discussed in this paper, a bid consists of the pair start-up time and duty cycle, and the objective function is the effective speedup. As soon as a decision is made, all agents involved share their knowledge with local schedulers. It is assumed that all schedulers perform some form of multi-queue scheduling to support at least two classes of tasks, e.g., non CPU-intensive local tasks and compute-intensive remote tasks. A scheduler accepts from a local broker requests to allocate to the class of compute-intensive tasks a certain fraction of the CPU cycles available, and maintains statistical data concerning the actual allocation of cycles among different classes of tasks executing on that particular workstation.

As stated above, a parallel application is coordinated by an *application manager.* Consider the functions that the application manager must perform.

(a) It requests from the system a subset, $G$, of the $Q$ workstations, where $|G|$ =P and $P$ is in the range $P_{low} \leq P \leq P_{high}$. The values of $P_{low}$ and $P_{high}$ are computed from the following considerations. To compute $P_{low}$ the deadline by which the results are needed, $T_{deadline}$, as well as a start up time $T_S(G) = 0$, are used. Let the constant $\eta_{avg}$ be the assumed average value of the the duty cycle throughout the system. The value of $P_{low}$ is the minimum number of processors for which the equation below still holds.

$$T_{deadline} \leq (1 + \eta_{avg}))T(P_{low})$$

The value of $P_{high}$ is determined by the following equation:

$$P_{high} = \min\{Q, R\}$$

where $Q$ is the number of workstations in the system and $R$ is the maximum number of workstations that can be included while still maintaining a monotonically increasing speedup. Using the information supplied by the application manager, a decision making agent determines $G_{opt}$, the optimal subset of workstations that leads to the largest speedup, as well as the common start-up time $T_S(G_{opt})$.

(b) The application manager decomposes the data domain into $P_{opt} = | G_{opt} |$ sub-domains and then maps the data sub-domains to the $P_{opt}$ available workstations. Then it distributes the executable code to each workstation. It is assumed that the load assigned to each workstation is the same. Indeed, existing mathematical software packages attempt to distribute the computational load uniformly, by partitioning the data domain into equal sub-domains [3]. Of course, for some classes of numerical applications a partition of the data domain into sub-domains of unequal size is entirely feasible. This approach could lead to a higher speedup in a heterogeneous system or when the resources available, the CPU cycles in particular, differ from one workstation to another and it is considered in Section 4. For simplicity in Section 3, it is assumed that all workstations use the same family of processors, sub-domains are of equal size, and the executable code is the same. It is also assumed that all workstations have access to the file server containing the data.

(c) Lastly, the application manager gathers the final sub-task results from the $P_{opt}$ workstations and presents them to the end user.

The problem as formulated raises a number of subtle issues. The first issue is how to select the set $G$ of workstations that will ensure the highest effective speedup and how to reach consensus among a group of $P = |G|$ workstations upon a start-up time $T_S(G)$. Clearly, the larger $P$, the size of the group requested, the larger the start-up time may be, but the shorter will be the actual parallel execution time $T(P)$. If the execution time for a problem is data-independent, then algorithm analysis techniques can be used to derive $T(P)$. However, in general, estimation of $T(P)$ for a given $P$ is a non-trivial problem in itself. If the execution time is data dependent, as it is in most cases, statistical data from previous executions or information supplied by the user as part of the problem description is necessary to properly estimate $T(P)$. If the serial execution time, $T(1)$, is known then the parallel execution time can be approximated by $T(P) = \frac{T(1)}{P}$ if the overhead due to communication and control can be neglected.

Assume that $T(1) = 10$ hours, $T(P) = \frac{T(1)}{P}$, there are two groups $G'$ and $G''$, $\eta(G') = \eta(G'') = 0$, $T_S(G') = 5$ minutes, $T_S(G'') = 14$ minutes, $|G'| = 60$, and $|G''| = 100$. Then

$$S_e(G') = \frac{600}{5+10} = 40$$
$$S_e(G'') = \frac{600}{14+6} = 30$$

Hence it is better to use a group of 60 workstations capable of starting earlier, than to wait for 100 workstations with a later start-up time. If the duty cycle of the first group is, say, $\eta(G') = 0.2$ while the duty cycle of the second group is higher, say $\eta(G'') = 0.7$, then choosing the first group is even more beneficial, because

$$S_e(G') = \frac{600}{5 + 10 \times 1.2} = \frac{600}{17.0} = 35.3$$
$$S_e(G'') = \frac{600}{14 + 6 \times 1.7} = \frac{600}{24.2} = 24.8$$

Consider the case where the duty cycle of the first group is, say, $\eta(G') = 0.6$, while the duty cycle of the second group is much lower, say $\eta(G'') = 0.1$. Such a circumstance could occur if $G''$ is disjoint from $G'$, and $G''$ is a set of workstations busy with another sub-task until $T_S(G'')$. Then the situation is reversed from the one above, because

$$S_e(G') = \frac{600}{5 + 10 \times 1.6} = \frac{600}{21.0} = 20.5$$
$$S_e(G'') = \frac{600}{14 + 6 \times 1.1} = \frac{600}{20.6} = 20.9$$

Another issue is how to ensure "fairness," and to avoid "starvation", i.e., how to guarantee that a request will eventually be granted. A related issue is *processor fragmentation*. Processor fragmentation will occur when all the processor groups that can be located are of a smaller size than the size of the groups needed to solve current problems. A more general issue is how local concerns, e.g., the desire to obtain optimal effective speedups for individual parallel applications, could be reconciled with the goal of minimizing the number of idle cycles of all workstations. Equally difficult are the issues related to error recovery. When a processor allocated to a problem fails, the other members of the group must be able to complete the parallel computation with a minimal amount of cycles lost. These are challenging issues but beyond the scope of this paper, which is focused on finding co-scheduling algorithms that ensure maximal speedups.

## 3 Algorithms for Optimal Co-Scheduling

### 3.1 Basic Assumptions

The goal of a co-scheduling algorithm is to determine $G$, the group of workstations assigned to a parallel computation, the duty cycle $\eta(G)$ for the group, and the start-up time $T_S(G)$ for the group, such that $P = |G|$ is in the range $P_{low} \leq P \leq P_{high}$ and $S_e(G)$ is maximized. A co-schedule is *optimal* if it maximizes the effective speedup, $S_e(G)$.

The following assumptions are made

1. There are $Q$ workstations, $W_1, \ldots, W_Q$.
2. Each workstation $W_i$ can supply to a decision making agent the tuple $(\eta_i, T_{S,i})$ containing its duty cycle and earliest start-up time.
3. The load assigned to each workstation is the same. This assumption is not a fundamental limitation of this method, but it reflects the fact that the numerical problems considered are typically harder to partition into unequal pieces than into equal ones. But this assumption is not essential to the analysis presented, and in fact this co-scheduling scheme works for other load-sharing methods as well (this is discussed later, in Section 4).
4. The decision making agent receives from the application manager the following information: $T(1)$, the serial execution time of the application, $T_{deadline}$, the deadline for obtaining the results, and an estimate of the overhead for inter-workstation communication and control as a function of $P$. Based on this data the decision making agent can compute an estimate of $T(P)$, the parallel execution time with $P$ workstations, for any $P$, and can also estimate the values of $P_{high}$ and $P_{low}$. If $T_{cc}(P)$ is the time for communication and control for a parallel execution with $P$ workstations, and if $T_l(P)$ is the time to send and load the code and data, as well as gather the final sub-task results and present them to the user, then the parallel execution time with $P$ workstations whose duty cycles and startup times are all zero, is
$$T(P) = \frac{T(1)}{P} + T_{cc}(P) + T_l(P).$$
5. The parallel execution time is a monotonically decreasing function of $P$, $T(P+k) < T(P)$ for $P_{low} \leq P < P_{high}$ and $0 < k \leq P_{high} - P$.

Observe that if all $\eta_i$ values were equal to one another, then for a given fixed value of $|G|$, the optimal G consists of the $|G|$ workstations having the $|G|$ smallest $T_{S,i}$ values, which easily implies an $O(Q \log Q)$ time algorithm (e.g., sort the workstations by their $T_{S,i}$ values and then for each possible value of $|G|$ check in constant time its corresponding effective speedup). A similar algorithm could be used if all $T_{S,i}$ values were equal to one another and the $\eta_i$ values were not. Then the sorting would be done by the $\eta_i$ values.

### 3.2 A General Algorithm for Optimal Co-Scheduling

Consider now the general case when the duty cycle and start-up times of any pair of workstations may be different. In this case, the effective speedup attainable with a group $G$ of $P = |G|$ workstations is
$$S_e(G) = \frac{T(1)}{\max_{W_i \in G} T_{S,i} + (1 + \max_{W_i \in G} \eta_i) T(P)}.$$
An algorithm leading to an optimal co-scheduling for the general case follows.

Let $A$ be the set of subsets (i.e., the power set) of $\{W_1, \ldots, W_Q\}$. An $O(Q^2)$ time and $O(Q)$ space algorithm for computing the quantity $\min_{B \in A} g(B)$ is given, where
$$g(B) = \max_{W_i \in B} T_{S,i} + (1 + \max_{W_i \in B} \eta_i) T(|B|).$$
This algorithm also determines the set $B \in A$ (call it $\hat{B}$) for which $g(B)$ is minimized. Without loss of generality, it is assumed that $P_{low} = 1$, $P_{high} = Q$, and that $i \neq j$ implies $T_{S,i} \neq T_{S,j}$ and $\eta_i \neq \eta_j$ (the algorithm can easily be modified for the general case).

Let $\pi$ be a permutation of $\{1, \ldots, Q\}$ such that $\eta_{\pi(1)} < \eta_{\pi(2)} < \cdots < \eta_{\pi(Q)}$. Of course $\pi$ can be obtained in $O(Q \log Q)$ time, and henceforth it is assumed that it is available.

**Definition 1** *Let $A_k$ denote the subset of $A$ such that $B \in A_k$ if and only if $\max_{W_i \in B} T_{S,i} = T_{S,k}$, for a fixed $k$, $1 \leq k \leq Q$. Let $Best_k$ denote $\min_{B \in A_k} g(B)$, and let $\hat{B}_k$ be the $B$ at which this minimum is achieved.*

Now, observe that $\min_{B \in A} g(B) = \min_k Best_k$, because $A = \cup_k A_k$. For the same reason, if $k'$ is the index for which $\min_{B \in A} g(B) = Best_{k'}$, then $\hat{B} = \hat{B}_{k'}$. Therefore, to show that $\hat{B}$ and $g(\hat{B})$ can be computed in $O(Q^2)$ time and $O(Q)$ space, it suffices to give an $O(Q)$ time and space algorithm for computing $Best_k$ and $\hat{B}_k$ for a particular value of $k$. This is what it is done next (so in what follows $k$ is fixed).

**Definition 2** *Let $A_{k,P}$ denote the set of elements of $A_k$ that have cardinality $P$ for a fixed $P$, $1 \leq P \leq Q$. That is, $B \in A_{k,P}$ if and only if: (i) $\max_{W_i \in B} T_{S,i} = T_{S,k}$, and (ii) $|B| = P$. Let $Best_{k,P}$ denote $\min_{B \in A_{k,P}} g(B)$, and let $\hat{B}_{k,P}$ be the $B$ at which this minimum is achieved.*

Now, observe that $Best_k = \min_P Best_{k,P}$, because $A_k = \cup_P A_{k,P}$. For the same reason, if $P'$ is the index for which $Best_k = Best_{k,P'}$, then $\hat{B}_k = \hat{B}_{k,P'}$. Therefore it suffices to compute, in $O(Q)$ time and space, $Best_{k,P}$ and $\hat{B}_{k,P}$ for *all* indices $P \in \{1, 2, \ldots, Q\}$. The description of each such $\hat{B}_{k,P}$ that is computed must be *implicit* and must take $O(1)$ space, because the elements of each $\hat{B}_{k,P}$ cannot be listed explicitly (otherwise the algorithm would use quadratic space because $\sum_P |\hat{B}_{k,P}|$ is proportional to $Q^2$, and if the space used is quadratic then clearly linear time is impossible).

The computation is based on the following lemma.

**Lemma 1** *Let $L_k$ be the sorted list containing the set $\{\eta_i : T_{S,i} < T_{S,k}, 1 \leq i \leq Q\}$, and assume that $|L_k| \geq P - 1$. Let the first (i.e., smallest) $P - 1$ elements of $L_k$ be $\eta_{i_1}, \eta_{i_2}, \ldots, \eta_{i_{P-1}}$ (listed in increasing order). Then the set $\hat{B}_{k,P} = \{W_{i_1}, W_{i_2}, \ldots, W_{i_{P-1}}, W_k\}$.*

**Proof.** Let $B \in A_{k,P}$. Then (by definition) $B$ contains $W_k$ and is such that $\max_{W_i \in B} T_{S,i} = T_{S,k}$. In addition to $W_k$, $B$ contains $P - 1$ other workstations whose $\eta_i$s appear in $L_k$; among these $P - 1$ $\eta_i$ values, the largest cannot be smaller than $\eta_{i_{P-1}}$. Therefore

$$g(B) \geq T_{S,k} + (1 + \max\{\eta_k, \eta_{i_{P-1}}\})T(P)$$

=

$$g(\{W_{i_1}, W_{i_2}, \ldots, W_{i_{P-1}}, W_k\}),$$

which completes the proof. □

The above lemma implies an algorithm for computing, in $O(Q)$ time and space, $Best_{k,P}$ and (an implicit description of) $\hat{B}_{k,P}$ for all indices $P \in \{1, 2, \ldots, Q\}$. To see that this is so, first observe that the sorted list $L_k$ can easily be obtained in $O(Q)$ time from the permutation $\pi$. Each element of the sorted sequence $\eta_{\pi(1)}, \ldots, \eta_{\pi(Q)}$ is considered in turn, and when considering (say) $\eta_{\pi(i)}$, simply test whether $T_{S,\pi(i)}$ is smaller than $T_{S,k}$. If the answer to the test is "yes," then $\eta_{\pi(i)}$ is included in $L_k$. The lemma implies that $L_k$ itself is an implicit description of $\hat{B}_{k,P}$ for all indices $P \in \{1, 2, \ldots, Q\}$, because $\hat{B}_{k,P}$ is described by the first $P - 1$ elements of $L_k$ (together with $W_k$, which by definition is always part of $\hat{B}_{k,P}$). Each value $Best_{k,P}$ is easily obtained in constant time from $L_k$. Let $\eta_{i_{P-1}}$ denote the $(P-1)$st smallest value in $L_k$ (as in the lemma), and then

$$Best_{k,P} = T_{S,k} + (1 + \max\{\eta_k, \eta_{i_{P-1}}\})T(P).$$

If $|L_k| < P - 1$ then of course $A_{k,P} = \{\}$ and hence $\hat{B}_{k,P} = \{\}$ and $Best_{k,P}$ is taken to be arbitrarily bad (i.e., equal to $\infty$).

## 3.3 Example

Consider the following example as an illustration of the algorithm. In this example, a network with only five workstations is considered.

The five workstations are characterized by the following values of $(T_{S,i}, \eta_i)$:

$$W_1 : (T_S = 6, \eta = .6)$$
$$W_2 : (T_S = 7, \eta = .5)$$
$$W_3 : (T_S = 4, \eta = .7)$$
$$W_4 : (T_S = 12, \eta = .3)$$
$$W_5 : (T_S = 0, \eta = .1)$$

The workstation values are first sorted on $\eta_i$ to form $\pi$. In this example, $\pi = 5, 4, 2, 1, 3$.

For every value of k, $1 \leq k \leq Q$, the following steps are performed. The list $L_k$ is formed by considering each element $i$ in $\pi$ in order and including the element in $L_k$ if $T_{S,i} < T_{S,k}$. For the purposes of the example, let k = 2. Then $L_2 = (0.1, 0.6, 0.7)$. $L_k$ implicitly represents the best subset of $A_{k,P}$ for any $P$. These subsets $\hat{B}_{k,P}$ are listed here for clarity.

$$\hat{B}_{2,1} = \{W_2\}.$$
$$\hat{B}_{2,2} = \{W_5, W_2\}.$$
$$\hat{B}_{2,3} = \{W_5, W_1, W_2\}.$$

$$\hat{B}_{2,4} = \{W_5, W_1, W_3, W_2\}.$$

$$\hat{B}_{2,5} = \{\}.$$

Subsets such as $\{W_3, W_2\}$ are not considered because it is known that the max $\eta$ of this subset is greater than the max $\eta$ of $\hat{B}_{2,2} = \{W_5, W_2\}$.

The values of $Best_{k,P}$ are now compared to find $Best_k$, where

$$Best_{k,P} = T_{S,k} + (1 + \max\{\eta_k, \eta_{i_{P-1}}\})T(P).$$

Recall that the value of $\eta_{iP-1}$ is found in constant time by indexing to the $(P-1)$st element of $L_k$. These values are as follows for the example, in which linear speedup is assumed in order to approximate $T(P)$. This approximation is not part of the algorithm; it is used here to simplify the example.

$$Best_{2,1} = T_{S,2} + (1 + .5)T(1) = T_{S,2} + (1.5)T(1).$$

$$Best_{2,2} = T_{S,2} + (1 + .5)T(2) = T_{S,2} + (1.5)\frac{T(1)}{2} = T_{S,2} + (.75)T(1).$$

$$Best_{2,3} = T_{S,2} + (1 + .6)T(3) = T_{S,2} + (1.6)\frac{T(1)}{3} = T_{S,2} + (.53)T(1).$$

$$Best_{2,4} = T_{S,2} + (1 + .7)T(4) = T_{S,2} + (1.7)\frac{T(1)}{4} = T_{S,2} + (.425)T(1).$$

$$Best_{2,5} = +\infty.$$

The minimum value is that for $\hat{B}_{2,4}$. Because $Best_k = \min_P Best_{k,P}$, in this case

$$Best_2 = Best_{2,4} = 7 + (.425)T(1).$$

These steps would be repeated for all other values of k while keeping the minimum $Best_k$ value found so far and its corresponding set $\hat{B}_k$.

## 4 Extension to Unequal Load Distribution

The above analysis assumed equal distribution of computational load among the chosen workstations. When the load can be partitioned unequally among the chosen workstations, it is obviously better to send more work to the faster workstations (those with a low $\eta_i$) in such a way that all the workstations terminate at the same time (so that none of them has to wait for the others to terminate). The method known as "scattered decomposition" [3], can be used to allocate unequal load to the set of workstations.

An analysis of the effective speedup function within this framework of unequal loads is presented below. Although this function will differ substantially from that for the equal load case, it will turn out that essentially the same algorithm as for the equal load case can solve the problem in that case as well.

First note that when the loads are unequal, the communication time becomes a function of the set of chosen workstations and of the load distribution, rather than a function of the number of chosen workstations. However, for compute-intensive tasks, the communication time can either be neglected or approximated by assuming that it depends only on the number of chosen workstations (in which case we could use the same $T_{cc}(P) + T_l(P)$ term as in the case of equal load distribution, keeping in mind that it is small compared to computation time). In other words, if $G$ is the set of chosen workstations, with $P = |G|$, then the total amount of work performed by $G$ is approximately the same as in the previous case of equal load distribution, namely $P \cdot T(P)$, where $T(P)$ is as defined in the previous section. Let $T_i(G)$ be the time needed for workstation $W_i \in G$ to complete its sub-task if $\eta_i = 0$. Then, because the same total amount of work must be done

$$\sum_{W_i \in G} T_i(G) = P \cdot T(P).$$

The goal is to have all $W_i \in G$ complete their subtasks simultaneously. Therefore, it is required that

$$(1 + \eta_i)T_i(G) = K,$$

for all $W_i \in G$, and the constant $K$, the common execution time. Thus,

$$T_i(G) = K(1 + \eta_i)^{-1}.$$

Substituting into the above summation gives

$$K = P \cdot T(P)(\sum_{W_i \in G} (1 + \eta_i)^{-1})^{-1}.$$

Hence,

$$T_i(G) = P \cdot T(P)(1 + \eta_i)^{-1}(\sum_{W_i \in G} (1 + \eta_i)^{-1})^{-1}$$

In this case the effective speedup function $S_e(G)$ differs from the equal-load case in that its denominator no longer contains the additive term $(1 + \max_{W_i \in G} \eta_i)T(P)$, as that term would instead be replaced by $P \cdot T(P)(\sum_{W_i \in G} (1 + \eta_i)^{-1})^{-1}$. Thus the effective speedup is now:

$$S_e(G) = \frac{T(1)}{\max_{W_i \in G} T_{S,i} + P \cdot T(P)(\sum_{W_i \in G} (1 + \eta_i)^{-1})^{-1}}.$$

It is not hard to see that the algorithm sketched in the previous section still works in this case as well, because for a given $A_k$ and a given $P$, it is best to choose the $P-1$ elements of $L_k$ having smallest $\eta_i$ values. The time is still quadratic if, in addition to each $L_k$, the array $L'_k$ whose $j$th entry $(1 \leq j \leq |L_k|)$ is the sum

$$\sum_{i=1}^{j}(1+L_k(i))^{-1}$$

is also computed (of course $L'_k$ is computed from $L_k$ in linear time).

## 5 Examples of Other Uses of Co-Scheduling

The model and algorithm presented here can be applied to other situations by adjusting the interpretation of $\eta_i$. Two examples are considered in this section.

The first example involves scheduling resources in a reconfigurable large-scale parallel processing system, with homogeneous processors, when faults can occur. A high-level overall model for automatically and dynamically allocating resources among concurrent sub-tasks to minimize the total task execution time is presented in [2]. In that approach, fixed-size groups of fault-free processor/memory pairs (call them PE-groups) are the resources scheduled (e.g., PASM [10]). Sets of PE groups are dynamically assigned to sub-tasks (see [2] for details). In terms of the co-scheduling model, each PE-group $i$ has its own $T_{S,i}$, and $\eta_i$ is always zero when a PE-group is available to be scheduled (i.e., there are no background jobs and only one sub-task uses a PE-group at a time). A value of $\eta_i > 0$ can be used to present performance degradation of a PE-group due to faculty components in that PE-group, with the magnitude of $\eta_i$ proportional to the degradation. The co-scheduling algorithm can then be adapted for use in the automatic reconfiguration system.

As a second example, consider the application of co-scheduling to the problem of "distributed heterogeneous supercomputing" [4], [5]. In this situation, a suite of heterogeneous computing devices (e.g., a vector, a MIMD, and a SIMD machine) are available to jointly execute a task, where each machine is used for those code segments that execute most quickly on that type of architecture. Current work in this area has concentrated on the selection of a suite of machines to purchase given one or more classes of tasks [4], [5]. Given a heterogeneous suite of supercomputers and a sequence of tasks to execute, each with its own mix of code types, an algorithm for optimal processor assignment has not yet been addressed. The co-scheduling approach can be used in the development of a solution to this problem. It is desired to have all resources available concurrently, code segments from only one task will be executed on a processor at a time, a set of processors must be chosen based on available start times ($T_{S,i}$s) and processing capabilities ($\eta_i$s), and effective speedup is to be maximized. In this case, the $\eta_i$ for each machine is a $\tau$-tuple, where there are $\tau$ distinct code types and the value of the $\eta_i$ $\tau$-tuple reflects how effectively that processor can execute each of the code types. This information, in conjunction with the percentages of each code type in the task, can be used to choose the optimal set of machines to execute the task by extending the co-scheduling algorithm for workstations. Details are under development.

## 6 Summary

A distributed computing environment is discussed in which a set of high performance workstations are interconnected by a high speed network. It is assumed that the sustained needs for local non-intensive computing (e.g., text editing) are far below the peak performance of the computing engines. Methods of using the idle cycles of the workstations are investigated. The main focus is the study of co-scheduling, a form of resource management required by applications with sub-tasks that need to communicate and synchronize during execution. Co-scheduling implies that resources are allocated and deallocated in groups. The size of a group depends upon the needs of the application and upon the availability of resources. For the parallel applications discussed, the effective speedup provides an objective function to be maximized. A high level model of the system and an $O(Q^2)$ time and $O(Q)$ space algorithm for the optimal co-scheduling of $Q$ workstations are presented and analyzed.

## 7 References

1. D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer*, Vol. 23, No. 5, pp. 35-43, May 1990.

2. C. Chu, E. J. Delp, L. H. Jamieson, H. J. Siegel, F. J. Weil, and A. B. Whinston, "A Model for an Intelligent Operating System for Executing Image Understanding Tasks on a Reconfigurable

Parallel Architecture," *Journal of Parallel and Distributed Computing*, Vol. 6, No. 3, pp. 598-622, June 1989.

3. G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. W. Waker, *Solving Problems on Concurrent Processors*, Prentice Hall, 1988.

4. R. Freund, "Optimal Selection Theory for Superconcurrency," *Proc. Supercomputing '89*, pp. 699-703, November 1989.

5. R. Freund and D. S. Conwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, Vol. 3, No. 10, pp. 47-50, October 1990.

6. R. Hagmann, "Processor Server: Sharing Processing Power in a Workstation Environment," *Proc. 6th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 260-267, May 1986.

7. E. D. Lazowska, H. H. Levy, G. T. Ames, M. J. Fisher, R. J. Fowler, and S. C. Vestal, "The Architecture of the Eden System," *Proc. 8th ACM Symposium on Operating Systems Principles*, pp 148-159, December 1981.

8. D. C. Marinescu, J. R. Rice, B. Waltsburger, C. E. Houstis, T. Kunz, and H. Waldschmidt, "Distributed Supercomputing," *Proc. Workshop Future Trends in Distributed Computing*, IEEE Computer Society Press, pp. 381-387, October 1990.

9. M. W. Mutka and M. Litvny, "Scheduling Remote Processing Capacity in a Workstation-Processor Bank Network," *Proc. 7th International Conference on Distributed Computing Systems*, IEEE Computer Society Press, pp. 2-9, September 1987.

10. H. J. Siegel, T. Schwederski, J. T. Kuehn, and N. J. Davis IV, "An Overview of the PASM Parallel Processing System," in *Computer Architecture*, edited by D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, IEEE Computer Society Press, Washington, D.C., pp. 387-407, 1987.

11. L. J. Siegel, H. J. Siegel, and P. H. Swain, "Performance Measures for Evaluating Algorithms for SIMD Machines," *IEEE Transactions on Software Engineering* , Vol. SE-8, No. 4, pp. 319-331, July 1982.

12. D. Taylor, "The Promise of Cooperative Computing," *HP Design and Automation*, Vol 5, No. 11, pp. 25-36, November 1989.