JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 19, 271–278 (1993)

A Methodology for Exploiting Concurrency among Independent Tasks in Partitionable Parallel Processing Systems*

WAYNE G. NATION

IBM Corporation, Highway 52 and 37th Street NW, Rochester, Minnesota 55901

AND

ANTHONY A. MACIEJEWSKI AND HOWARD JAY SIEGEL

Parallel Processing Laboratory, School of Electrical Engineering, Purdue University, West Lafayette, Indiana 47907-1285

One benefit of partitionable parallel processing systems is their ability to execute multiple, independent tasks simultaneously. Previous work has identified conditions such that, when there are k tasks to be processed, partitioning the system so that all k tasks are processed simultaneously results in a minimum overall execution time. An alternate condition is developed that provides additional insight into the effects of parallelism on execution time. This result, and previous results, however, assume that execution times are data independent. It is shown that data-dependent tasks do not necessarily execute faster when processed simultaneously even if the condition is met. A model is developed that provides for the possible variability of a task's execution time and is used in a new framework to study the problem of finding an optimal mapping for identical, independent data-dependent execution time tasks onto partitionable systems. Executing one, some, or all of the k tasks simultaneously is considered. Because this new framework is general, it can also serve as a new method for the study of data-independent tasks. Extension of this framework to situations where the k tasks are nonidentical is discussed.

1. INTRODUCTION

Large-scale parallelism involves the use of thousands of processors cooperating to process a task, where a *task* is an instance of a problem that can be solved on one or more processors, independent of other tasks. In many cases, one way to efficiently utilize a large-scale parallel processing system is to partition the system, allowing a collection of tasks to execute concurrently, each on a portion of the entire machine. This work examines when partitioning is beneficial for minimizing the execution time of a set of multiple concurrent tasks by exploiting partitioning.

* This work was 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-1937.

Partitionable systems can be subdivided into smaller independent submachines of various sizes. Advantages of partitionable systems include fault tolerance, fault detection, program development on smaller submachines, subtask parallelism, and multiple simultaneous tasks [21]. The last advantage is addressed here. Specifically, given a set of identical, independent tasks, the problem of determining the "best" number of processors to allocate to each task and the way the tasks may be "placed" on the system in relation to one another to achieve a minimum overall execution time is explored. Such situations where multiple identical, independent tasks are to be performed occur frequently. The wealth of applications that can exploit concurrency of independent tasks ranges from system simulation, where many copies of the simulator are executed, each with a different set of input parameters, to applications involving the processing of a set of images where the same operation is performed independently on each image.

A simulation study [11] pointed out that a set of four identical global histogram tasks execute in the shortest time if all four are processed simultaneously, each on a submachine consisting of one-fourth of the partitionable system. An analytical study [9] showed that this result is true for all algorithms whose execution times on various submachine sizes match a certain condition. That is, given an N-processor partitionable system and k identical, data-independent tasks matching the condition, it is always best to partition the system into k submachines, each of size N/k, and process all tasks simultaneously. An alternate condition is developed here that provides additional insight into the effects of the use of parallelism on the execution time. This result, and most previous work, implicitly assumes that execution time is *data-independent*, i.e., all k tasks have exactly the same execution time on any data set, given the same number of processors. Likewise, for tasks with *data-dependent* execution time, two or more identical tasks may have differ-

0743-7315/93 $5.00

ent execution times, even though they are given the same number of processors. The work here extends previous results by coupling an expression representing the execution time of a single task with a new expression for the total execution time of a collection of tasks where one, some, or all of the k tasks are executed simultaneously. On this new platform the issue of algorithms with data-dependent execution time is addressed, where it is seen that it may not be best to process all tasks simultaneously, even if the data-independent condition derived in [9] is met. Because this new framework is general, it can also serve as a new method for the study of data-independent tasks. In [12], the task allocation problem for data-dependent tasks was introduced. Execution time was modeled as a random variable and several specific parallel algorithms were studied in detail.

While this work has a heavy emphasis on the mapping of tasks onto a set of processors, it does not consider the problem of decomposing a single task (or sets of communicating tasks) and finding an optimal mapping of that decomposition onto a parallel system. That is a related problem and has received much attention in the literature, e.g., [3, 4, 6, 8, 16, 23]. Stated concisely, results from these related works indicate that the work of a task should be distributed as evenly as possible while minimizing interprocessor communication. However, these related works cannot be applied directly to the problem of allocating identical, data-dependent tasks onto partitionable systems. This is because the models used do not take into account the variability of execution time, based on the nature of the algorithm and data, and the interactions of this variability on the spatial and temporal juxtaposition of other tasks.

The architectural model assumed is a partitionable system with N processors, N memory modules, and an interconnection network. The *interconnection network* provides for communications among processors and memory modules of the system and must also support partitioning. When the system is divided into submachines the network topology must ensure that the computations and communications in one submachine do not interfere with other submachines [17]. Examples of interconnection networks suitable for large-scale parallel processing systems that support partitionability are the single-stage cube (hypercube) [17], multistage cube [18, 20], and ADM/IADM [18]. The processors may be paired with local memory modules to form *processing elements* (*PEs*) communicating via message passing. This is the *PE-to-PE* configuration. With the *processor-to-memory* configuration, processors are placed on one side of the interconnection network, memory modules on the other side, and communication among processors is through shared memory. The results here apply to both configurations. Some partitionable systems that have been constructed include MIMD systems (nCube [7]), SIMD systems with multiple control units (CM-2 [22]), and reconfigurable SIMD/MIMD systems (PASM [19,21] and TRAC [13]). It is seen that the results are also applicable to nonpartitionable SIMD systems.

Section 2 presents a general expression for the execution time of a single task that is used to represent the execution time of k tasks on N processors where either N or N/k processors are used to process each data-independent task. Basic properties of this expression, and thus of the algorithm it represents, are derived. Assigning arbitrary numbers of processors to data-independent tasks is considered in Section 3. In Section 4, calculating partition sizes to minimize the execution time of sets of identical, data-dependent tasks is examined.

2. GENERAL MODEL OF EXECUTION TIME

This section introduces a general expression for the time required to execute a single task in parallel and explores properties of the expression. A task will require t_s seconds to execute on a serial machine. This represents the minimum amount of work needed to process the task and assumes an optimal coding of the "best" serial algorithm. Because this discussion is based on execution time and not work, it is necessary to assume that the serial machine is based on a single processor of the same computational power that is used in each of the N processors of the parallel machine. It is assumed that any parallel program for the task would distribute this minimum amount of work among N processors such that at least t_s/N seconds are required to execute the task. While this may not always be true (due to, for example, elimination of a loop index when N processors are used), it is a reasonable approximation.

Often, a different algorithm is chosen for the parallel program because the "best" serial algorithm. The added instructions in the parallel program related to the change in basic computation method is *algorithmic overhead* [14]. Also, the parallel program may incorporate some additional instructions to handle communication and/or synchronization. The time spent by a parallel machine on communication, synchronization, and algorithmic overhead is the *overhead for parallelism* ($V(N)$). Included in the overhead for parallelism is the time spent on *intratask idle time*, due to memory communication latency.

Thus, the time to execute a task on N processors, $t_p(N)$, is $t_p(N) = t_s/N + V(N)$, which is a general expression that parallel execution time is the sum of the time spent on the minimum amount of work that is necessary and on overhead for parallelism. The expression $t_p(N)$ specifies *total* execution time for N processors, and does not indicate anything about the time spent by an *individual* processor on either the required minimum amount of work (its part of t_s) or overhead for parallelism. Previous studies of the decomposition of a problem onto N processors use similar general expressions, e.g., [4, 14, 23].

The problem of deciding whether to partition a parallel machine and execute all k tasks simultaneously, each on an N/k-processor submachine, or to process the k tasks sequentially, each on an N-processor machine (in effect, by not partitioning), reduces to determining the validity of the expression $t_p(N/k) \leq k \cdot t_p(N)$. Theorem 1 shows when this expression is true for data-independent tasks.

THEOREM 1. *The overhead function for each of the k identical data-independent tasks satisfies the condition $d\,V(N)/d\,N = V'(N) \geq -V(N)/N$, if and only if $t_p(N/k) \leq k \cdot t_p(N)$.*

Proof. The proof is straightforward once $N \cdot V(N)$ is shown to be monotonically increasing (i.e., $d\,V(N)/d\,N = N \cdot V'(N) + V(N) \geq 0$) [15]. The monotonicity implies $(N/k) \cdot V(N/k) \leq N \cdot V(N)$ from which the result follows by algebra. A discrete version of Theorem 1 can be derived but is also omitted here for the sake of brevity. ■

Solving $V'(N) = -V(N)/N$ gives the overhead function with the minimum allowed rate of change such that $t_p(N/k) \leq k \cdot t_p(N)$ is true. This "minimal" overhead function has the form $V(N) = A/N$, where A is a positive constant (i.e., for $V(N) = A/N$, $V'(N) = d\,(A/N)/d\,N = -A/N^2 = -V(N)/N$). Intuitively, an algorithm with this "minimal" overhead function consists of a constant amount of overhead for parallelism that is evenly distributed over all processors. Thus, the condition $V'(N) \geq -V(N)/N$ implies that all algorithms have overhead functions satisfying $V(N) \geq A/N$. Not only can $V(N)$ be any increasing function of N, but it can be a decreasing function as well.

The condition of Theorem 1 implies that $N \cdot V(N)$ is monotonically increasing which means that when increasing the submachine size, the overhead for parallelism ($V(N)$) may decrease but only by a factor less than the increase in submachine size. This property is analogous to the condition given in [9]. Here, the condition $V'(N) \geq -V(N)/N$ is used to provide additional insight.

In the next section, Theorem 1 is built upon to consider arbitrary submachine sizes, in addition to those of size N or N/k. Both data-independent and data-dependent tasks are considered.

3. ALLOCATING TASKS HAVING DATA-INDEPENDENT EXECUTION TIMES

This section studies the benefits of partitioning a parallel system to minimize the total execution time of collections of tasks having data-independent execution time. First, a general expression to represent the execution time of k tasks is derived that incorporates other strategies (e.g., allocating more than N/k processors to tasks such that submachines process some number of tasks sequentially). This expression forms the basis of the framework for studying data-dependent tasks in a later section, and, thus, its derivation is of importance even though in this section it is used to reinforce known results [9].

The *total execution time* of k tasks, T_k, is the time elapsed from the time the first task begins execution to the time the last task has concluded. The total time that processor ϕ, $0 \leq \phi < N$, is busy working on any one or more of the k tasks in B_ϕ (which includes intra-task idle time). For the model used here, *intertask idle time* is denoted I_ϕ, which is defined to be the total time that processor ϕ, $0 \leq \phi < N$, is not working on any of the k tasks. I_ϕ also includes time spent by processors waiting for the last task to complete. Therefore, the value of $B_j + I_j = B_i + I_i = T_k$, $0 \leq i, j < N$, and T_k can be expressed as

$$T_k = \frac{1}{N} \sum_{\phi=0}^{N-1} (B_\phi + I_\phi) = B_\phi + I_\phi \qquad \text{for any } \phi.$$

Let n_j be the number of processors assigned to task j, $0 \leq j < k$, and $t_{pj}(n_j)$ be the time to process task j on n_j processors and let t_{sj} and $V_j(n_j)$ be the serial execution time of task j and the overhead for parallelism when task j is executed on n_j processors, respectively. Then the preceding expression reduces to [15]

$$T_k = \frac{1}{N} \sum_{j=0}^{k-1} t_{sj} + \frac{1}{N} \sum_{j=0}^{k-1} n_j \cdot V_j(n_j) + \frac{1}{N} \sum_{\phi=0}^{N-1} I_\phi.$$

The t_{sj} component of T_k is independent of the task allocation strategy and, thus, can be ignored when comparing the relative merits of any two strategies. Overhead of parallelism is represented by the second term and is an explicit function of the number of processors assigned to each task and the overhead function for that task (i.e., the penalty for using parallelism for the single task, which includes the intratask idle time). The idle time given by the third term of T_k depends on the relative placement of the tasks in time and space on the partitionable system and is a measure of the penalty paid for intertask idle time. With this framework, the general result can be proven.

THEOREM 2. *The total execution time, T_k, for k identical tasks with data-independent execution times satisfying the condition $V'(N) \geq -V(N)/N$ on an N-processor partitionable system is minimized when each task is allocated N/k processors and all tasks are processed simultaneously.*

Proof. When all tasks are identical and are processed simultaneously, each on N/k processors, no processor experiences intertask idle time (i.e., $I_\phi = 0$, $0 \leq \phi < N$). It is claimed that this results in the *minimum execution*

time ($(T_k)_{\min}$) and that all other task-to-submachine assignments will not result in a T_k less than $(T_k)_{\min}$, where

$$(T_k)_{\min} = \frac{1}{N}\sum_{j=0}^{k-1} t_{sj} + \frac{1}{N}\sum_{j=0}^{k-1} (N/k) \cdot V_j(N/k)$$
$$= t_{sj}/(N/k) + V_j(N/k) = t_{pj}(N/k).$$

All other possible assignments of tasks to submachines fall into two cases: (A) one or more tasks are assigned to less than N/k processors, and (B) one or more tasks are assigned to more than N/k processors while the rest (if any) are assigned to N/k processors. The remainder of the proof shows that no assignment in either Case A or Case B results in an execution time of less than $(T_k)_{\min}$.

Case A. Choose one of the tasks assigned to less than N/k processors, j'. The execution time of task j' is $t_{pj'}(n_{j'})$ and is no less than $t_{pj}(N/k) = (T_k)_{\min}$. If such a case arises, where a smaller submachine size yields a smaller execution time, then it can be shown that the algorithm for the larger submachine is suboptimal. The larger submachine algorithm can be improved to match the performance of the smaller submachine algorithm by using the smaller submachine algorithm on the larger submachine and forcing some of the processors to be idle. Thus, there is at least one task with execution time no less than $t_{pj}(N/k)$ seconds and no assignment in Case A results in an execution time less than $(T_k)_{\min}$.

Case B. Because $n_j V_j(n_j)$ is a monotonically increasing function of n_j (from the Theorem 1 proof):

$$\sum_{j=0}^{k-1} (N/k) \cdot V_j(N/k) \le \sum_{j=0}^{k-1} n_j \cdot V_j(n_j),$$

because $\exists\, j$ such that $n_j > N/k$. It follows that

$$(T_k)_{\min} \le \frac{1}{N}\sum_{j=0}^{k-1} t_{sj} + \frac{1}{N}\sum_{j=0}^{k-1} n_j \cdot V_j(n_j) + \frac{1}{N}\sum_{\phi=0}^{N-1} I_\phi,$$

where $\exists\, j$ such that $n_j > N/k$. The proof is complete. ■

The exploration of several algorithm examples to determine their overhead functions and a demonstration that they meet the $V'(N) \ge -V(N)/N$ condition is available [15].

4. ALLOCATING TASKS HAVING DATA-DEPENDENT EXECUTION TIMES

The basic result of Section 3 is that tasks meeting the condition $V'(N) \ge -V(N)/N$ with data-independent execution time always achieve minimal execution time when all of the tasks are processed simultaneously. If the same strategy (of allocating N/k processors per task and executing tasks simultaneously) is taken when execution times are data-dependent; (1) the tasks will not all conclude at the same time, (2) the total execution time will be determined by the processing time of the longest task, and (3) the processor *not* allocated to the longest task will experience some idle time.

Consider the simplistic case of a data-dependent algorithm with $V(N) = 0$ for all data sets, i.e., there is no penalty for allocating more processors per task. For this case, total execution time is minimized when no submachine is idle at any time during the tasks' execution, i.e., intertask idle time $I_\phi = 0$. The only way to guarantee this is to process the tasks sequentially, each on N processors. This contradicts previous results due to the following. With data-dependent tasks and any other allocation, one submachine may finish before another, implying that there $\exists\phi$ such that $I_\phi > 0$. However, by Theorem 2, with data-independent tasks, an allocation of N/k processors per task yields $I_\phi = 0\ \forall\phi$. Thus, where the results of Section 3 indicated that maximal partitioning minimizes execution time, there are some trade-offs with regard to partitioning when considering tasks with data-dependent execution times.

Because the model of [9] has no provision for idled submachines and forces simultaneous execution of tasks, it is not directly applicable to this problem. Another scheduling algorithm [2] allows sequential execution of some or all tasks, but it assumes that the execution time of a given task with a given number of processors is fixed.

The following analyses model the execution time of tasks as functions of random variables. The notation $\hat{X}$ indicates that X is a random variable. When execution time is modeled as a function of random variables the expression for T_k, the total execution time for k tasks, is a function of random variables. Because the total execution time $\hat{T}_k$ is a random variable, it would be insightful to know $E[\hat{T}_k]$, the expected value of $\hat{T}_k$. The $E[\hat{T}_k]$ is the quantity which is minimized and can be expressed as

$$E[\hat{T}_k] = \frac{1}{N}\sum_{j=0}^{k-1} \sum_{\substack{\forall\phi \text{ where} \\ \text{processor } \phi \\ \text{is allocated} \\ \text{to task } j}} E[\hat{t}_{pj}(n_j)] + \frac{1}{N}\sum_{\phi=0}^{N-1} E[\hat{I}_\phi].$$

Task j has an expected (or average) execution time of $\mu_j(n_j)$ when assigned to n_j processors (i.e., $E[\hat{t}_{pj}(n_j)] = \mu_j(n_j)$) and a standard deviation of $\sigma_j(n_j)$. The $E[\hat{I}_\phi]$ term is dependent on the task allocation strategy and the values of $\mu_j(n_j)$ and $\sigma_j(n_j)$.

One way to determine values for $\mu_j(n_j)$ and $\sigma_j(n_j)$ in a production environment is to require users to provide

collections of typical data sets along with their programs. An automated system could then execute the task on different submachine sizes with the various data sets and collect statistics about the execution time to select appropriate scheduling strategies. More sophisticated users could observe execution times during the coding and debugging phase of development and estimate the execution time statistics that are needed. Algorithmic complexity analyses are yet another method to determine execution time statistics.

For this work what matters is the execution time as a function of the input data and the numbers of PEs allocated. It a collection of *nonidentical* tasks exhibit similar execution-time statistical characteristics, then the task scheduler can treat these tasks as identical; i.e., it is only execution time statistics that matter to the scheduler. Thus, for situations such as these, the framework developed here for identical tasks can be directly extended to nonidentical tasks. For example, this work would be useful in exploiting concurrency among nonidentical tasks in "computing centers," where tasks are diverse but may be well understood. For the most general case, where nonidentical tasks do not exhibit similar execution time statistics, the problem of scheduling such tasks is known to be NP-complete (i.e., the bin-packing problem).

4.1. Strategies To Reduce Execution Time

Consider the following three straightforward task allocation strategies for the situation where all k tasks are identical. Although the implications of the following strategies may be intuitive, it can be seen that each can be analyzed under a common framework. Users can utilize the actual statistics gathered from their applications in this framework to determine the "best" task allocation. Furthermore, this framework can be applied to other strategies, such as combinations of Strategy A.

Strategy A. Tasks are assigned to submachines of lN/k processors and batches of k/l tasks are processed simultaneously. All k/l tasks in a batch must conclude before the next batch can begin processing.

Figure 1 illustrates this strategy by showing a time/space map for the case where $k = 16$ data-dependent tasks are processed in $l = 2$ synchronized batches of $k/l = 8$ tasks. Although tasks are independent of each other and there is little intuitive reason to force submachines to synchronize, this may be the only option for some systems. For example, a SIMD system with a partitionable interconnection network but only one control unit could use this strategy on iterative algorithms, for example; disabling submachines one by one until the last submachine concludes execution.

The total execution time is determined by the sum of

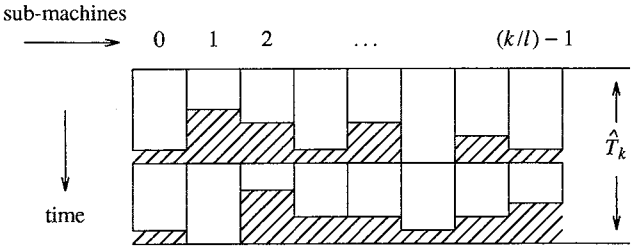

FIG. 1. Time/space map of $k = 16$ data-dependent tasks executing in $l = 2$ synchronized batches tasks on $k/l = 8$ submachines (Strategy A). Each submachine consists of $lN/k = N/8$ processors. The shaded areas indicate time where submachines are idle.

the longest execution time in each batch. Thus, the expected value of $\hat{T}_k$ is l times the expected value of the time to execute one batch of k/l tasks.

$$E[\hat{T}_k] = l \times \left[\mu_j(lN/k) + \frac{1}{N}\sum_{\phi=0}^{N-1} (E[\text{time of longest task in batch}] - \mu_j(lN/k))\right].$$

Intuitively and algebraically, this reduces to $E[\hat{T}_k] = l \times E[\text{time of longest task in batch}]$, and, using well known properties of *order statistics,* an upper bound is known [5]:

$$E[\hat{T}_k] \le l \cdot \mu_j(lN/k) + l \cdot \sigma_j(lN/k) \times \frac{(k/l - 1)}{(2k/l - 1)^{1/2}}.$$

If it is known that the distribution of $\hat{t}_{pj}(lN/k)$ is symmetric, then a better upper bound exists [5]. Also, an exact solution for the optimal value of l can be found by collecting execution time statistics or modeling execution time stochastically.

Strategy B. All tasks are assigned to submachines of lN/k processors and each submachine processes l tasks sequentially.

Figure 2 illustrates this strategy by showing a time/space map for the case where $k = 16$ data-dependent tasks are processed on k/l submachines. Each submachine executes $l = 2$ tasks in sequence.

The random variable $\hat{s}_l(lN/k)$ denotes the execution time of a sequence of l tasks on lN/k processors. Thus,

$$E[\hat{T}_k] = l \cdot \mu_j(lN/k) + \frac{1}{N}\sum_{\phi=0}^{N-1} (E[\text{time of longest sequence}] - E[\hat{s}_l(lN/k)]).$$

Because $E[\hat{s}_l(lN/k)] = l \cdot \mu_j(lN/k)$, intuitively and alge-

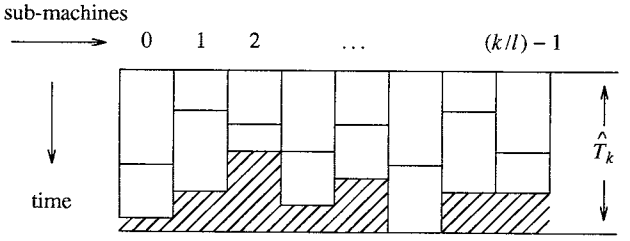

FIG. 2. Time/space map of $k = 16$ data-dependent tasks executing on $k/l = 8$ submachines, for $l = 2$ (Strategy B). Each submachine consists of $lN/k = N/8$ processors and executes $l = 2$ tasks in sequence. The shaded areas indicate time where submachines are idle.

braically, this reduces to

$$E[\hat{T}_k] = E[\text{time of longest sequence}].$$

The standard deviation of $\hat{s}_l(lN/k)$ is $\sqrt{l} \cdot \sigma_j(lN/k)$, and, as with Strategy A, an upper bound is known:

$$E[\hat{T}_k] \leq l \cdot \mu_j(lN/k) + \sqrt{l}\,\sigma_j(lN/k) \times \frac{(k/l - 1)}{(2k/l - 1)^{1/2}}.$$

Also, if it is known that the distribution of $\hat{s}_l(lN/k)$ is symmetric, then a better upper bound exists [5].

There are several ways an exact solution may be found. One method involves the observation of the execution times for sequences of l tasks to form an exact probability distribution for $\hat{s}_l(lN/k)$. Once this probability distribution function is known, the expected value of the longest time of k/l sequences can be calculated numerically by the same technique shown for Strategy A. Because

$$\hat{s}_l(lN/k) = \sum_{i=0}^{l-1} t_{pj}(lN/k),$$

the probability distribution of $\hat{s}_l(lN/k)$ is the result of l time convolutions of $t_{pj}(lN/k)$. Thus, the statistics of a task sequence can be found by observing the execution times of individual tasks, obviating the need to observe execution times of task sequences.

The question is, what value of l will minimize $E[\hat{T}_k]$? If $\mu_j(lN/k)$ and $\sigma_j(lN/k)$ are known, then the equation(s) above for the upper bound can be tabulated easily and the value of l that yields the smallest value for $E[\hat{T}_k]$ indicates that lN/k processors should be allocated to each task for this strategy. This assumes that the $\hat{t}_{pj}(lN/k)$ has the external distribution that equals the upper bound.

Strategy C. Tasks are dynamically assigned to submachines of lN/k processors as the submachines become available (each submachine processes an average of l tasks sequentially).

The dynamic assignment of tasks can lead potentially to lower total execution times because there is the assurance that a given task will not be forced to wait for another task to finish if there is an idle submachine in the system. However, the particular submachine that a given task executes on is not known a priori and cannot be preloaded. This introduces some additional overhead because processors may be idled while the next task is being loaded [10]. This is true for both the PE-to-PE and processor-to-memory configurations (this occurs in the processor-to-memory case due to network conflicts that will occur, in general, if data is preloaded arbitrarily). With Strategy A, systems that allow the overlap of I/O and computation, e.g., MPP [1] and PASM [19, 21], can preload tasks so submachines are not idled waiting for the next task to be loaded into memory. Using Strategy B, these systems cannot fully utilize overlapped I/O capabilities because, in general, the target submachines for the next task to be assigned is not known until a submachine becomes available.

With Strategy B, each submachine executes a sequence of l tasks. An ideal schedule may require some submachines to process more than l tasks while others process less, depending on the relative execution time of their tasks. Unfortunately, exact execution times are not known in advance.

There are some cases where, for $\sigma_j(lN/k) \geq 0$, a static schedule of l tasks per submachine (Strategy B) outperforms the dynamic schedule (Strategy C) due to the reduction in overhead. Such a case is illustrated in the following. Recall that for Strategy B an upper bound for the E[time of longest sequence] was given. Likewise, a lower bound [5] for the E[time of shortest sequence] is

$$E[\text{time of shortest sequence}] \geq l \cdot \mu_j(lN/k) - \sqrt{l} \cdot \sigma_j(lN/k) \times \frac{(k/l - 1)}{(2k/l - 1)^{1/2}}.$$

If the expected difference in time between the shortest sequence and the longest sequence is less than the average execution time of a single task, then it is likely that Strategy B offers the ideal schedule. That is, because moving the last task from the longest sequence to the shortest sequence will not reduce total execution time, on average. An exact solution for the average time difference between the shortest and longest sequence is possible if the probability distribution function of $\hat{s}_l(lN/k)$ is known, where $f_l(x)$ and $F_l(x)$ are the probability distribution and cumulative distribution functions of $\hat{s}_l(lN/k)$, respectively. The probability distribution function $f_l(x)$ is

the l-fold convolution of the probability distribution function of $\hat{t}_{pj}(lN/k)$ with itself. Thus [5],

E[time of shortest sequence] =

$$\sum_{x=0}^{\infty} x \cdot (k/l) \cdot f_l(x) \cdot (1 - F_l(x))^{k/l-1}$$

and

E[time of longest sequence] =

$$\sum_{x=0}^{\infty} x \cdot (k/l) \cdot f_l(x) \cdot F_l(x)^{k/l-1}.$$

If the following condition is true, then it is better, on average, to use Strategy B rather than Strategy C:

$$\mu_j(lN/k) \stackrel{?}{\leq} \sum_{x=0}^{\infty} x \cdot (k/l) \cdot f_l(x) \cdot F_l(x)^{k/l-1} \cdot (1 - F_l(x))^{k/l-1}.$$

The assumption that the execution times are independently and identically distributed holds for a broad class of problem domains. For example, in image understanding programs, the size, shape, type, and number of elements to be discerned may vary widely from image to image. Speech understanding exhibits similar characteristics. A study that addresses the use of Strategy A in a more concrete example is available [15].

5. CONCLUSION

Previous work indicates that, when there are k tasks to be processed and the execution times of the tasks on various submachines meet a certain condition, partitioning the system such that all k tasks are processed simultaneously results in a minimum overall execution time. A general expression was formulated to study the execution time of single tasks. From this general expression, an analogous condition was developed that provides additional insight into previous results. This and previous results, however, assume that execution times are data-independent. A new framework that represents the total execution time of a collection of k tasks was developed. This framework provides for the possible variability of a tasks' execution time and was used to study the problem of finding an optimal mapping for identical independent data-dependent execution time tasks onto partitionable systems. It was seen that tasks whose execution times are data-dependent do not necessarily execute faster when processed simultaneously. By applying the methods described here, users of large-scale partitionable parallel systems can begin to predict which machine partitionings minimize the total execution time of sets of identical, data-dependent tasks.

ACKNOWLEDGMENTS

The authors gratefully acknowledge useful discussions with James Armstrong, Nicholas Giolmas, Murray Supple, and Daniel Watson. A preliminary version of this work was presented at the "Sixth International Parallel Processing Symposium, 1992."

REFERENCES

1. Batcher, K. E. Bit serial parallel processing systems. *IEEE Trans. Comput.* **C-31,** 5 (May 1982), 377–384.
2. Belkhale, K. P., and Banerjee, P. Approximate algorithms for the partitionable independent task scheduling problem. *1990 International Conference on Parallel Processing*. IEEE Computer Society, Silver Spring, MD, 1990, Vol. I, pp. 72–75.
3. Bokhari, S. H. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. Software Engrg.* **SE-7,** 6 (Nov. 1981), 583–589.
4. Cvetanovic, Z. The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems. *IEEE Trans. Comput.* **C-36,** 4 (Apr. 1987), 421–432.
5. David, H. A. *Order Statistics*. Wiley, New York, 1970.
6. Freund, R. F. Optimal selection theory for superconcurrency. *Supercomputing '89*. 1989, pp. 699–703.
7. Hayes, J. P., and Mudge, T. N. Hypercube supercomputers. *Proc. IEEE* **77,** 12 (Dec. 1989), 1829–1841.
8. Jamieson, L. H. Characterizing parallel algorithms. In Jamieson, L. H., Gannon, D. B., and Douglass, R. J. (Eds.). *The Characteristics of Parallel Algorithms*. MIT Press, Cambridge, MA, 1987, pp. 65–100.
9. Krishnamurti, R., and Ma, E. The processor partitioning problem in special-purpose partitionable systems. *1988 International Conference on Parallel Processing*. IEEE Computer Society, Silver Spring, MD, 1988, Vol. I, pp. 434–443.
10. Kruskal, C. P., and Weiss, A. Allocating independent subtasks on parallel processors. *1984 International Conference on Parallel Processing*. IEEE Computer Society, Silver Spring, MD, 1984, pp. 236–443.
11. Kuehn, J. T., and Siegel, H. J. Simulation studies of a parallel histogramming algorithm for PASM. *Seventh International Conference on Pattern Recognition* **1984,** 646–649.
12. Kung, H. T. Synchronized and asynchronous parallel algorithms for multiprocessors. In J. F. Traub (Ed.). *Algorithms and Complexity: New Directions and Recent Results*. Academic Press, New York, 1976, pp. 153–200.
13. Lipovski, G. J., and Malek, M. *Parallel Computing: Theory and Comparisons*. Wiley, New York, 1987.
14. Marinescu, D. C., Rice, J. R., and Vavalis, E. A. Communication and control in SPMD parallel numerical computations. Tech. Rep. CSD-TR-981, CAPO Report CER-90-23, Computer Sciences Department, Purdue University, 1990.
15. Nation, W. G., Maciejewski, A. A., and Siegel, H. J. Exploiting concurrency among tasks in partitionable parallel processing systems. *Sixth International Parallel Processing Symposium*. IEEE Computer Society, Silver Spring, MD, 1992, pp. 30–38.

16. Nicol, D. M. Optimal partitioning of random programs across two processors. *IEEE Trans. Software Engrg.* **SE-15,** 2 (Feb. 1989), 134–141.

17. Siegel, H. J. The theory underlying the partitioning of permutation networks. *IEEE Trans. Comput.* **C-29,** 9 (Sept. 1980), 791–801.

18. Siegel, H. J. *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies,* 2nd ed. McGraw–Hill, New York, NY, 1990.

19. Siegel, H. J., Armstrong, J. B., and Watson, D. W. Mapping computer vision related tasks onto reconfigurable parallel processing systems. *Computer* 25, 2 (Feb. 1992), 54–63.

20. Siegel, H. J., Nation, W. G., Kruskal, C. P., and Napolitano, L. M., Jr., Using the multistage cube network topology in parallel supercomputers. *Proc. IEEE* **77,** 12 (Dec. 1989), 1932–1953.

21. Siegel, H. J., Siegel, L. H., Kemmerer, F. C., Mueller, P. T., Jr., Smalley, H. E., Jr., and Smith S. D. PASM: A partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comput.* **C-30,** 12 (Dec. 1981), 934–947.

22. Tucker, L. W., and Robertson, G. G., Architecture and applications of the connection machine. *Computer* **21,** 8 (Aug. 1988), 26–38.

23. Vrsalovic, D., Gehringer, E. F., Segall, Z. Z., and Siewiorek, D. P. The influence of parallel decomposition strategies on the performance of multiprocessor systems. *12th Annual Symposium on Computer Architecture.* IEEE Computer Society, Silver Spring, MD, 1985, pp. 396–405.

Received December 1992; revised February 16, 1992; accepted May 4, 1993

WAYNE G. NATION received the M.S.E.E. and Ph.D. degrees from Purdue University, West Lafayette, IN, in 1986 and 1991. Since 1991, he has been with the IBM Corporation and is currently with the Application Business Systems division in Rochester, MN. In spring 1992, he was on the adjunct faculty at the State University of New York—Binghamton. He has coauthored over 15 technical papers related to his research interests in interconnection networks, partitionable parallel processing systems, parallel algorithms, and the design of the PASM prototype.

ANTHONY A. MACIEJEWSKI received the B.S., M.S., and Ph.D. degrees in electrical engineering from The Ohio State University, Columbus, in 1982, 1984, and 1987, respectively. Since 1988 he has been an assistant professor with the School of Electrical Engineering at Purdue University, West Lafayette. His primary research interests center on solving the equations of motion for robotic systems.

H. J. SIEGEL is a Professor and Coordinator of the Parallel Processing Laboratory in the School of Electrical Engineering at Purdue. He received two B.S. degrees from MIT, and the M.A., M.S.E., and Ph.D. degrees from Princeton. He has coauthored over 150 technical papers, has edited/coedited five volumes, and has authored one book. His current research focuses on interconnection networks, heterogeneous computing, and the use and design of the PASM reconfigurable mixed-mode parallel system. He is a Fellow of the IEEE and was Coeditor-in-Chief of the *Journal of Parallel and Distributed Computing* (1989–1991). He is currently on the Editorial Board of the *IEEE Transactions on Parallel and Distributed Systems* and is Program Chair of the "8th International Parallel Processing Symposium" (1994).