On Modeling Partitioned Multiprocessor Systems

Lawrence W. Dowdy[†]
Department of Computer Science
Vanderbilt University
Nashville, Tennessee

Michael R. Leuze[‡]
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, Tennessee

ABSTRACT

In recent years, multiprocessor systems have been developed to exploit parallelism within individual programs. However, a multiprocessor system may be used more efficiently if its processors are partitioned among independent parallel programs. The partitioning of a multiprocessor system is addressed in this paper.

A simple yet powerful model is proposed for the analysis of various partitioning schemes. The model parameterizes both a multiprocessor system and its parallel workload. Attention is restricted to a multiprocessor system partitioned between two parallel programs. The model is studied to determine 1) when partitioning is worthwhile, 2) the extent of performance improvement under optimal partitioning, and 3) which of three partition scheduling schemes is best.

Index Terms — multiprocessor partitioning, multiprocessor performance, performance modeling, sensitivity analysis, workload characterization

---

[†]Supported in part by the Alexander von Humboldt Foundation while on leave to the University of Erlangen-Nürnberg, Erlangen, FRG.

[‡]Supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy.

On Modeling Partitioned Multiprocessor Systems

1. Motivation

Parallelism within a program is exploited by assigning a set of program subtasks to a set of cooperating processes which then execute simultaneously on a set of processors. Often, the work of a program is divided so that the number of tasks to be performed is equal to the number of processors to be used. If an additional processor is made available, it may be possible to reallocate the program's work to a larger number of processes, thereby allowing the program to use the extra processor. A program's characteristic speedup function, sometimes called its "execution signature" [4, 11], relates its execution time to the number of allocated processors. A program's speedup function depends upon how effectively it is able to use additional processors and upon characteristics of the multiprocessor architecture. When allocated an additional processor, a program will typically execute more quickly. However, for every program, there is a maximum number of processors that can be used effectively on a fixed-size problem. This maximum number of processors may be reached when the work of the program is subdivided to the greatest extent possible or when the overhead of communicating with an additional process becomes greater than the benefit derived from the power of an additional processor. Attempting to use more processors may actually increase a program's execution time.

If a program is allocated all of the processors it can use effectively, additional processors may be used by a different, independent program. Even if a program has fewer processors than its effective maximum, additional processors may be better used by another program, since the speedup curve of a program typically flattens as it approaches its maximum. Thus, in attempting to make the best possible use of the resources of a multiprocessor system, it is reasonable to consider dividing the processors among a set of parallel programs. In this paper, we use the term "partitioning" to refer to the process of dividing the processors of a multiprocessor system into sets and the term "partition" to refer to one of these sets.

Our model is based on a high-level view of a parallel program's execution sequence. Programs are submitted to a host machine that acts as a front end to the multiprocessor system. Editing, compiling, I/O, and accounting are the responsibility of the host machine. After compilation, the host assigns a program to a partition of the multiprocessor for execution. When the program completes execution, control passes back to the host for final I/O and accounting.

Ignored in this high-level view are many details, including the number of partitions and processing elements (PEs), the interconnection topology among PEs, communication and synchronization mechanisms between PEs, and the functionality of the various components of the multiprocessor system [12]. However, this view is sufficiently detailed to allow certain issues to be addressed. In particular, we focus upon the partitioning of the PEs among a set of parallel programs and consider the following questions.

- Under what conditions is partitioning worthwhile?
- Given a system and a workload, what are the optimal partition sizes?

- How sensitive is performance to changes in partition sizes?
- How sensitive are the optimal partition sizes to changes in the system or in the workload?
- How should the host assign programs to the various partitions? That is, when is it better to leave an available but suboptimal partition free and wait until a better partition becomes available?

Complete answers to these questions are impossible without detailed implementation, measurement, and experimentation. To provide partial answers, a simple yet powerful model is developed for the analysis of various partitioning schemes. This model is designed to be reflective of actual applications, and is validated experimentally with parallel programs executing on a multiprocessor system. A study of the model identifies those situations not worth further examination and those situations where a more detailed analysis is likely to be beneficial. In this paper, we restrict our focus to the partitioning of a multiprocessor system between two parallel programs. In addition, only statically partitioned systems are considered, since such systems are currently more practical and incur much less overhead than dynamically partitioned systems. There are practical difficulties in suspending a parallel program in execution, repartitioning the system, remapping the program to run on a different number of processors, and resuming execution. This study expands and strengthens the work described in [11].

The model is outlined in Section 2, in which notation and assumptions are summarized. In Section 3, results of the model analysis and validation are presented. Section 4 summarizes the findings and attempts to place them into perspective.

2. The Model

We focus upon a system restricted to having only two partitions. The workload consists of two parallel programs, each with multiple processes. The programs may not be identical and may place different demands upon the system.

2.1. Notation

N — the total number of processing elements (PEs) in the multiprocessor.

n_i — the number of PEs allocated to partition i (P_i). $n_1 + n_2 = N$.

p_i — the fraction of processors allocated to P_i. $p_i = n_i/N$. $p_1 + p_2 = 1$.

(A, B) — the current system state. Initially, it is assumed that there are two programs in the system and that program i ($i = 1, 2$) is always assigned to P_i. "A" represents the location of program 1, either at the host, denoted by H, or in its partition, P_1. "B" represents the location of program 2, either at the host or in its partition, P_2. Thus, the set of allowable system states is $\{(H, H), (P_1, H), (H, P_2), (P_1, P_2)\}$.

$\mu_j(A,B)$ — the state dependent service rate of device j, $j \in \{H, M\}$. "H" denotes the host. "M" denotes the multiprocessor. All service times are assumed to be exponentially distributed.

$S_M(A,B)$ — a state dependent factor used to reduce the effective processing rate of the multiprocessor due to contention for shared resources, such as communication paths or memory.

X — the throughput of the multiprocessor system.

The model is represented by the closed queuing network model in Figure 1.

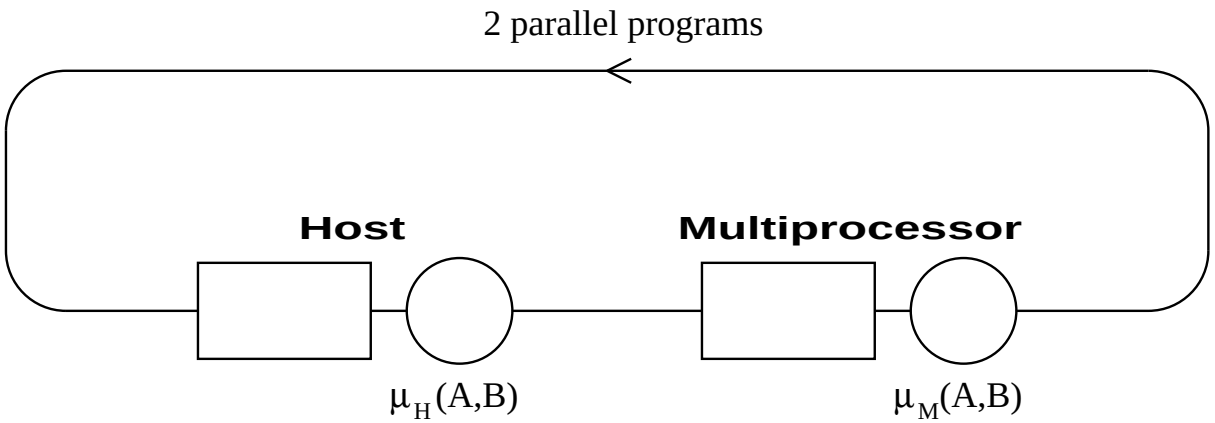

Figure 1: *Queuing network model.*

2.2. Assumptions

Programs circulate between the host and the multiprocessor. When a program finishes execution, it is replaced by an identical program. The host controls the load placed on the multiprocessor. This closed model is equivalent to an open queuing model in which the multiprocessor can service at most two programs simultaneously.

A number of other assumptions about the speed of the host, the speed of the multiprocessor, and the effect of contention at the multiprocessor must be made:

$\mu_H(A,B) = \lambda$. The host treats both programs identically. That is, the demands each program places on the host (e.g., editing, compiling, I/O) are statistically identical. Thus, the host's service time is state independent and is assumed to be exponentially distributed with parameter λ. Processor sharing is assumed.

$\mu_M(P_1,H)$, $\mu_M(H,P_2)$. These are the characteristic speedup functions of the parallel programs. A speedup function specifies the processing rate of a parallel program when no other program is present on the multiprocessor. Speedup functions depend upon the specific architecture and upon of the number of PEs assigned to the program. Thus, in state (P_1,H), only program 1 is active at the multiprocessor using partition 1, which consists of the fraction p_1 of the PEs. (Program 2 is obtaining service at the

host.) Similarly, in state (H, P_2), only program 2 is active at the multiprocessor using partition 2, which consists of the fraction p_2 $(= 1 - p_1)$ of the PEs. Expressions for $\mu_M(P_1, H)$ and $\mu_M(H, P_2)$ are suggested below.

$\mu_M(P_1, P_2) = (\mu_M(P_1, H) + \mu_M(H, P_2)) \times S_M(P_1, P_2)$. When both programs are active in different partitions of the multiprocessor, their relative execution rates are determined by their speedup functions. However, processing rates are slowed by the state dependent factor $S_M(P_1, P_2)$, which is assumed to affect each program's progress equally. Thus, program 1 executes at rate $\mu_M(P_1, H) \times S_M(P_1, P_2)$, and program 2 executes at rate $\mu_M(H, P_2) \times S_M(P_1, P_2)$.

$S_M(H, H) = S_M(P_1, H) = S_M(H, P_2) = 1$. When a single program is active at the multiprocessor, the program experiences no interference.

$S_M(P_1, P_2) = s$. When both programs are executing on the multiprocessor, each program degrades the other's performance. In general, the degree of degradation depends upon a number of factors, including the partition, the architecture, and the workload. To limit complexity, we model degradation with the constant potential factor s, $(0 \leq s \leq 1)$. When $s = 1$, the individual programs do not degrade each other's performance and each executes without interference from the other. When $s = 0$, the interference between programs prohibits either from progressing.

2.3. Speedup functions

The speedup function of a parallel program specifies the rate at which the program executes on a given multiprocessor when no other programs are present. This rate is a function of the specific architecture and of the number of PEs assigned to the program.

Various speedup functions have been proposed in the literature. A simple example from [1] is

$$\mu_M(P_1, H) \equiv \frac{T_1(1)}{T_1(k)} \quad \text{and} \quad \mu_M(H, P_2) \equiv \frac{T_2(1)}{T_2(k)},$$

where $\mu_M()$ is the execution rate of the program when no other program is present at the multiprocessor, and $T_i(k)$ is the time required to execute program i when using k PEs. An advantage of this characterization is its intuitive and historical appeal. Given the $T_i(k)$'s, $k = 1, \ldots, n$, any arbitrary speedup curve can be represented. However, values for each $T_i(k)$ must be known, measured, or assumed. These values depend upon the program, the architecture, and the number of assigned PEs.

With minimal assumptions (e.g., no superlinear behavior), the above speedup function can be bounded by

$$\frac{kA_i}{k + A_i - 1} \leq \mu_M() \leq min(k, A_i)$$

where A_i is the average parallelism of program i [5]. Average parallelism is defined as the average number of busy processors during the execution of the program when as many processors as needed are available. These bounds are independent of the specific values of the

$T_i(k)$'s and depend only upon two parameters, k and A_i. The value of A_i can be empirically observed.

Other speedup expressions have been proposed [2, 3, 8, 9, 13]. For a general comparison of various speedup expressions, their properties, and mappings between them, see [13].

In this paper, speedup functions are assumed to be of the form

$$\mu_M(P_1, H) \equiv \frac{p_1}{D_{11}p_1 + D_{12}} \quad \text{and} \quad \mu_M(H, P_2) \equiv \frac{p_2}{D_{21}p_2 + D_{22}} \tag{1}$$

where $\mu_M()$ is the execution rate of the program when no other program is present at the multiprocessor, p_i denotes the fraction of PEs allocated to program i, and D_{i1} and D_{i2} are empirically determined parameters reflecting characteristics of the architecture and of program i. For example, if $D_{i1} = 0$, program i exhibits linear speedup. As the percentage of processors allocated to program i increases, execution speed increases linearly by the factor $1/D_{i2}$. If $D_{i2} = 0$, program i exhibits a constant execution rate. That is, program i executes at rate $1/D_{i1}$, independent of the number of PEs allocated to it.

Beneficial properties of speedup functions of the form (1) include:

- $D_{i2}/(D_{i1}p_i + D_{i2})^2$ is the first derivative of the speedup function with respect to p_i. Thus, if D_{i2} is positive, allocating more processors to program i will increase its execution speed. Although positive D_{i2} does not allow speedup functions which increase to an optimal number of processors and then decrease, it does allow speedup functions which approach an asymptote. Since that portion of a speedup function beyond a maximum is typically not of interest (processors which degrade performance are not assigned), the use of non-decreasing speedup functions does not limit the usefulness of the model. Thus, D_{i2} will be set to a positive value.

- When D_{i1} and D_{i2} are both positive, the second derivative of the speedup function with respect to p_i, $-2D_{i1}D_{i2}/(D_{i1}p_i + D_{i2})^3$, is negative. Thus, the speedup function is increasing concave. Allocating more processors to the program will cause it to execute faster, but with diminishing returns. Since this property is natural, D_{i1} and D_{i2} will be set to positive values.

- Similarly, when D_{i2} is positive, the mean service time, $1/\mu_M()$, is decreasing convex. This implies that as more processors are allocated, the mean service time decreases, but with diminishing returns.

Intuitively, D_{i1} affects how the speedup function behaves initially, and D_{i2} affects how the speedup function behaves asymptotically. As an example, suppose that program 1 executes at rate 1.0 when allocated 50% of the PEs and at rate 1.1 when allocated 100% of the PEs. The resulting speedup function[1] for program 1 is

$$\mu_M(P_1, H) = \frac{p_1}{0.8182p_1 + 0.0909}. \tag{2}$$

[1]Given two fixed points through which the speedup curve must pass, the values of D_{i1} and D_{i2} are determined easily. For example, for program 1, solving the equations $1.0 = 0.5/(0.5D_{11} + D_{12})$ and $1.1 = 1.0/(1.0D_{11} + D_{12})$ yields $D_{11} = 9/11$ and $D_{12} = 1/11$.

Similarly, suppose that program 2 executes at rate 1.0 when allocated 50% of the PEs and at rate 1.6 when allocated 100% of the PEs. The resulting speedup function for program 2 is

$$\mu_M(H, P_2) = \frac{p_2}{0.2500p_2 + 0.3750}. \tag{3}$$

The speedup functions for programs 1 and 2 are shown in Figure 2a, and their mean service times are shown in Figure 2b.

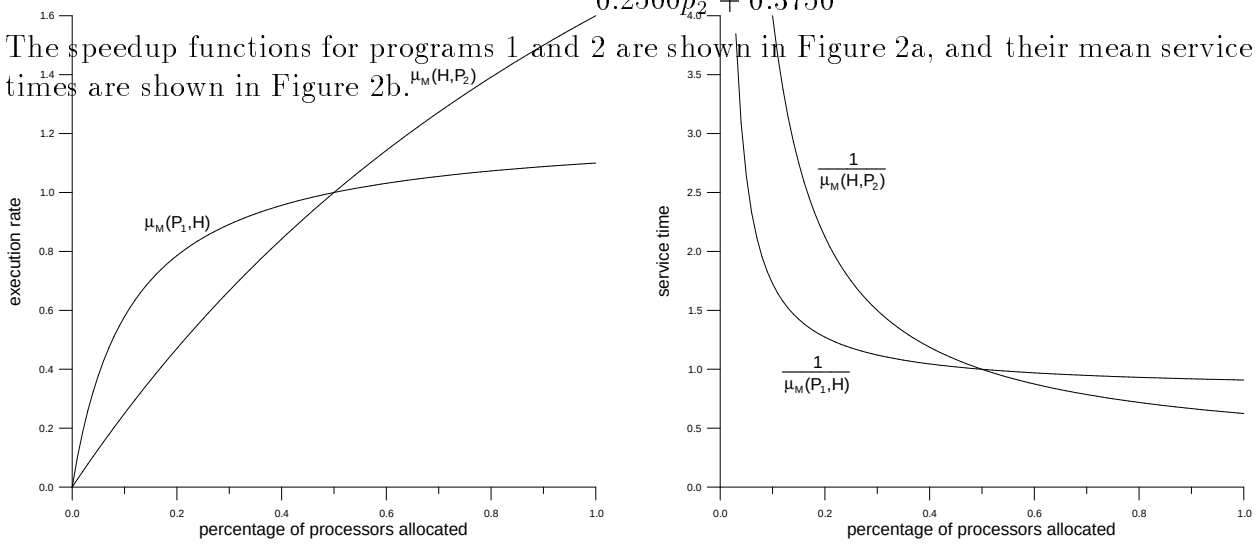

Figure 2: *Speedup functions and mean service times for example programs 1 and 2.*

Other speedup functions with properties similar to form (1) are described in [13]. An example of such a function is

$$\mu_M(P_1, H) \equiv \sqrt{-p_1^2 + 2E_{11}p_1 + E_{12}^2} - E_{12}$$

and

$$\mu_M(H, P_2) \equiv \sqrt{-p_2^2 + 2E_{21}p_2 + E_{22}^2} - E_{22},$$

where E_{i1} and E_{i2} are empirically determined parameters reflecting the specific characteristics of the architecture and of program i, and the remaining parameters are as in form (1) above. This function can represent speedup curves that increase to a unique maximum and then decrease. However, because the goal in this paper is to characterize actual speedup functions with a simple form that is easy to parameterize and easy to use in a comparative analysis, form (1) is sufficient. Validation of the the use of form (1) to characterize an actual workload is given in Section 3.3.

2.4. Model solution

For convenience of notation, let $p_1 = p$ and let $p_2 = q = 1 - p$. The Markov diagram for the state space corresponding to the model is shown in Figure 3. Arcs are labeled with the appropriate transition flow rates.

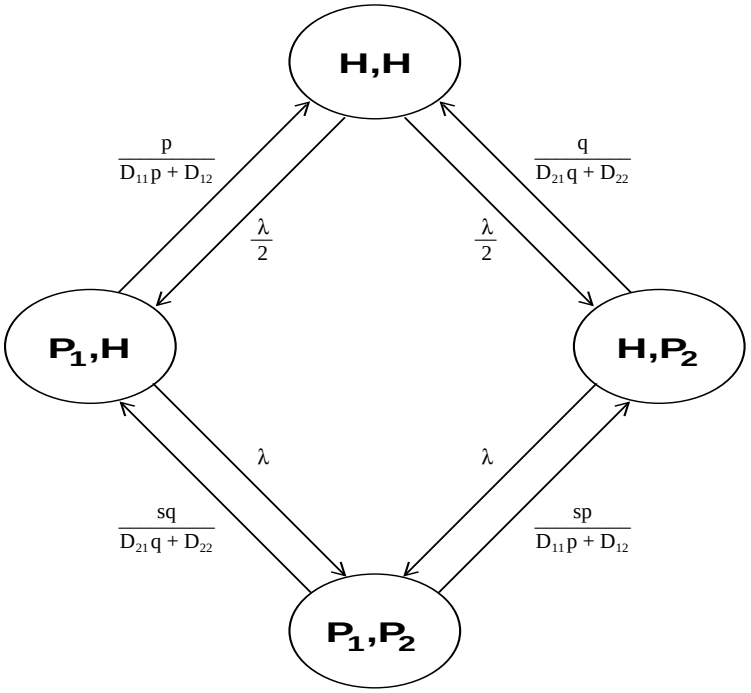

Figure 3: *Base model's Markov diagram.*

The steady state solution, found by applying standard techniques [7] to the global balance equations, is

$$\begin{aligned} P(H,H) &= \frac{1}{G} \\ P(P_1,H) &= \frac{1}{G}\left(\frac{\lambda(D_{11}p + D_{12})}{2p}\right) \\ P(H,P_2) &= \frac{1}{G}\left(\frac{\lambda(D_{21}q + D_{22})}{2q}\right) \\ P(P_1,P_2) &= \frac{1}{G}\left(\frac{\lambda^2(D_{11}p + D_{12})(D_{21}q + D_{22})}{2spq}\right) \end{aligned}$$

where $P(A,B)$ is the steady state probability of being in state (A,B) and where

$$G = 1 + \frac{\lambda(D_{11}p + D_{12})}{2p} + \frac{\lambda(D_{21}q + D_{22})}{2q} + \frac{\lambda^2(D_{11}p + D_{12})(D_{21}q + D_{22})}{2spq}.$$

The performance measure chosen for analysis is total throughput (X), the number of programs per unit time that pass through the host (or, equivalently, through the multiprocessor). Other performance metrics could be chosen. For example, the average response

time (R) of a program (i.e., the time required to cycle once between the host and the multiprocessor) is directly related to X, $R = 2/X$. Power, the ratio of a system's throughput to the system's response time, is likewise related to X, Power$= X/R = X^2/2$.

Throughput, as a function of the system parameters, is

$$X(\lambda, D_{11}, D_{12}, D_{21}, D_{22}, p, s) = \lambda\,(P(H,H) + P(P_1,H) + P(H,P_2))$$

$$= \frac{\lambda^2 s(D_{22}p + D_{12}q + (D_{11}+D_{21})pq) + 2\lambda spq}{\lambda^2(D_{12}D_{22} + D_{11}D_{22}p + D_{12}D_{21}q + D_{11}D_{21}pq) + \lambda s(D_{22}p + D_{12}q + (D_{11}+D_{21})pq) + 2spq}. \qquad (4)$$

Equation (4) represents the throughput of the model as a function of the speed of the host, the potential factor, and the speedup function parameters. The speedup function parameters depend in turn upon the speed of the multiprocessor and upon the partition of the PEs. Expressions for the response time and power of the system are simple functions of throughput, $2/X$ and $X^2/2$, respectively. Thus, maximum throughput, minimum response time, and maximum power all occur at the same point.

3. Results

In this section, the sensitivity of the throughput expression is analyzed, the model is compared against models for other partitioning schemes, validation is discussed, and extensions to the model are examined. For a more complete set of sensitivity curves, see [4].

3.1 Sensitivity analysis of the model

The throughput expression (4) is analyzed with respect to the free variables, including the speed of the host (λ), the potential factor (s), the speedup function parameters of the programs (D_{i1} and D_{i2}), and the partition size (p). For purposes of comparison, a baseline model is assumed. Unless otherwise stated, $\lambda = 2.0$ and $s = 0.9$. In the baseline model, programs 1 and 2 are assumed to have speedup functions (2) and (3), respectively.

λ — the speed of the host

We are interested in the effect upon performance as λ is perturbed. The sensitivity of throughput with respect to the partition size for various values of λ is shown in Figure 4a. Figure 4b shows the sensitivity of throughput with respect to λ when the partition size is set to its optimal value, p^*. These values of p^* can be derived analytically, as is shown in the p sensitivity section below.

As expected, throughput (X) generally increases concavely as the speed of the host (λ) increases. When the host is slow and is the system bottleneck (i.e., its utilization, given by the sum of probabilities $P(H,H) + P(P_1,H) + P(H,P_2)$, is high relative to that of the multiprocessor), throughput increases proportionally as λ increases. Sensitivity to changes in λ decreases when the host is relatively fast and its utilization low.

For certain values of the model parameters, throughput decreases as the speed of the host increases. For example, if the potential factor s is reduced to 0.15, throughput increases with λ until $\lambda = 1.324$. Increasing host speed beyond $\lambda = 1.324$ degrades performance. A small value of s results when competition among programs at the multiprocessor seriously degrades

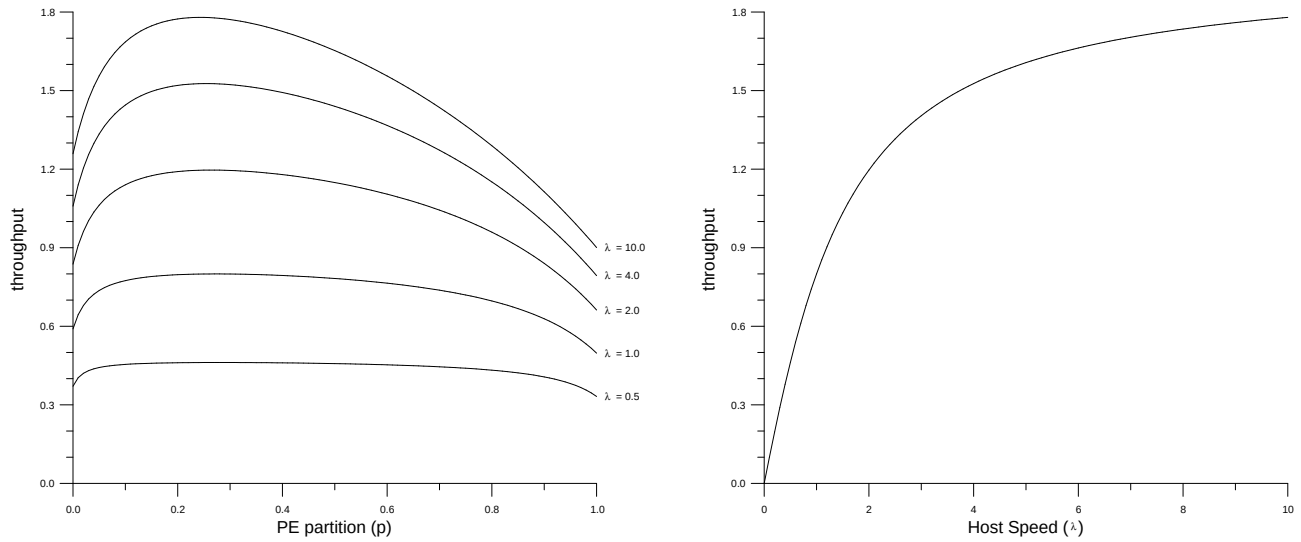

Figure 4: *Sensitivity of the model with respect to host speed. Speedup functions of programs 1 and 2 are specified by equations (2) and (3), respectively. The potential factor $s = 0.9$. In (b), the optimal partition size is used.*

performance. Severe degradation might result, for example, from a "hot spot" in a multistage switching network connecting a set of PEs to a shared memory [14]. Increasing host speed causes programs to spend relatively more time at the multiprocessor, thus increasing multiprocessor contention and decreasing overall throughput. This phenomenon is not unlike thrashing, where increased utilization can result in dramatic system degradation. Slowing down the host reduces contention at the multiprocessor, thus improving overall performance.

Figure 4a indicates that partition size p can affect performance significantly. For example, in the base case with $\lambda = 10.0$, performance with an optimal partition is 39% better than performance with the worst partition. However, when the host is slow, performance is less sensitive to changes in the partition size. The optimal partition size decreases as a function of λ, but only slightly. For the base case, p^* ranges from 0.2899 when $\lambda = 0$, to 0.2303 when $\lambda = \infty$.

s — the potential factor

The potential factor s models contention experienced by programs that are simultaneously active at the multiprocessor.

As expected, throughput degrades as the potential factor decreases and the overhead increases. Interestingly, the optimal partition size is unaffected by s, as is shown in the p sensitivity section below.

D_{i1} and D_{i2} — the speedup functions

Sensitivity analysis of the speedup function parameters could be performed in a number of different ways. We take an intuitive approach in which speedup functions change in a natural way. If a change in the efficiency of a program is brought about, for example, by changes in the volume of interprocess communication, execution times for small numbers of PEs will be relatively unaffected, but execution times for large numbers of PEs may change

significantly. Therefore, when varying a speedup function, we fix the function's derivative at zero, thus allowing only small changes in the function values for small numbers of PEs. Holding constant the value of the derivative at zero corresponds to fixing the value of D_{i2}.

Sensitivity analysis of the speedup function parameters yields expected results. Throughput increases as program efficiency increases. The optimal partition size increases as a program becomes more efficient. Thus, a more efficient program should be allocated more PEs.

p — the partition size

Unlike the other model parameters, throughput as a function of p is not monotonic but has a single global maximum.

The optimal partition size, p^*, calculated from (4) is

$$p^* = \frac{\begin{array}{c} D_{12}(\lambda(D_{11}D_{22} + D_{21}^2 + D_{21}D_{22}) \\ +2(D_{21} + D_{22})) \end{array} \pm \sqrt{\begin{array}{c} D_{12}D_{22}(\lambda(D_{11}D_{21} + D_{11}D_{22} + D_{12}D_{21}) + 2(D_{11} + D_{12})) \\ \times(\lambda(D_{11}D_{21} + D_{11}D_{22} + D_{12}D_{21}) + 2(D_{21} + D_{22})) \end{array}}}{\lambda(D_{12}D_{21}^2 - D_{11}^2D_{22}) + 2(D_{12}D_{21} - D_{11}D_{22})}. \tag{5}$$

3.2 Comparison with other partition scheduling schemes

The model of Section 2 assumes that each program is assigned to its own private partition, a scheduling strategy we call *private partition scheduling*. Two alternative scheduling strategies are examined and compared with the private partition scheduling strategy. *Largest available partition scheduling* assigns an arriving parallel program to the largest unused partition. *No partitioning* requires each arriving program to wait until the multiprocessor is completely free before beginning execution.

3.2.1 Largest Available Partition Scheduling

In this variation of the model, it is assumed that a program arriving at the multiprocessor is assigned to the largest available partition. Without loss of generality, it can be assumed that the partitions are ordered so that partition i is smaller than partition $i + 1$. Thus, assigning a program to the largest available partition is equivalent to assigning the program to the highest numbered available partition. It is assumed that when a program is assigned to a partition, it remains there until its execution is completed. If a larger partition becomes available during execution, the program does not switch partitions.

In the model of Section 2, states (P_2, H), (H, P_1), and (P_2, P_1) were not possible. Under the largest available partition scheduling scheme, however, these states can occur. The Markov diagram for the state space of the largest available partition scheduling scheme is shown in Figure 5. The partition fractions for P_1 and P_2 are denoted by p and q, respectively, where $p = p_1$ and $q = p_2$ $(p + q = 1)$.

The steady state solution to the global balance equations yields an expression for throughput, corresponding to equation (4). Let the speedup function terms be denoted by

$$a \equiv \frac{p}{D_{11}p + D_{12}}, \quad b \equiv \frac{q}{D_{11}q + D_{12}}, \quad c \equiv \frac{p}{D_{21}p + D_{22}}, \quad d \equiv \frac{q}{D_{21}q + D_{22}},$$

and for notational convenience let

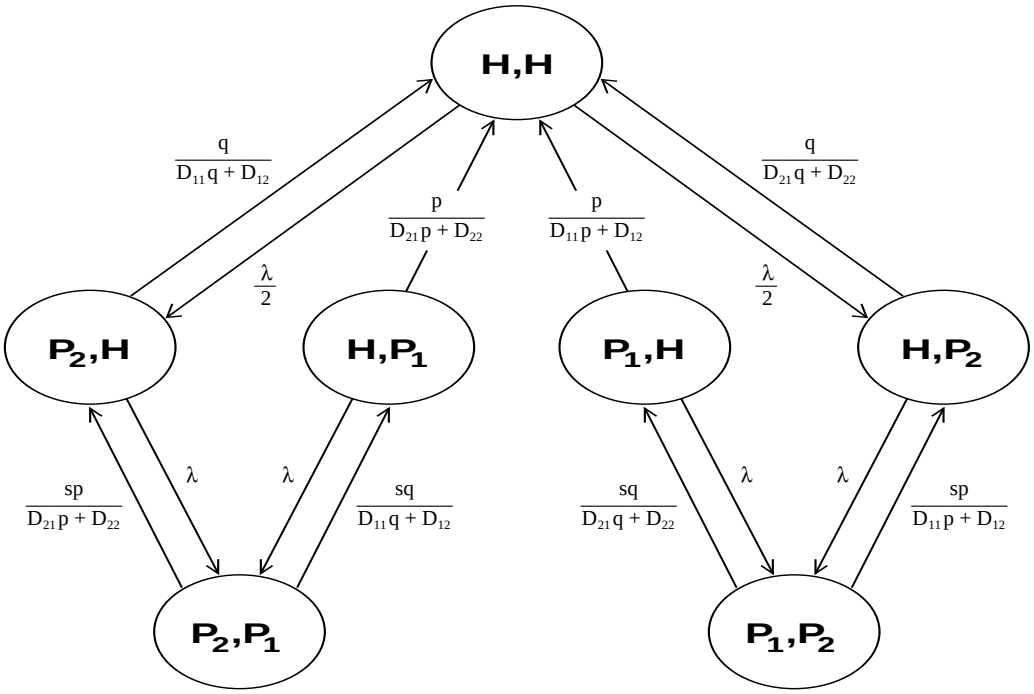

Figure 5: *Markov diagram for largest available partition scheduling.*

$$t_1 \equiv \left(\frac{sb}{\lambda} + \frac{sc}{\lambda} - \frac{sb}{\lambda + c}\right), \quad t_2 \equiv (\lambda + b)t_1 - sc,$$

$$t_3 \equiv \left(\frac{sa}{\lambda} + \frac{sd}{\lambda} - \frac{sd}{\lambda + a}\right), \quad t_4 \equiv (\lambda + d)t_3 - sa.$$

The steady state solution to the global balance equations is

$$\begin{aligned}
P(H,H) &= \left(\frac{1}{G}\right)\left(\frac{2}{\lambda}\right) \\
P(H,P_1) &= \left(\frac{1}{G}\right)\left(\frac{sb}{(\lambda + c)t_2}\right) \\
P(P_1,H) &= \left(\frac{1}{G}\right)\left(\frac{sd}{(\lambda + a)t_4}\right) \\
P(H,P_2) &= \left(\frac{1}{G}\right)\left(\frac{t_3}{t_4}\right) \\
P(P_2,H) &= \left(\frac{1}{G}\right)\left(\frac{t_1}{t_2}\right) \\
P(P_1,P_2) &= \left(\frac{1}{G}\right)\left(\frac{1}{t_4}\right) \\
P(P_2,P_1) &= \left(\frac{1}{G}\right)\left(\frac{1}{t_2}\right)
\end{aligned}$$

where

$$G = \frac{2}{\lambda} + \frac{1}{t_2} + \frac{1}{t_4} + \frac{t_1}{t_2} + \frac{t_3}{t_4} + \frac{sb}{(\lambda + c)t_2} + \frac{sd}{(\lambda + a)t_4}.$$

Thus, the expression for throughput is

$$X(\lambda, D_{11}, D_{12}, D_{21}, D_{22}, p, s) = \lambda(P(H,H) + P(H,P_1) + P(P_1,H) + P(H,P_2) + P(P_2,H)) \qquad (6)$$

3.2.2 No Partitioning

In the no partitioning case, a program arriving at the multiprocessor waits until the multiprocessor becomes totally free. The multiprocessor operates in uniprogramming mode on a first-come-first-serve basis.

The Markov diagram for the no partitioning case is shown in Figure 6.

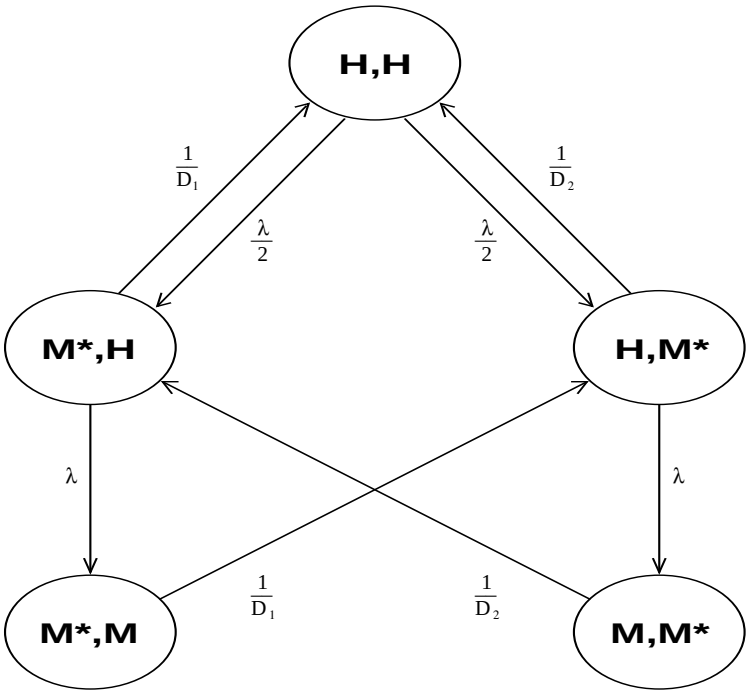

Figure 6: *Markov diagram for no partitioning.*

A minor notation change is needed since there is no partitioning of the PEs. When both programs are at the multiprocessor, one program waits for the other to complete service. State (M^*, H) represents the situation when program 1 is receiving service at the multiprocessor and program 2 is receiving service at the host. State (M^*, M) represents the situation when both programs are at the multiprocessor with program 1 receiving service. Program 2 waits until the multiprocessor becomes free. States (H, M^*) and (M, M^*) are similarly defined. A program executing at the multiprocessor is allocated all of the PEs (i.e., $p = 1$).

The speedup function of a program in the no partitioning case simplifies to $1/(D_{i1} + D_{i2})$. Therefore, $1/D_i$ is used to denote the execution rate.

By solving the global balance equations for the no partitioning case, throughput is found to be

$$X(\lambda, D_1, D_2) = \frac{4D_1D_2\lambda^3 + 3(D_1 + D_2)\lambda^2 + 2\lambda}{2D_1D_2(D_1 + D_2)\lambda^3 + (D_1^2 + D_2^2 + 4D_1D_2)\lambda^2 + 3(D_1 + D_2)\lambda + 2} \tag{7}$$

3.2.3 Comparisons

The sensitivities of (6) and (7) with respect to λ are shown in Figures 7 and 8.

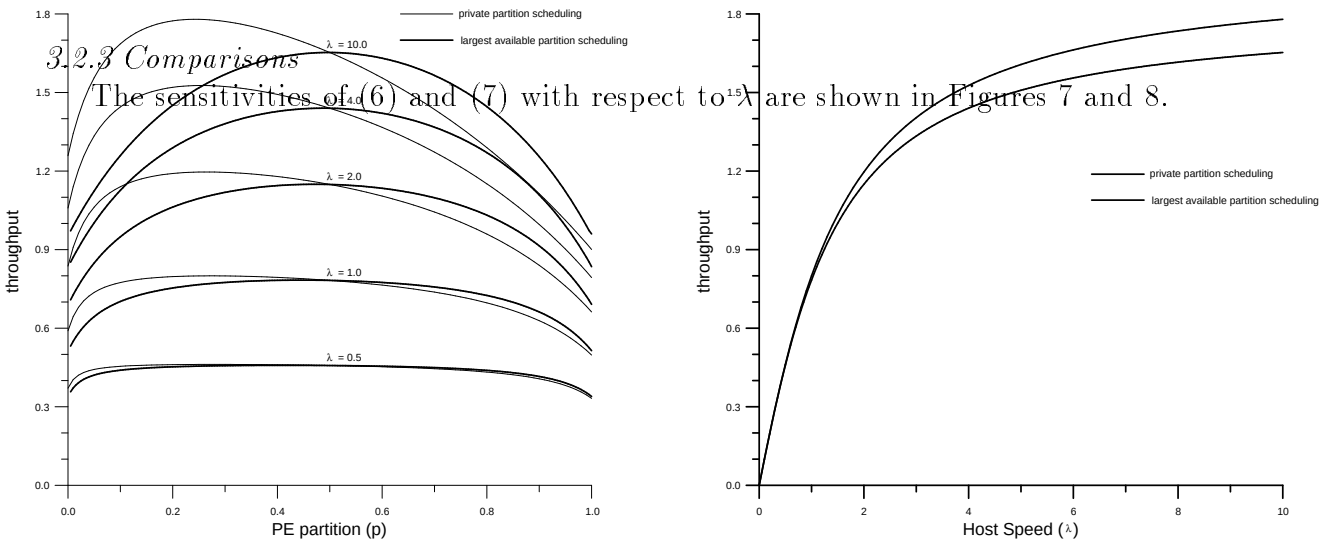

Figure 7: *Sensitivity of largest available partition scheduling with respect to host speed. Speedup functions for programs 1 and 2 are specified by equations (2) and (3), respectively. The potential factor $s = 0.9$. In (b), the optimal partition size is used.*

Analysis of largest available partition scheduling yields a number of results similar to results from analysis of private partition scheduling:

1. Throughput increases as a function of the host speed unless the overhead is high.
2. An optimal partitioning can improve throughput significantly over a poor partitioning.
3. Throughput increases concavely as the potential factor increases and overhead decreases.
4. The optimal partition size is unaffected by the potential factor s. In the baseline case, the optimal partition size is $p^* = 0.26$ for private partition scheduling and $p^* = 0.47$ for largest available partition scheduling.
5. As the efficiency of a program increases, so does throughput.

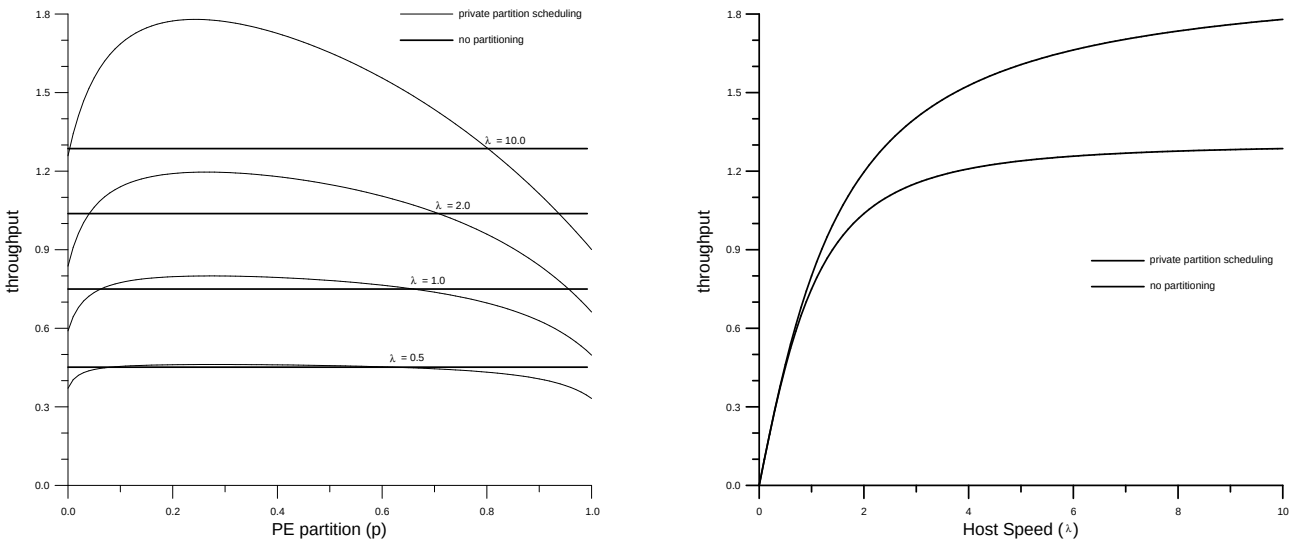

Figure 8: *Sensitivity of no partitioning with respect to host speed. Speedup functions for programs 1 and 2 are specified by equations (2) and (3), respectively. The potential factor s = 0.9. In (b), the optimal partition size is used.*

However, one property of largest available partition scheduling is opposite that of private partition scheduling. The optimal partition size p^* increases as a function of the host speed. Thus, as the probability of both programs being at the multiprocessor increases (i.e., as λ increases), it is better to partition the PEs more equally. Intuitively, equal size partitions minimize the probability that a faster program will be scheduled to execute in a smaller partition. As λ increases from 0.5 to 10.0, p^* increases from 0.41 to 0.50.

As host speed decreases and host utilization increases, private partition scheduling and largest available partition scheduling become more alike with respect to throughput. Private partition scheduling potentially yields better performance than largest available partition scheduling. However, to realize better performance, the program that can best use additional PEs must be allocated the majority of the PEs. A poor partitioning will adversely affect private partition scheduling more than largest available partition scheduling. Thus, if the program that makes the better use of additionally allocated PEs can reliably be assigned to the larger partition, then private partition scheduling is the preferred method. If such an assignment cannot reliably be made, setting $p = 0.5$ and assigning programs to any available partition is recommended.

For the no partitioning approach, throughput increases as a function of the host speed, except when the host is very fast (e.g., $\lambda > 153$ in the base example). This phenomenon is due to the processor-sharing scheduling discipline of the host. Performance could be improved by slowing the host, thus allowing the faster program to pass the slower program.

If the partition sizes are properly set, private partitioning and largest available partitioning outperform no partitioning for sufficiently large values of the potential factor s (e.g., when $s > 0.7$ in the baseline example). Since lowering the potential factor degrades the performance of private partition scheduling and of largest available partition scheduling but has no affect on the no partitioning approach, there is a potential level below which no

partitioning is best. This point can be found by setting expression (7) equal to expression (4) and solving for s.

3.3 Validation

The model of Section 2 has been applied [11] to a 32-node Intel iPSC/2 hypercube multiprocessor with a workload consisting of two parallel wavefront algorithms [6] for solving triangular systems of linear equations. The wavefront algorithms pass messages consisting of segments of the solution vector. Smaller segments increase the potential for parallelism but also increase the number of messages that must be passed. To test the model, program 1, which solves a triangular system of order 1500 with a segment size of 10, and program 2, which solves a triangular system of order 1800 with a segment size of 10, were selected. Their empirically derived speedup curves were fit with functions of form (1), as shown in Figure 9. Experimentally observed throughput values from the wavefront algorithms match

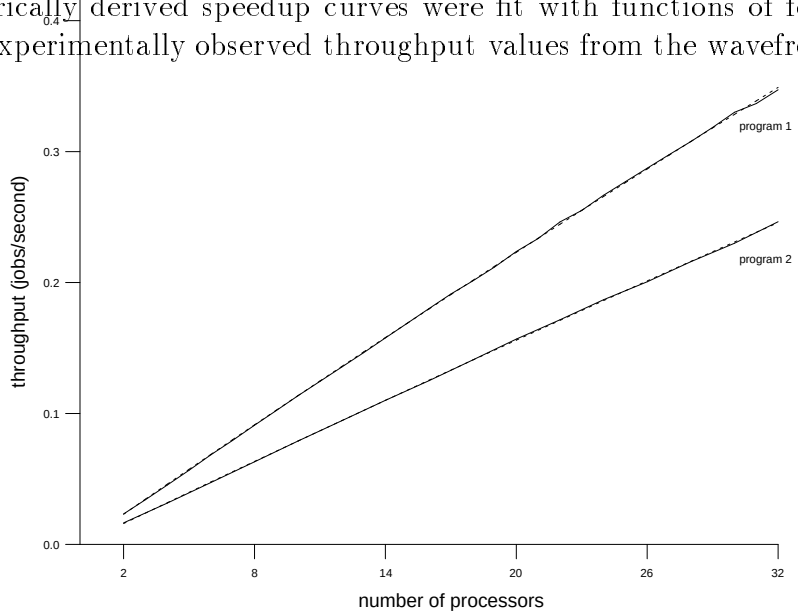

Figure 9: *Speedup curves of the two test programs. The solid curves are empirically determined; dotted curves are least-squares fits of functions of form (1) to the empirical curves.*

predictions of the model quite well, as Figure 10 illustrates. For each experiment in this set, the mean host delay was set to 3.0 seconds.

The model was then further validated by varying the host delay from 0.5 seconds to 10.0 seconds, while keeping the partitions fixed at 16 processors per program. Again, as is illustrated in Figure 11, the model predictions are quite close to the empirical values. Experiments with a number of other test programs produced similarly accurate results.

3.4 Extensions

In future work, we plan to consider a number of variations on the model of Section 2. These variations include

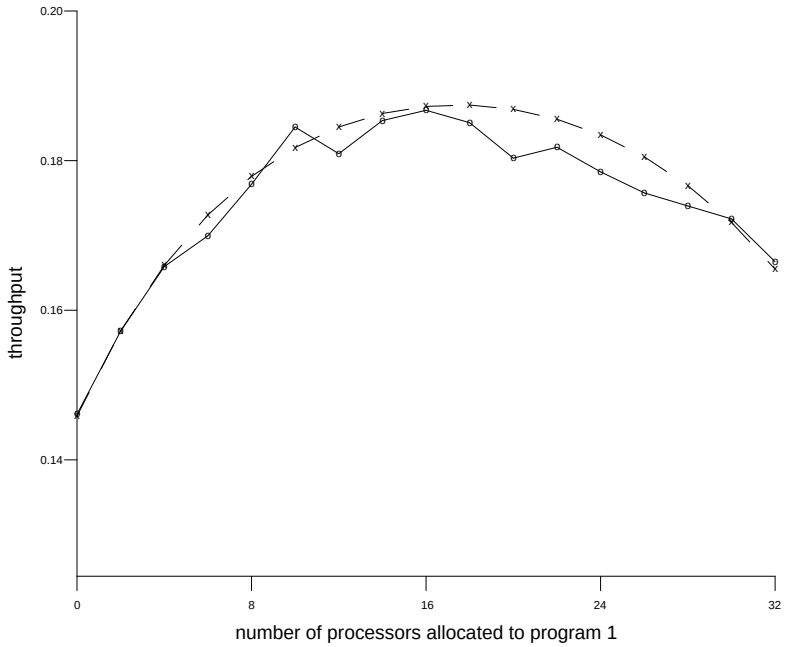

Figure 10: *Throughput as a function of the number of processors allocated to program 1. The solid line connects observed values. The dashed line connects values predicted by the model.*

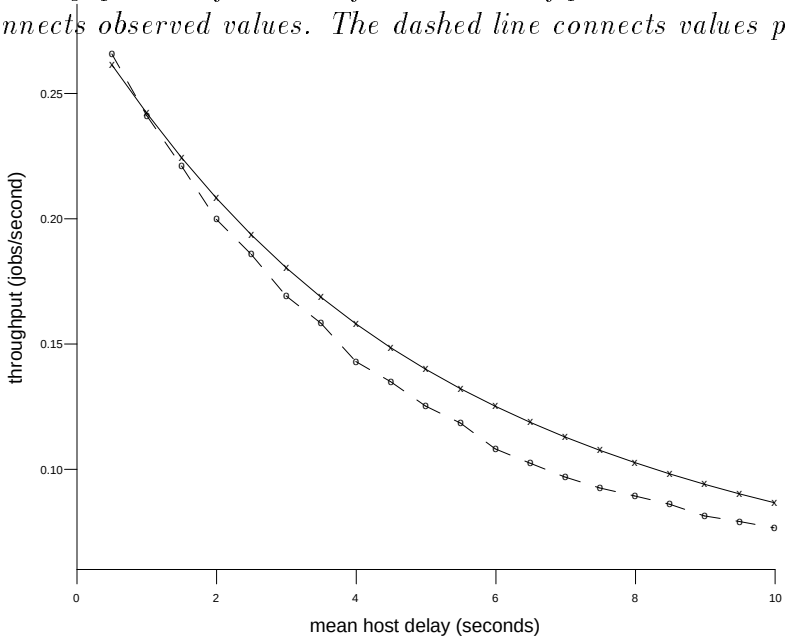

Figure 11: *Throughput as a function of the mean host delay. The solid line connects observed values. The dashed line connects values predicted by the model.*

- changing the form of the speedup functions from that in expression (1),
- allowing dynamic partitioning,
- allowing more partitions,
- changing the potential factor from a constant s to a function of the partition size p and the speedup functions,
- changing the potential factor to penalize certain programs more than others,
- using an open queuing network model, allowing programs to arrive at the multiprocessor according to a specified arrival time distribution, and
- allowing the host to treat different programs differently.

The most difficult extension is apparently that of scale, allowing more partitions and, thus, more programs. As the number of partitions and programs increases, the corresponding Markov diagrams and the resulting throughput equations quickly become intractable. However, in these cases, one can analyze multiple partitions in a piecewise fashion. For example, suppose that programs i and j are two programs from a larger set of programs. From program i's perspective, only the effects of program j and of the rest of the set of programs are seen. The rest of the programs can be viewed as slowing down the host and the multiprocessor by a certain amount. Sensitivities with respect to these parameters can be analyzed. Thus, the optimal partitioning between programs i and j can be found in isolation using a simple Markov analysis. The same can be done for programs i and k, programs j and k, and so on. After all the piecewise comparisons are made, a renormalization of the actual partition sizes can be performed. This decomposition approach to hierarchical modeling [10] can be applied to obtain good approximate solutions.

4. Perspective

The preceding analysis demonstrates that simple models can be used to study a difficult problem. Although the accuracy of the model cannot be verified without direct implementation and experimentation, the model does capture those effects that are most relevant to the problem of assigning PEs to parallel programs. It can be used to discard poor partitioning schemes and to indicate promising partitioning schemes.

We return to the questions of Section 1.

- Under what conditions is partitioning worthwhile? The comparison between no partitioning and private partition scheduling in Section 3.2 indicates that partitioning is worthwhile except when interference from other programs at the multiprocessor is high.
- What is the optimal partition size? In the case of private partition scheduling, an expression for the optimal partition size p^* is given by expression (5).

- How sensitive is performance to changes in the partition? When host utilization is high, partition size is relatively unimportant. However, if host utilization is low, a good choice for partition size can increase throughput significantly, particularly in the private partition scheduling scheme.
- How sensitive is the optimal partition setting to system and workload changes? In the private partition scheduling and largest available partition scheduling schemes, a program is increasingly favored as its efficiency increases. Somewhat surprisingly, the optimal partition size p^* is independent of the potential factor s. As the speed of the host increases, the private partition scheduling scheme works best with partitions that are increasingly diverse in size; the largest available partition scheduling scheme, on the other hand, works best with partitions that are increasingly similar in size.
- How should the host assign programs? Three scheduling strategies were compared, private partition scheduling, largest available partition scheduling, and no partitioning. There are parameter settings for which each scheme is the best and for which each scheme is the worst. No partitioning is best when the interference between competing jobs is significant. Otherwise, if something is known about the relative efficiencies of the programs, private partition scheduling offers the highest potential performance.

These answers are incomplete. However, the modeling technique used to generate these answers provides a simple analytical means to study the partitioning problem.

Acknowledgment

The suggestions of V. Almeida, W. Dulz, G. Geist, U. Herzog, R. Kinicki, A. Krantz, M. Paterok, C. Romine, A. Sieber, K. Thalhofer, and J. Zahorjan significantly improved this paper.

References

[1] G. M. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," *AFIPS Conf. Proc.*, **30** (1967) 483–485.

[2] I. Y. Bucher, "The computational speed of supercomputers," *Proc. of the ACM Sigmetrics Conf.*, (August 1983) 151–165.

[3] T. C. Chen "Overlap and pipeline processing," in **Introduction to Computer Architecture** by H. S. Stone, SRA, 1975.

[4] L. W. Dowdy, "On the partitioning of multiprocessor systems," *Technical Report*, Department of Computer Science, Vanderbilt University, Nashville, TN, 37235, March 1988.

[5] D. L. Eager, J. Zahorjan, and E. D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Trans. on Computers* **38**(3) (March 1989) 408–423.

[6] M. T. Heath and C. H. Romine, "Parallel solution of triangular systems on distributed memory multiprocessors," *SIAM J. Sci. Statit. Comput.* **9**(3) (May 1988) 558–588.

[7] L. Kleinrock, **Queueing Systems Volume I: Theory**, John Wiley & Sons, New York, 1975.

[8] L. Kleinrock, "Distributed systems," *CACM* **28**(11) (November 1985) 1200–1213.

[9] D. J. Kuck, P. P. Budnik, S-C. Chen, D. H. Lawrie, R. A. Towle, R. E. Strebendt, E. W. Davis, Jr., J. Han, P. W. Kraska, and Y. Muraoka, "Measurements of parallelism in ordinary FORTRAN programs," *Computer* **7**(1) (January 1974) 37–46.

[10] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, **Quantitative System Performance**, Prentice-Hall, 1984.

[11] M. R. Leuze, L. W. Dowdy, and K. H. Park, "Multiprogramming a distributed-memory multiprocessor," *Concurrency: Practice and Experience* **1**(1) (September 1989) 19–33.

[12] W. Lin and C-L. Wu, "A distributed resource management mechanism for a partitionable multiprocessor system," *IEEE Trans. on Computers* **37**(2) (February 1988) 201–210.

[13] K. H. Park, **Dynamic Processor Partitioning for Multiprogrammed Multiprocessor Systems**, Ph.D. Dissertation, Dept. of Computer Science, Vanderbilt University, 1990.

[14] G. Pfister and V. Norton, "Hot spot contention and combining in multistage interconnection networks," *Proc. Int. Conf. Parallel Processing* (1985) 790–797.