Exploiting Concurrency Among Tasks in Partitionable Parallel Processing Systems

Wayne G. Nation[†] nation@gdls4.vnet.ibm.com
Anthony A. Maciejewski[‡] maciejew@ecn.purdue.edu
Howard Jay Siegel[‡] hj@ecn.purdue.edu

[†]IBM Corporation
Endicott Engineering Laboratory
Endicott, NY 13760 USA

[‡]Parallel Processing Laboratory
School of Electrical Engineering
Purdue University
West Lafayette, IN 47907-1285 USA

Abstract - *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 such 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 will be 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. Extension of this framework to situations where the k tasks are non-identical 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. Thus, large-scale parallel processing system users may be able to minimize the execution time of a set of multiple concurrent tasks by exploiting partitioning. This work examines when partitioning will be beneficial.

Partitionable systems can be subdivided into smaller independent submachines of various sizes to use the systems more efficiently [25]. Here, the problem of determining the "best" number of processors to allocate to each of a given a set of identical, independent tasks is explored, as well as the way the tasks may be "placed" on the system in relation to one another to achieve a minimum overall execution time. Such situations where multiple identical, independent tasks are to be performed occur frequently. Examples in the signal processing domain include the processing of satellite imagery and the intermittent recalibration of radar.

A simulation study [15] 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 [12] 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., the execution time of a task is independent of the data set values. 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 [12] is met.

In [14], 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. It was concluded in [14] that trade-offs exist in choosing the optimum submachine size and scheduling strategy for data-dependent tasks, and that it "is of interest to build abstract models for studying these trade-offs." The development of a general framework for studying the allocation of data-dependent tasks is the focus of this work. It will be seen that this new framework is extendible to cases where the k tasks may not be identical, but have similar execution time statistical characteristics.

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., [5, 7, 9, 11, 18, 22, 27]. Stated concisely, results from these related works indicate that the work of a task should be distributed as evenly as possible while minimizing inter-processor communication. This reinforces the result of [12] because communication is minimized when all tasks are executed simultaneously, i.e., only N/k processors are assigned to each task (minimizing communication) and all N processors are busy. However, these related works cannot be applied directly to the problem of allocating identical, data-dependent tasks

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 number N00014-90-J-1937.

0-8186-2672-0/92 $03.00 © 1992 IEEE

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. The network topology must ensure that the computations and communications in one submachine do not interfere with other submachines [20]. Examples of interconnection networks suitable for large-scale parallel processing systems that support partitionability are the single-stage cube (hypercube) [20], multistage cube [21, 23, 28], ADM/IADM [21], and gamma [19]. 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 the PE-to-PE and processor-to-memory configurations. Some partitionable systems that have been constructed include: MIMD systems (BBN Butterfly [3] and NCube [10]), SIMD systems with multiple control units (CM-2 [26]), and reconfigurable SIMD/MIMD systems (PASM [24] and TRAC [16]). The results are also applicable to non-partitionable systems: SIMD system users can benefit when processing identical tasks (where execution times are independent of the data values) on disjoint subsets of processors, all following a single instruction stream [21].

Section 2 presents a general expression that will be 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 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" parallel algorithm is not necessarily a parallel version of the "best" serial algorithm. The added instructions in the parallel program related to the change in basic computation method is algorithmic overhead [17]. 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 intra-task idle time.

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. It will be a useful representation for parallel execution time in the remaining sections. Previous studies of the decomposition of a problem onto N processors use similar general expressions, e.g., [7, 17, 27].

As stated in Section 1, this work focuses on task allocation strategies for mapping k identical tasks onto a partitionable system such that total execution time is minimized. 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 $t_p(N/k) \leq k\, t_p(N)$. Theorem 1 shows when this expression is true for data-independent tasks.

Theorem 1: If the overhead function for each of the k identical data-independent tasks satisfies the condition

$$\frac{d\,V(N)}{dN} = V'(N) \geq -\frac{V(N)}{N}\ , \text{ then } t_p(N/k) \leq k\, t_p(N)\,.$$

Proof: If the condition $V'(N) \geq -V(N)/N$ is true, then it can be shown that $g(N) = NV(N)$ is monotonically increasing. This result is immediate.

$$V'(N) \geq -V(N)/N$$

$$NV'(N) + V(N) \geq 0$$

$$g'(N) = NV'(N) + V(N) \geq 0$$

$$g'(N) \geq 0$$

Because $g'(N) \geq 0$, $g(N)$ is monotonically increasing [6]. Intuitively, this means that the total time spent by a task on overhead by all processors of a submachine increases as the submachine size increases. The following concludes the proof of Theorem 1. Because $NV(N)$ is monotonically increasing:

$$(N/k)\, V(N/k) \leq N\, V(N)$$

$$V(N/k) \le k\,V(N)$$

$$t_p(N/k) - k\,t_s/N \le k\,(t_p(N) - t_s/N)$$

$$t_p(N/k) \le k\,t_p(N)$$

and the proof is complete. □

A discrete version of Theorem 1 can be derived but is omitted here for the sake of brevity.

The condition of Theorem 1 implies that $N\,V(N)$ is monotonically increasing which, as stated above, 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 [12]. Here, the condition $V'(N) \ge -V(N)/N$ will be used to provide additional insight.

Solving $V'(N) = -V(N)/N$ gives the overhead function with the minimum allowed rate of change such that $t_p(N/k) \le k\,t_p(N)$ is true. This "minimal" overhead function has the form $V(N) = A/N$, where A is a positive constant (i.e., for $V(N) = A/N$, $V'(N) = d(A/N)/dN = -A/N^2 = -V(N)/N$). 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 $V'(N) \ge -V(N)/N$ implies that all algorithms have overhead functions satisfying $V(N) \ge A/N$. Not only can $V(N)$ be any increasing function of N, but it can be a decreasing function as well.

In the next section, Theorem 1 is built upon to consider arbitrary submachine sizes. This requires extending the execution time equation to be able to represent a sequence of tasks being executed on a submachine. Both data-independent and data-dependent tasks are considered.

3. Data-Independent Execution Time Tasks

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 will be 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. Next, the result of Section 2 is generalized to state that, where all tasks are identical, data-independent and satisfy the condition $V'(N) = -V(N)/N$, it is always best to partition a machine and execute all tasks simultaneously.

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, $0 \le \phi < N$, is busy working on any one or more of the k tasks is B_ϕ. Recall from Section 2 that B_ϕ includes intra-task idle time. For the model used here, inter-task idle time is denoted as I_ϕ, which is defined to be the total time that processor ϕ, $0 \le \phi < N$, is not working on any of the k tasks. That is, I_ϕ is the total time that processor ϕ is not a member of some submachine working on a task. I_ϕ also includes time spent by processors waiting for the last task to complete. Therefore, the value of $B_j + I_j = B_i + I_i = T_k$, $0 \le i,j < N$, and T_k can be expressed as

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

If n_j is the number of processors assigned to task j, $0 \le j < k$, and $t_{pj}(n_j)$ is the time to process task j on n_j processors, then

$$T_k = \frac{1}{N}\sum_{\phi=0}^{N-1}\left[\sum_{\substack{\forall j \text{ where}\\ \text{task } j \text{ is}\\ \text{assigned to } \phi}} t_{pj}(n_j)\right] + \frac{1}{N}\sum_{\phi=0}^{N-1} I_\phi\,.$$

Stated equivalently,

$$T_k = \frac{1}{N}\sum_{j=0}^{k-1}\left[\sum_{\substack{\forall\phi \text{ where}\\ \text{processor } \phi\\ \text{is allocated}\\ \text{to task } j}} t_{pj}(n_j)\right] + \frac{1}{N}\sum_{\phi=0}^{N-1} I_\phi\,.$$

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, $t_{pj}(n_j) = t_{sj}/n_j + V_j(n_j)$ and

$$\sum_{\substack{\forall\phi \text{ where}\\ \text{processor } \phi\\ \text{is allocated}\\ \text{to task } j}} (t_{pj}(n_j)) = n_j(t_{pj}(n_j)) = n_j\,(t_{sj}/n_j) + n_j\,V_j(n_j)\,.$$

Then,

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

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 intra-task idle time). The idle time given by the third term of T_k will depend on the relative placement of the tasks in time and space on the partitionable system and is a measure of the penalty paid for inter-task idle time.

With this framework, the general result can be proven. This is analogous to a result reported in [12]. However, here the result is based on an alternative view of execution time (i.e., the $V'(N) \ge -V(N)/N$ condition) and on the new general framework developed here that can also accommodate data-dependent tasks, as seen in the next section.

Theorem 2: The total execution time, T_k, for k identical tasks with data-independent execution times satisfying the condition $V'(N) \ge -V(N)/N$ 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 inter-task idle time (i.e., $I_\phi = 0$, $0 \le \phi < N$). 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

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

$$= t_{sj}/(N/k) + V_j(N/k) = t_{pj}(N/k).$$

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 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 sub-optimal. 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 $n_jV_j(n_j)$ is a monotonically increasing function of n_j (from Theorem 1 proof):

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

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

$$\frac{1}{N}\sum_{j=0}^{k-1} t_{sj} + \frac{1}{N}\sum_{j=0}^{k-1} (N/k)V_j(N/k)$$

$$\le \frac{1}{N}\sum_{j=0}^{k-1} t_{sj} + \frac{1}{N}\sum_{j=0}^{k-1} n_jV_j(n_j) + \frac{1}{N}\sum_{\phi=0}^{N-1} I_\phi$$

and thus,

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

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

The remainder of this section explores an algorithm example to determine its overhead function and to demonstrate that it meets the $V'(N) \ge -V(N)/N$ condition. A PE-to-PE configuration is assumed.

Consider smoothing an $M \times M$ image (see [21], page 111-112). One way to smooth an image is to replace each pixel with the average value of that pixel and its eight nearest neighbors. When the time to "smooth" a single pixel is denoted T_{SMOOTH}, then the serial execution time is $t_s = M^2 \times T_{SMOOTH}$ (ignoring incorrect values computed for boundary pixels). In a parallel implementation, if the PEs are treated as a logical $\sqrt{N} \times \sqrt{N}$ grid, M^2/N pixels are assigned to each PE as an $(M/\sqrt{N}) \times (M/\sqrt{N})$ subimage. To smooth the pixels at the edge of a subimage, pixels from adjacent subimages must be transferred. Therefore, each PE requires at most $M/\sqrt{N}$ pixels from each of its four adjacent neighbors and one pixel from each of the four PEs diagonally adjacent to the PE. Thus, assuming that the time to transfer a pixel is T_{TRANS} and that computation cannot be overlapped with data transfers,

$$t_p(N) = (M^2/N) \times T_{SMOOTH} + (4M/\sqrt{N} + 4) \times T_{TRANS}$$

where $\quad V(N) = (4M/\sqrt{N} + 4) \times T_{TRANS}$

and $\quad V'(N) = -2M/\sqrt{N^3} \times T_{TRANS}$.

The condition $V'(N) \ge -V(N)/N$ reduces to $M \ge -2\sqrt{N}$, which is always true.

At this point, the expression for T_k has been developed and used to show known results for data-independent tasks. T_k was developed to serve as a framework to study data-dependent tasks in the next section.

4. Data-Dependent Execution Time Tasks

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 processors *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., inter-task 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 may $\exists\phi$ 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 will be some trade-offs with regard to partitioning when considering tasks with data-dependent execution times.

Because the model of [12] has no provision for idled submachines, it is not directly applicable to this problem. Also, the scheduling algorithm of [12] does not permit any processor to execute more than one task, i.e., it forces simultaneous execution of tasks.

Another scheduling algorithm [4] allows sequential execution of some or all tasks. However, it assumes that the execution time of a given task with a given number of processors is fixed, i.e., the model used does not include tasks whose execution times are functions of the data set as well as the number of processors used.

The following analyses model the execution time of tasks as functions of random variables and at several points the following question will be raised: what is the expected (mean) execution time of the last task to finish processing? Order statistics will be used to answer this question.

Order statistics is the study of the statistics of ordered sequences of random variables [8]. The notation $\hat{X}$ indicates that X is a random variable. If the execution times of the k tasks are represented by the k random variables $\hat{t}_0, \hat{t}_1, \hat{t}_2, ...,$ $\hat{t}_{k-1}$ and are ordered such that $\hat{t}_{(j)}$ represents the random variable in the sequence with the j-th least value, then $\hat{t}_{(0)} \leq \hat{t}_{(1)} \leq \hat{t}_{(2)} \leq \ldots \leq \hat{t}_{(k-1)}$. (The parentheses in the subscripts denote order.) Thus, the expected value of the largest random variable is equal to the $\mathrm{E}[\hat{t}_{(k-1)}]$. If all that is known about the probability distribution of $\hat{t}_j$ is its mean value μ and standard deviation σ then the following upper bound [8] is known for the $\mathrm{E}[\hat{t}_{(k-1)}]$ where $\hat{t}_j$, $\forall j$, are independently and identically distributed.

$$\mathrm{E}[\hat{t}_{(k-1)}] \leq \mu + \sigma \times \frac{(k-1)}{(2k-1)^{1/2}}\,.$$

If it is known that the distribution of $\hat{t}_j$ is symmetrical, then a better upper bound can be expressed, see [8].

An exact solution for $\mathrm{E}[\hat{t}_{(k-1)}]$ can be found if the probability distribution function $f_j(x)$ of $\hat{t}_j$ is known. The probability distribution function $f_j(x)$ and cumulative distribution function $F_j(x)$ of $\hat{t}_j$ are, by definition, $f_j(x)$ = probability of the event $\{\hat{t}_j = x\}$ and $F_j(x)$ = probability of the event $\{\hat{t}_j \leq x\}$ where $F_j(x) = \int_0^x f_j(t)dt$. The condition that $\hat{t}_j, \forall j$, be identically distributed can be relaxed for the exact solution. The probability distribution function of $\hat{t}_{(k-1)}$, the task with the longest execution time, is $f_{(k-1)}(x)$ and is given [8] by $f_{(k-1)}(x) = F_j(x)^{k-1}\, k\, f_j(x)$. Thus,

$$\mathrm{E}[\hat{t}_{(k-1)}] = \int_0^\infty x\, F_j(x)^{k-1}\, k\, f_j(x)\, dx\,.$$

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. Thus,

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

Because the total execution time $\hat{T}_k$ is a random variable, it would be insightful to know $\mathrm{E}[\hat{T}_k]$, the expected value of $\hat{T}_k$. The $\mathrm{E}[\hat{T}_k]$ is the quantity which will be minimized and can be expressed as

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

Task j has an expected (or average) execution time of $\mu_j(n_j)$ when assigned to n_j processors (i.e., $\mathrm{E}[\hat{t}_{pj}(n_j)] = \mu_j(n_j)$) and a standard deviation of $\sigma_j(n_j)$.

One way to determine approximate 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.

Consider the case where all k tasks are identical and each task is assigned to lN/k processors, for some fixed l, $1 \leq l \leq k$. No fewer than N/k processors are assigned to each task because this forces some processors to remain idle for the duration of T_k. Not all values of l may be feasible due to the partitioning rules of the system under consideration. Thus, for $n_j = lN/k, \forall j$,

$$\frac{1}{N}\left[\sum_{j=0}^{k-1}\left[\sum_{\substack{\forall\phi \text{ where} \\ \text{processor } \phi \\ \text{is allocated} \\ \text{to task } j}} (\mathrm{E}[\hat{t}_{pj}(n_j)])\right]\right] = \frac{1}{N}\left[k\left[\frac{lN}{k}(\mathrm{E}[\hat{t}_{pj}(n_j)])\right]\right],$$

and the equation for $\mathrm{E}[\hat{T}_k]$ reduces to

$$\mathrm{E}[\hat{T}_k] = l\, \mu_j(lN/k) \;+\; \frac{1}{N} \sum_{\phi=0}^{N-1} \mathrm{E}[\hat{I}_\phi]\,.$$

The $\mathrm{E}[\hat{I}_\phi]$ term is dependent on the task allocation strategy and the values of $\mu_j(n_j)$ and $\sigma_j(n_j)$. For this work what matters is the execution time as a function of the input data and the numbers of PEs allocated. Thus, this work would also be useful in exploiting concurrency among *non*-identical tasks in "computing centers," where tasks are diverse but may be well understood.

4.1 Strategies To Reduce Execution Time

Consider the following five straight-forward task allocation strategies for the situation where all k tasks are identical. Although the implications of the following strategies may be intuitive, it will 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 Strategies A-E. An example of the practical application of this method to compare strategies is given in Subsection 4.2.

Strategy A: All tasks are processed concurrently (N/k processors per task).

Strategy B: All tasks are processed sequentially (N processors per task).

Strategy C: 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, i.e., all k/l submachines wait for the last task of the current batch to finish (and synchronize) before continuing.

Strategy D: Tasks are assigned to submachines of lN/k processors and each submachine processes l tasks sequentially, without the synchronization of Strategy C.

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

Strategies A and B are special cases of Strategies C, D, and E when $l=1$ and $l=k$, respectively, but are examined here separately for the intuitive insight they provide.

Strategy A: All tasks are processed concurrently (N/k processors per task, $l=1$).

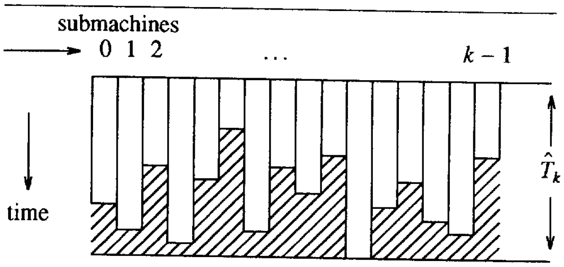

Figure 1: Time/space map of k = 16 data-dependent tasks executing on k = 16 submachines simultaneously (Strategy A). Each submachine consists of N/k processors. The shaded areas indicate time where submachines are idle.

Figure 1 illustrates this strategy. The expected value of $\hat{T}_k$ is

$$E[\hat{T}_k] = \mu_j(N/k) + \frac{1}{N}\sum_{\phi=0}^{N-1}(\,E[\text{time of longest task}] - \mu_j(N/k)\,)\,.$$

Intuitively and algebraically, this reduces to

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

The E[time of longest task] can be found by studying the order statistics of the execution time of the k tasks. If all that is known about the probability distribution of $\hat{t}_{pj}(N/k)$ is $\mu_j(N/k)$ and $\sigma_j(N/k)$ then the following upper bound for Strategy A is known:

$$E[\hat{T}_k\,] = E[\text{time of longest task}]$$

$$\le \mu_j(N/k) + \sigma_j(N/k) \times \frac{(k-1)}{(2k-1)^{1/2}}$$

If the distribution of $\hat{t}_{pj}(N/k)$ is symmetrical, then a better upper bound can be expressed (as described earlier). An approximate solution for $E[\hat{T}_k]$ can be found by following the method described above. This approximate solution can be computed efficiently.

In general, many execution trials must be performed to collect a statistically significant probability distribution function for $\hat{t}_{pj}(N/k)$. However, it may be possible to obtain fairly accurate estimates for $\mu_j(lN/k)$ and $\sigma_j(lN/k)$ with only a relatively smaller number of trials if the input data sets are generally "similar" in nature. Thus, the upper bound equations are useful with limited knowledge of task execution time.

Strategy B: All tasks are processed sequentially (N processors per task, $l=k$).

Because all processors are busy all of the time, $\hat{I}_\phi = 0$, $0 \le \phi < N$. Thus, $E[\hat{T}_k] = k\,\mu_j(N)$.

Strategy C: 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.

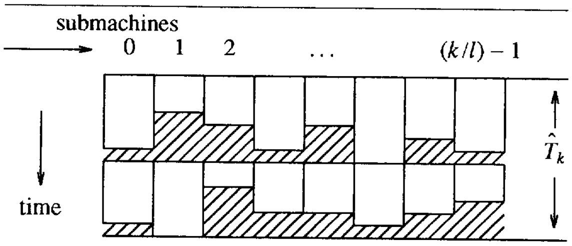

Figure 2: Time/space map of k = 16 data-dependent tasks executing in $l=2$ synchronized batches tasks on $k/l=8$ submachines (Strategy C). Each submachine consists of $lN/k = N/8$ processors. The shaded areas indicate time where submachines are idle.

Figure 2 illustrates this strategy. 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, an 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 the longest execution time in each batch. Thus, the expected value of $\hat{T}_k$ will be l times the expected value of the time to execute one batch of k/l tasks.

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

Intuitively and algebraically, this reduces to

$$E[\hat{T}_k\,] = l \times E[\text{time of longest task in batch}],$$

and, as with Strategy A, an upper bound is known:

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

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

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

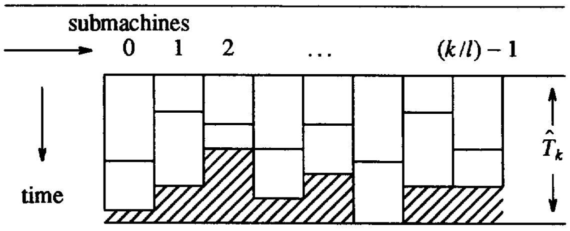

Figure 3: Time/space map of $k = 16$ data-dependent tasks executing on $k/l = 8$ submachines, for $l = 2$ (Strategy D). Each submachine consists of $lN/k = N/8$ processors and executes $l = 2$ tasks in sequence. The shaded areas indicate time where submachines are idle.

Figure 3 illustrates this strategy. The random variable $\hat{s}_l(lN/k)$ denotes the execution time of a sequence of l tasks on lN/k processors. Thus,

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

Because $E[\hat{s}_l(lN/k)] = l\,\mu_j(lN/k)$, intuitively and algebraically, this reduces to

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

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

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

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

There are several ways an approximate solution may be found. One method involves the observation of the execution times for sequences of l tasks to form an approximate 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 $\hat{t}_{pj}(lN/k)$. 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 extremal distribution that equals the upper bound.

When finding an approximate solution for the optimal value of l, it is noted that the collection of run-time data requires the observation of only a single submachine.

Strategy E: 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 will execute 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 [13]. 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 Strategies C and D, systems that allow the overlap of I/O and computation, e.g., MPP [2], PASM [24], can preload tasks so submachines are not idled waiting for the next task to be loaded into memory. Using Strategy E, these systems cannot fully utilize overlapped I/O capabilities.

With Strategy D, 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.

For data-independent tasks (i.e., $\sigma_j(lN/k) = 0$), the ideal schedule is known; each submachine should execute one task (Theorem 2). For this case, dynamic scheduling (Strategy E) and the overhead it incurs is not necessary. In fact, there may be some cases where, for $\sigma_j(lN/k) > 0$, a static schedule of l tasks per submachine (Strategy D) outperforms the dynamic schedule (Strategy E) due to the reduction in overhead. Such a case is illustrated in the following. Recall that for Strategy D an upper bound for the E[time of longest sequence] was shown. Likewise, a lower bound [8] for the E[time of shortest sequence] is

E[time of shortest sequence] ≥

$$l\,\mu_j(lN/k) - \sqrt{l}\,\sigma_j(lN/k) \times \frac{(k/l - 1)}{(2k/l - 1)^{1/2}}\ .$$

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, on average, Strategy D 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. Restated, if the following condition is true,

$$\mu_j(lN/k) \overset{?}{\le} 2\sqrt{l}\,\sigma_j(lN/k) \times \frac{(k/l - 1)}{(2k/l - 1)^{1/2}}\ ,$$

then, on average, it is "likely" that Strategy D offers the ideal schedule. The word "likely" can be removed by providing an exact solution for the average time difference between the shortest and longest sequence. This is possible if the probability distribution function of $s_l(lN/k)$ is known, where $f_l(x)$ and $F_l(x)$ are the probability distribution and cumulative distribution functions of $s_l(lN/k)$, respectively. The probability distribution function $f_l(x)$ is the l-fold convolution of the probability distribution function of $t_{pj}(lN/k)$ with itself. Thus [8],

$$\mathrm{E[time\ of\ shortest\ sequence]} = \sum_{x=0}^{\infty} x\,(k/l) f_l(x) (1 - F_l(x))^{k/l-1}$$

and

$$\mathrm{E[time\ of\ longest\ sequence]} = \sum_{x=0}^{\infty} x\,(k/l) f_l(x) F_l(x)^{k/l-1}\ .$$

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

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

4.2 Applying the Task Allocation Strategies

This subsection explores the use of Strategies A-C in a more concrete example. Consider the class of parallel synchronized iterative algorithms, e.g., those that can be used to solve differential equations, find solutions to systems of equations, and search for extrema in functions, e.g., [1]. Synchronized iterative algorithms are characterized by the repeated execution of a code kernel that consists of a computation phase followed by a communication phase where intermediate data are transferred among processors. Typically, the number of iterations required is data-dependent.

If the time to execute the code kernel on lN/k processors, $t_{ck}(lN/k)$, is represented by $t_{ck}(lN/k) = kt_s/lN + V(lN/k)$ and the number of iterations is modeled as $\hat{i}$, a random variable, then the total time to execute a single synchronized iterative task is $t_p(lN/k) = \hat{i}\,(kt_s/lN + V(lN/k))$. Where the mean and standard deviation of $\hat{i}$ are μ_i and σ_i, then $\mu(lN/k) = \mu_i\,(kt_s/lN + V(lN/k))$ and $\sigma(lN/k) = \sigma_i\,(kt_s/lN + V(lN/k))$.

As an example, recall the image smoothing algorithm from Section 3 and consider an application which calls for the repeated smoothing of an input image until a certain convergence (or "smoothness") criterion is met. In image processing applications one way to remove the effects of aliasing is to smooth an image multiple times. The number of times that an image may have to be smoothed is random. To simplify some of the notation, a change of variables $x = lN/k$ will be used (x is the number of processors in each submachine). Thus, $\mu(x) = \mu_i\,(t_s/x + V(x))$ and $\sigma(x) = \sigma_i\,(t_s/x + V(x))$, where $t_s = M^2$ and $V(x) = (4M/\sqrt{x} + 4) \times T_{TRANS}$. Further assume that the number of iterations required is a uniformly distributed random variable (with mean μ_i and standard deviation σ_i). It is well known that the expected value of the largest of P independently, identically distributed random variables with a uniform distribution having mean μ and standard deviation σ is $\mu + \sqrt{3}\,\sigma(P-1)/(P+1)$ [8]. Given k/l tasks are executed simultaneously, the expected value of the maximum value of $\hat{i}$ is:

$$\mathrm{E}[\hat{i}_{\max}] = \mu_i + \sigma_i \sqrt{3}\,\frac{(k/l - 1)}{(k/l + 1)}\ .$$

Thus, the expected time for the longest task in each batch of k/l tasks is:

$$\mathrm{E[\ longest\ task\ in\ batch\ ]} = (kt_s/lN + V(lN/k))\,\mathrm{E}[\hat{i}_{\max}]\ .$$

and $\mathrm{E}[\hat{T}_k] = l\,\mathrm{E}$[longest task in batch]. By substituting in known quantities for the image smoothing algorithm with $x = lN/k$,

$$\mathrm{E}[\hat{T}_k] = \frac{kN}{x}\left[\frac{M^2}{x} T_{SMOOTH} + (\frac{4M}{\sqrt{x}} + 4) T_{TRANS}\right]\left[\mu_i + \sigma_i \sqrt{3}\,\frac{N-x}{N+x}\right]$$

For $N = 1024$ PEs, $k = 256$ images, where each image is $M \times M = 1024 \times 1024$, and $T_{TRANS} = 4\,T_{SMOOTH}$, $\mathrm{E}[\hat{T}_k]$ reduces to

$$\mathrm{E}[\hat{T}_k] = \mu_i T_{SMOOTH} \frac{(256)(1024)}{x}\left[\frac{1024^2}{x} + \frac{16(1024)}{\sqrt{x}} + 16\right]$$
$$\times\left[1 + \frac{\sigma_i}{\mu_i}\sqrt{3}\,\frac{(1024 - x)}{(1024 + x)}\right].$$

Because the relative and not absolute values of $\mathrm{E}[\hat{T}_k]$ are of interest, it is possible to normalize the solution for $\mu_i = T_{SMOOTH} = 1$.

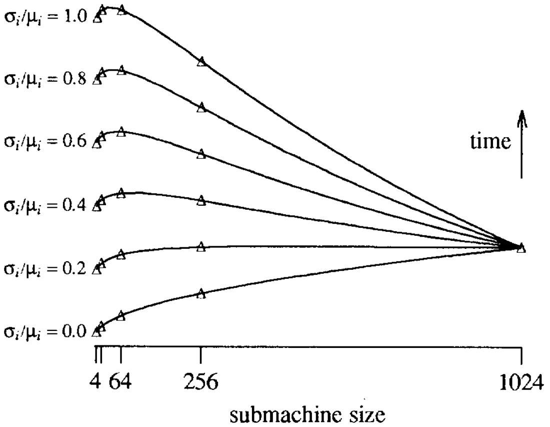

Figure 4: Plot of submachine size versus normalized expected execution time for the data-dependent image smoothing example under Strategy C. The normalized expected execution time is plotted for various values of σ_i/μ_i. Due to the structure of the algorithm, only the points where $l = 4^i$, $0 \leq i \leq 4$ are of interest (denoted by Δ).

Figure 4 is a graph of submachine size versus expected normalized execution time for the image smoothing example under Strategy C. The curve for σ_i/μ_i = 0.0 corresponds to the data-independent case where it is seen that it is best to allocate each task to the smallest submachine size (N/k) and execute all tasks simultaneously. From the graph, this is true for σ_i/μ_i ratios up to ~0.3. The graph in Figure 4 for σ_i/μ_i ratios above ~0.3 indicates that the optimal submachine size is 1024 PEs, i.e., execution time is minimized if tasks are processed sequentially. Because the family of curves for this particular example are concave, the optimal submachine size will be either $N/k = 4$ or $N = 1024$, depending on the σ_i/μ_i ratio. The family of curves for algorithms with different overhead functions yields curves of different shapes. The salient point of this example is that there exists a large class of algorithms with data-dependent execution time whose total execution time can be minimized by applying the techniques of this work.

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. An analogous condition was developed that provides additional insight into previous results because the new condition is based on the time spent on overhead for parallelism, not just the execution time of the task. 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 that 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. Tasks whose execution times are data-dependent do not necessarily execute faster when processed simultaneously.

Using this model and given execution statistics of a task, the choice of partitioning or not can be made based on expected execution times. Because the new framework is general, it also serves as a new method for the study of data-independent tasks. It can also be used for non-identical tasks having similar execution time statistical characteristics. This extension could be useful in exploiting concurrency among tasks in "computing centers" where tasks, while possibly diverse, may be well understood.

Acknowledgement: The authors are grateful for useful discussions with J. Armstrong, N. Giolmas, M. Supple, and D. Watson.

References

[1] V.D. Agrawal and S.T. Chakradhar, "Performance estimation in a massively parallel system," *Supercomputing '90,* Nov. 1990, pp. 306-313.

[2] K.E. Batcher, "Bit serial parallel processing systems," *IEEE Trans. Comp.,* Vol. C-31, May 1982, pp. 377-384.

[3] BBN Advanced Computers, Inc., *Inside the Butterfly Plus,* BBN Advanced Computers, Inc., Cambridge, MA 02238, 1987.

[4] K.P. Belkhale and P. Banerjee, "Approximate algorithms for the partitionable independent task scheduling problem," *1990 Int'l Conf. Par. Proc.,* Aug. 1990, pp. 72-75.

[5] S.H. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. Soft. Eng.,* Vol. SE-7, Nov. 1981, pp. 583-589.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms,* MIT Press, Cambridge, MA, 1990.

[7] Z. Cvetanovic, "The effects of problem partitioning, allocation, and granularity on the performance of multiple-processor systems," *IEEE Trans. Comp.,* Vol. C-36, April 1987, pp. 421-432.

[8] H.A. David, *Order Statistics,* John Wiley & Sons, New York, NY, 1970.

[9] R.F. Freund, "Optimal selection theory for superconcurrency," *Supercomputing '89,* Nov. 1989, pp. 699-703.

[10] J.P. Hayes and T.N. Mudge, "Hypercube supercomputers," *Proc. IEEE,* Vol. 77, Dec. 1989, pp. 1829-1841.

[11] L.H. Jamieson, "Characterizing parallel algorithms," in *The Characteristics of Parallel Algorithms,* L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds., MIT Press, Cambridge, MA, 1987, pp. 65-100.

[12] R. Krishnamurti and E. Ma, "The processor partitioning problem in special-purpose partitionable systems," *1988 Int'l Conf. Par. Proc.,* Aug. 1988, pp. 434-443.

[13] C.P. Kruskal and A. Weiss, "Allocating independent subtasks on parallel processors," *1984 Int'l Conf. Par. Proc.,* Aug. 1984, pp. 236-443.

[14] H.T. Kung, "Synchronized and asynchronous parallel algorithms for multiprocessors," in *Algorithms and Complexity: New Directions and Recent Results,* J.F. Traub, ed., Academic Press, New York, NY, 1976, pp. 153-200.

[15] J.T. Kuehn and H.J. Siegel, "Simulation studies of a parallel histogramming algorithm for PASM," *7th Int'l Conf. Pattern Recognition,* July 1984, pp. 646-649.

[16] G.J. Lipovski and M. Malek, *Parallel Computing: Theory and Comparisons,* John Wiley & Sons, New York, NY, 1987.

[17] D.C. Marinescu, J.R. Rice, and E.A. Vavalis, *Communication and Control in SPMD Parallel Numerical Computations,* Report CSD-TR-981, Computer Sciences Dept., Purdue University, 1990.

[18] D.M. Nicol, "Optimal partitioning of random programs across two processors," *IEEE Trans. Soft. Eng.,* Vol. SE-15, Feb. 1989, pp. 134-141.

[19] D.S. Parker and C.S. Raghavendra, "The gamma network," *IEEE Trans. Comp.,* Vol. C-33, Apr. 1984, pp. 367-373.

[20] H.J. Siegel, "The theory underlying the partitioning of permutation networks," *IEEE Trans. Comp.,* Vol. C-29, Sept. 1980, pp. 791-801.

[21] H.J. Siegel, *Interconnection Networks for Large-Scale Parallel Processing: Theory and Case Studies, Second Edition,* McGraw-Hill, New York, NY, 1990.

[22] H.J. Siegel, J.B. Armstrong, and D.W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel processing systems," *Computer* Vol. 25, Feb. 1992.

[23] H.J. Siegel, W.G. Nation, C.P. Kruskal, and L.M. Napolitano, Jr., "Using the multistage cube network topology in parallel supercomputers," *Proc. IEEE,* Vol. 77, Dec. 1989, pp. 1932-1953.

[24] H.J. Siegel, T. Schwederski, J.T. Kuehn, and N.J. Davis IV, "An overview of the PASM parallel processing system," in *Computer Architecture,* D. D. Gajski, V. M. Milutinovic, H. J. Siegel, and B. P. Furht, eds., IEEE Computer Society Press, Washington, DC, 1987, pp. 387-407.

[25] H.J. Siegel, L.J. Siegel, F.C. Kemmerer, P.T. Mueller, Jr., H.E. Smalley, Jr., and S.D. Smith, "PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition," *IEEE Trans. Comp.,* Vol. C-30, Dec. 1981, pp. 934-947.

[26] L.W. Tucker and G.G. Robertson, "Architecture and applications of the Connection Machine," *Computer,* Vol. 21, Aug. 1988, pp. 26-38.

[27] D. Vrsalovic, E.F. Gehringer, Z.Z. Segall, and D.P. Siewiorek, "The influence of parallel decomposition strategies on the performance of multiprocessor systems," *12th Ann. Symp. Comp. Arch.,* 1985, pp. 396-405.

[28] C.-L. Wu and T.Y. Feng, "On a class of multistage interconnection networks," *IEEE Trans. Comp.,* Vol. C-29, Aug. 1980, pp. 694-702.