JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 16, 319–327 (1992)

Models and Algorithms for Coscheduling Compute-Intensive Tasks on a Network of Workstations

MIKHAIL J. ATALLAH,* CHRISTINA LOCK BLACK, AND DAN C. MARINESCU†

Computer Sciences Department, Purdue University, West Lafayette, Indiana 47907

HOWARD JAY SIEGEL‡

Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907

AND

THOMAS L. CASAVANT§

Parallel Processing Laboratory, Department of Electrical and Computer Engineering, University of Iowa, Iowa City, Iowa 52242

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 subtasks, hence the need for coscheduling to guarantee that subtasks 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 coschedule, for the given model and class of applications addressed. © 1992 Academic Press, Inc.

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.

* 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 and DAAL03-90-0107, by NATO under Grant 891107, and by the NSF under Grant CCR-9119388.

‡ 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 Grant CCR-9024484 and by DARPA under Contract N00174-91-C-0116.

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 needs for the sustained personal computing intensity 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 speeds in the 80–100 Mbits/sec range) and high performance network interfaces are already emerging, efforts are being made to efficiently use this excess computing capacity and commercial products are now emerging [20]. Clearly, sharing CPU cycles poses challenging problems in a variety of areas, such as computer security, network management, and resource management in a distributed environment.

While distributed scheduling has been studied for quite some time [3], it is currently being undertaken at a more practical level. Several papers have presented and analyzed various distributed computing systems and have addressed different schemes for scheduling distributed resources. More recently, since the proliferation of low-cost, powerful workstation networks, *general purpose* scheduling of this idle capacity has been addressed. Hagmann [9] addressed this with a scheme for locating idle workstations, and Douglis and Ousterhout examined the details of migrating task contexts in such an environment [5]. Since then, there has been a steady progression in the effort either to speed task execution by locating idle resources [1, 14], or simply to achieve higher levels of hardware utilization via load balancing or load sharing [10–12, 21]. However, little attention has been given to the problem in which a great deal is known a priori about task characteristics—particularly groups of compute-intensive tasks which cooperate (yet with small in-

0743-7315/92 $5.00

terprocess communication requirements) to solve a single problem on a group of workstations. This paper focuses on a particular problem of resource management called "coscheduling" or "gang scheduling" [2, 15]. This involves dividing a large task into subtasks that are then scheduled to execute concurrently on a set of workstations [13]. The subtasks need to coordinate their execution, to start at essentially the same time, and to compute at the same pace. Though there may be other classes of applications that require coscheduling, 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 [6].

In [15] the concept of *coscheduling* is introduced in the context of scheduling of coherent task forces (i.e., groups of processes that need to be coscheduled) by the Medusa operating system [16] on the Cm* experimental system developed at CMU in the late 1970's [19]. There, a pause mechanism is proposed "to allow communicating processes to proceed at full speed by ensuring that processes are available for interaction when needed." In [15] it is assumed that process working sets are initially specified by the programmer and the algorithms assume that the scheduler has full knowledge about all task forces. The objective of their coscheduling algorithms is to "maximize the average number of processors executing coscheduled processes."

The present paper approaches the problem from a different perspective. The process working set of a task group is not given a priori but is determined as a result of a bidding process. Moreover, the broker acting on behalf of an application does not have any knowledge about other task forces except what it may infer from the bids received from the other brokers. The objective of the broker is to maximize the speedup for that particular application without any regard for global resource utilization in the system. On a more philosophical note, this point of view reflects our belief that a user's requirements (e.g., to have a task completed in the shortest possible time) will have precedence over a system's requirements (e.g., to minimize the processor idle time) in the case when resources (in this case CPU cycles) are abundant.

Various parameters affect the efficiency of a coschedule 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 coscheduling. 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 coscheduling assuming equal load distribution is introduced and analyzed in Section 3. Section 4 extends the results of Section 3 to unequal load distribution, and Section 5 provides examples of other uses of coscheduling.

2. PROBLEM FORMULATION

In this section, the parameters to be considered in Section 3 for the coscheduling 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 $T(1) = 10$ h. If P workstations are available, a parallel algorithm would allow each workstation to work on a data sub-domain of size $(N/\sqrt{P} \times N/\sqrt{P})$. The workstations assigned to neighboring subdomains will exchange boundary values at each iteration [6].

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 $S(P) = T(1)/T(P)$ will be called the *speedup* [18]. Assuming a linear speedup with $T(P) = T(1)/P$, and a number of workstations $P = 100$, the parallel execution time for the previous example would be $T(100) = 6$ min. However, the use of P workstations may lead to a better than linear speedup, $T(P) < T(1)/P$, 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 contribute 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. In general, these two effects will cancel one another, but good speedups may be expected.

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" η_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 nonnegative 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 that all workstations receive identically sized subtasks (subdomains) of the compute-intensive remote task (the "unequal" load case is considered in Section 4). If a workstation W_i has a duty cycle η_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 η_i value will complete an iteration faster than a workstation with a higher η_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 η_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 η_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 subtask at a time. If the workstation currently has no compute-intensive subtask, then $T_{S,i} = 0$; otherwise $T_{S,i}$ is the remaining computation time of that workstation's current subtask (i.e., the startup time of the next possible compute-intensive subtask). Because of the various practical solutions of the client-server problem in real workstation systems, the effect of migrating the executable image from disk-server to compute-client is ignored.

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 *coscheduling* 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 agents*, agents that coordinate the execution of an application; *decision making agents*, that are involved in establishing resource allocation policies; and *scheduling agents*, that enforce resource allocation policies. An application agent 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. Brokers receiving bids, and responding to them, must refrain from accepting other bids which would result in over-committing available resources. 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 schedules. It is assumed that all schedules perform some form of multiqueue 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 agent*. Consider the functions that the application agent 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 η_{avg} be the assumed average value of the duty cycle throughout the system. The value of P_{low} is the minimum number of processors for which the equation below still holds:

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

The value of P_{high} is determined by the 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 agent, 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 agent decomposes the data domain into $P_{opt} = |G_{opt}|$ subdomains and then maps the data subdomains to the P_{opt} available workstations. 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 subdomains [6]. Of course, for some classes of numerical applications a partition of the data domain into subdomains 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, subdomains 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 and code.

(c) Last, the application agent gathers the final subtask 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 subset G of workstations that will ensure the highest effective speedup and how to reach consensus among a group of $P = |G|$ workstations on 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 nontrivial problem in itself. If the execution time is data dependent, as it is in most cases, statistical data from previous executions or data supplied by the user as part of the problem description are necessary to properly estimate $T(P)$. If the serial execution time, $T(1)$, is known then the parallel execution time can be approximated by $T(P) = T(1)/P$ if the overhead due to communication and control can be neglected.

Assume, for example, that $T(1) = 10$ h, $T(P) = T(1)/P$, there are two groups G' and G'', $\eta(G') = \eta(G'') = 0$, $T_S(G'') = 5$ min $T_S(G'') = 14$ min, $|G'| = 60$, 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, however, 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 subtask 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 occurs 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 indi-

vidual 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 number of cycles lost. These are challenging issues, but beyond the scope of this paper, which is focused on finding coscheduling algorithms that ensure maximal speedups.

3. ALGORITHMS FOR OPTIMAL COSCHEDULING

3.1. Basic Assumptions

The goal of a coscheduling 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 $P_{low} \le P \le P_{high}$ and $S_e(G)$ is maximized. A coschedule 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 coscheduling 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 agent 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 interworkstation 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 to gather the final subtask 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

$$T(P) = \frac{T(1)}{P} + T_{cc}(P) + T_l(P).$$

To simplify the presentation it is assumed that $T_{cc}(P) = 0$ and $T_l(P) = 0$. This results in a reasonable approximation because the applications of interest are compute-intensive and different implementations of the executable image distribution from file server to compute agents are possible. Nevertheless, when communication and loading time can be estimated, the previous formula can incorporate this information.

5. The parallel execution time is a monotonically decreasing function of P, $T(P + k) < T(P)$ for $P_{low} \le P < P_{high}$ and $0 < k \le P_{high} - P$.

Observe that if all η_i values were equal to one another, then for a given fixed value of $|G|$, the optimal G would consist 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 η_i values were not. Then the sorting would be done by the η_i values.

3.2. A General Algorithm for Optimal Coscheduling

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

$$S_e(G) = \frac{T(1)}{T_S(G) + (1 + \max_{W_i \in G} \eta_i)T(P)}.$$

An algorithm leading to an optimal coschedule 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 $i \ne j$ implies $T_{S,i} \ne T_{S,j}$ and $\eta_i \ne \eta_j$. This is done to simplify the exposition; the algorithm can easily be modified for the general case.

Let π be a permutation of $\{1, \ldots, Q\}$ such that $\eta_{\pi(1)} < \eta_{\pi(2)} < \cdots < \eta_{\pi(Q)}$. Of course π 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, $1 \le k \le Q$. 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 $\min_{B \in A} g(B) = \min_k \text{Best}_k$, because $A = \bigcup_k A_k$. For the same reason, if k' is the index for which $\min_{B \in A} g(B) = \text{Best}_{k'}$, 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 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, $1 \le P \le Q$. 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 $\text{Best}_{k,P}$ denote $\min_{B \in A_{k,P}} g(B)$, and let $\hat{B}_{k,P}$ be the set of subsets B at which this minimum is achieved.

Now, observe that $\text{Best}_k = \min_P \text{Best}_{k,P}$, because $A_k = \bigcup_P A_{k,P}$. For the same reason, if P' is the index for which $\text{Best}_k = \text{Best}_{k,P'}$, then $\hat{B}_k = \hat{B}_{k,P'}$. Therefore it suffices to compute, in $O(Q)$ time and space, $\text{Best}_{k,P}$ 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 consisting of the elements of the set $\{\eta_i : T_{S,i} < T_{S,k},\ 1 \le i \le Q\}$, and assume that $|L_k| \ge P - 1$. 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 η_i's appear in L_k; among these $P - 1$ η_i values, the largest cannot be smaller than $\eta_{i_{P-1}}$. Therefore

$$g(B) \ge T_{S,k} + (1 + \max\{\eta_k, \eta_{i_{P-1}}\})T(P) = g(\{W_{i_1}, W_{i_2}, \ldots, W_{i_{P-1}}, W_k\}),$$

which completes the proof. ■

The above lemma implies an algorithm for computing, in $O(Q)$ time and space, $\text{Best}_{k,P}$ 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 π. Each element of the sorted sequence $\eta_{\pi(1)}, \ldots, \eta_{\pi(Q)}$ is considered in turn, and when considering, for example, $\eta_{\pi(i)}$, 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 $\text{Best}_{k,P}$ is easily obtained in constant time from L_k. Let $\eta_{i_{P-1}}$ denote the $(P - 1)$th smallest value in L_k (as in the lemma), and then

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

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

3.3. Example

Consider the following example as an illustration of the algorithm. In this example, a network with 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 η_i to form π. In this example, $\pi = 5, 4, 2, 1, 3$.

For every value of k, $1 \le k \le Q$, the following steps are performed. The list L_k is formed by considering each element i in π 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 η of this subset is greater than the max η of $\hat{B}_{2,2} = \{W_5, W_2\}$.

The values of $\text{Best}_{k,P}$ are now compared to find Best_k, where

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

Recall that the value of $\eta_{i_{P-1}}$ is found in constant time by indexing to the $(P - 1)$th 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 merely used here to simplify the example.

$$\text{Best}_{2,1} = T_{S,2} + (1 + .5)T(1) = T_{S,2} + (1.5)T(1).$$

$$\text{Best}_{2,2} = T_{S,2} + (1 + .5)T(2) = T_{S,2} + (1.5)\frac{T(1)}{2}$$

$$= T_{S,2} + (.75)T(1).$$

$$\text{Best}_{2,3} = T_{S,2} + (1 + .6)T(3) = T_{S,2} + (1.6)\frac{T(1)}{3}$$

$$= T_{S,2} + (.53)T(1).$$

$$\text{Best}_{2,4} = T_{S,2} + (1 + .7)T(4) = T_{S,2} + (1.7)\frac{T(1)}{4}$$

$$= T_{S,2} + (.425)T(1).$$

$$\text{Best}_{2,5} = +\infty.$$

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

$$\text{Best}_2 = \text{Best}_{2,4} = 7 + (.425)T(1).$$

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 η_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" [6] can be used to allocate an unequal load to the set of workstations.

An analysis of the effective speedup function with 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 it had* $\eta_i = 0$. Then, because the same total amount of work must be done, it follows that

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

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

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

for all $W_i \in G$, where K is the common execution time. Thus,

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

Substituting into the above summation gives

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

Hence,

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

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 η_i values. The time is still quadratic, if in addition to each L_k, the array L'_k whose jth entry $(1 \le j \le |L_k|)$ 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 COSCHEDULING

The model and algorithm presented here can be applied to other situations by adjusting the interpretation of η_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 [4]. In that approach, fixed-size groups of fault-free processor/memory pairs (call them PE-groups) are the resources scheduled (e.g., PASM [17]). Sets of PE groups are dynamically assigned to sub-tasks (see [4] for details). In terms of the coscheduling model, each PE-group i has its own $T_{S,i}$, and η_i is always zero when a PE-group is available to be scheduled (i.e., there are no background jobs and only one subtask 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 faulty components in that PE-group, with the magnitude of η_i proportional to the degradation. The coscheduling algorithm can then be adapted for use in the automatic reconfiguration system.

As a second example, consider the application of coscheduling to the problem of "distributed heterogeneous supercomputing" [7, 8]. 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 [7, 8]. 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 coscheduling approach can be used in the development of a solution to this problem. All resources are to be available concurrently, code segments from only one task are to 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 (η_i's), and effective speedup is to be maximized. In this case, the η_i for each machine is a τ-tuple, where there are τ distinct code types and the value of the η_i τ-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 coscheduling 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 nonintensive 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 coscheduling, a form of resource management required by applications with subtasks that need to communicate and synchronize during execution. Coscheduling 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 coschedule of Q workstations are presented and analyzed.

7. REFERENCES

1. Alonso, R., and Cova, L. L. Sharing jobs among independently owned processors. *Proc. 8th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1988, pp. 282–288.
2. Black, D. L. Scheduling support for concurrency and parallelism in the mach operating system, *IEEE Comput.* **23,** 5 (May 1990), 35–43.
3. Casavant, T. L., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Engrg.* **SE-14,** 2 (February 1988), 141–154.
4. Chu, C., Delp, E. J., Jamieson, L. H., Siegel, H. J., Weil, F. J., and Whinston, A. B. A model for an intelligent operating system for executing image understanding tasks on a reconfigurable parallel architecture. *J. Parallel Distrib. Comput.* **6,** 3 (June 1989), 598–622.
5. Douglis, F., and Ousterhout, J. Process migration in the Sprite operating system. *Proc. 7th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1987, 18–25.
6. Fox, G. C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., and Waker, D. W. *Solving Problems on Concurrent Processors*. Prentice–Hall, Englewood Cliffs, NJ, 1988.
7. Freund, R. Optimal selection theory for superconcurrency. *Proc. Supercomputing '89*. 1989, pp. 699–703.
8. Freund, R., and Conwell, D. S. Superconcurrency: A form of distributed heterogeneous supercomputing. *Supercomp. Rev.* **3,** 10 (October 1990), 47–50.
9. Hagmann, R. Processor server: Sharing processing power in a workstation environment. *Proc. 6th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1986, pp. 260–267.
10. Kleinrock, L., and Korfhage, W. Collecting unused processing capacity: An analysis of transient distributed systems. *Proc. 9th International Conference on Distributed Computing Systems*. IEEE Computer Society. 1989, pp. 482–489.

11. Krueger, P., and Chawla, M. The stealth distributed scheduler, *Proc. 11th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1991, pp. 336–343.
12. Litzkow, M. J., Livny, M., and Mutka, M. W. Condor—The hunter of idle workstations. *Proc. 8th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1988, pp. 104–111.
13. Marinescu, D. C., Rice, J. R., Waltsburger, B., Houstis, C. E., Kunz, T., and Waldschmidt, H. Distributed superconducting. *Proc. Workshop Future Trends in Distributed Computing*. IEEE Computer Society, 1990, 381–387.
14. Mutka, M. W., and Livny, M. Scheduling remote processing capacity in a workstation-processor bank network. *Proc. 7th International Conference on Distributed Computing Systems*. IEEE Computer Society, 1987, pp. 2–9.
15. Ousterhout, J. K. Scheduling techniques for concurrent systems. *Proc. 3rd International Conference on Distributed Computing Systems*. 1982, pp. 22–30.
16. Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. Medusa: An experiment in distributed operating system structure. *Comm. ACM* **23,** 2 (February 1980), pp. 92–105.
17. Siegel, H. J., Schwederski, T., Kuehn, J. T., and Davis, T. J., IV. An overview of the PASM parallel processing system. In D. D. Gajski, D. D., Milutinovic, V. M., Siegel, H. J., and Furht, B. P. (Eds.), *Computer Architecture*. IEEE Computer Society, Washington, DC, 1987, pp. 387–407.
18. Siegel, L. J., Siegel, H. J., and Swain, P. H. Performance measures for evaluating algorithms for SIMD machines. *IEEE Trans. Software Engrg.* **SE-8,** 4 (July 1982), 319–331.
19. Swan, R. J., Fuller, S. H., and Siewiorek, D. P. Cm*: A modular multi-microprocessor. *National Computer Conference, AFIPS Conference Proceedings*. 1977, Vol. 46, pp. 645–655.
20. Taylor, D. The promise of cooperative computing. *HP Design and Automation* **5,** 1 (November 1989), 25–36.
21. Theimer, M. M., and Lantz, K. A. Finding idle machines in a workstation-based distributed system. *Proc. 8th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 1988, pp. 112–122.

MIKHAIL J. ATALLAH received the B.E. in electrical engineering from the American University, Beirut, Lebanon, in 1975 and the M.S.E. and Ph.D. in electrical engineering and computer science from Johns Hopkins University, Baltimore, Maryland, in 1980 and 1982, respectively. In August 1982 he joined Purdue University, West Lafayette, Indiana, where he is currently a professor of computer science. In 1985 he received a Presidential Young Investigator award from the National Science Foundation. His research interests include the design and analysis of algorithms, parallel computation, and computational geometry. Dr. Atallah is a member of the Association for Computing Machinery, the Society for Industrial and Applied Mathematics, and a Senior Member of the Institute of Electrical and Electronics Engineers. He serves on the editorial boards of the journals *Computational Geometry: Theory and Applications, Information Processing Letters, International Journal on Computational Geometry and Applications, Methods of Logic in Computer Science, Parallel Processing Letters,* and *SIAM Journal on Computing,* and is Guest Editor for a special issue of *Algorithmica* on computational geometry. He has also served on conference program committees and state and federal panels.

CHRISTINA LOCK BLACK graduated from Purdue University with a Master of Computer Science Degree in December 1990. She is currently employed by Grumman Data Systems as the knowledge engineer on a rule-based, voice-based inspection workstation project. She is a member of the IEEE and IEEE Computer Society.

DAN C. MARINESCU has been an associate professor in the computer science department at Purdue University since 1984. His research areas are Petri nets, distributed systems and networking, performance evaluation, parallel processing, and scientific computing. He has published over 60 papers in research journals, conference proceedings, or chapters of books in these areas. He has been the chief architect of a distributed real-time data acquisition and analysis system used for nuclear and high energy physics experiments.

H. J. SIEGEL received two B.S. degrees from the Massachusetts Institute of Technology (MIT), and M.A., M.S.E., and Ph.D. degrees from Princeton University. He is a professor and the Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue University. He has coauthored over 150 technical papers, coedited four volumes, authored one book, consulted, given tutorials, and prepared video tape courses, all on parallel processing. He was general chairman of the Third International Conference on Distributed Computing Systems (1982), program cochairman of the 1983 International Conference on Parallel Processing, general chairman of the Fifteenth Annual International Symposium on Computer Architecture (1988), and program chair of Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation (1992). He has been a guest editor of two issues of the *IEEE Transactions on Computers*. He is a Fellow of the IEEE and a member of the Association of Computing Machinery.

THOMAS LEE CASAVANT received the B.S. in computer science in 1982, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Iowa in 1983 and 1986. From 1986 to 1989, Dr. Casavant was on the faculty of the School of Electrical Engineering at Purdue University, where he also served as director of the Parallel Processing Laboratory. He is currently an associate professor of electrical and computer engineering at the University of Iowa. His research interests include parallel processing, computer architecture, programming environments for parallel computers, and performance analysis. Dr. Casavant has published over 50 technical papers on parallel and distributed computing and has presented his work at tutorials, invited lectures, and conferences in the United States, Asia, and Europe. He served as a guest editor for the August 1991 issue of *IEEE Computer* on distributed computing systems, and as guest editor for the March 1993 issue of the *Journal of Parallel and Distributed Computing* on visualization for parallel processing. Dr. Casavant is a senior member of the Institute of Electrical and Electronics Engineering and a member of the Association of Computing Machinery.

Received December 1, 1991; revised May 10, 1992; accepted June 9, 1992