Multiprogramming a Distributed-Memory Multiprocessor

Michael R. Leuze[†]
Mathematical Sciences Section
Oak Ridge National Laboratory
Oak Ridge, TN 37831-8083

Lawrence W. Dowdy[‡] and Kee Hyun Park
Department of Computer Science
Box 1679 – B
Vanderbilt University
Nashville, TN 37235

ABSTRACT

The development of computing systems with large numbers of processors has been motivated primarily by the need to solve large, complex problems more quickly than is possible with uniprocessor systems. Traditionally, multiprocessor systems have been uniprogrammed, i.e., dedicated to the execution of a single set of related processes, since this approach provides the fastest response for an individual program once it begins execution. However, if the goal of a multiprocessor system is to minimize *average* response time or to maximize throughput, then multiprogramming must be considered.

In this paper, a model of a simple multiprocessor system with a two-program workload is reviewed; the model is then applied to an Intel iPSC/2 hypercube multiprocessor with a workload consisting of parallel wavefront algorithms for solving triangular systems of linear equations. Throughputs predicted by the model are compared with throughputs obtained experimentally from an actual system. The results provide validation for the model and indicate that significant performance improvements for multiprocessor systems are possible through multiprogramming.

[†]Supported by the United States Department of Energy under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

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

Multiprogramming a Distributed-Memory Multiprocessor

1. Introduction

Multiprocessor systems with large numbers of processors are becoming commonplace. This development has been motivated principally by the need to solve large, complex problems that are intractible using conventional uniprocessor systems. Over the past 30 years, computing power (speed, capacity) has doubled approximately every two years. Until recently, these increases were achieved through miniaturization without resorting to changing the basic uniprocessor organization. However, further miniaturization has become increasingly difficult (and expensive) as fundamental physical limitations are approached. Hence, computer architects have begun to increase computing power by coupling large numbers of processors together into a single system, thus allowing them to work simultaneously on a single problem.

It is unlikely that such multiprocessor systems would have been developed without the motivation provided by large, complex problems. Nevertheless, now that multiprocessor systems do exist, it is important to determine how best to use such systems.

The behavior of many parallel programs on a given multiprocessor system can be characterized, at least in part, by speedup curves similar to the one depicted in Figure 1. The speedup curve indicates how effectively a program is able to use additional processors allocated to it. For a typical parallel program, there is a maximum number of processors that can be used effectively. Allocating more than this number of processors may result in reduced throughput. The actual cause of this reduction in performance will vary from system to system, but may be the result of increased contention for the use of shared variables or of an increase in the volume of interprocessor communication.

If the goal of a system manager is to minimize response time for certain high-priority programs, system policies should be designed to allocate such a program the maximum number of processors it can use effectively. If this number exceeds the number of processors available, all available processors should be allocated. If, on the other hand, the system manager's goal is to maximize system throughput, i.e., to maximize the number of jobs completed per unit time, or to minimize *average* response time, a different processor allocation policy is needed. Note, as is illustrated in Figure 1, that a typical speedup curve initially increases rather rapidly but flattens as the point of optimal processor allocation is approached. Thus, the effectiveness with which a parallel program uses an additional processor depends on the number of processors already allocated to the program. Throughput tends to be increased more by assigning an additional processor to a program allocated few processors relative to its optimal number than by assigning it to a program allocated close to its optimal number of processors. Consequently, maximizing multiprocessor system throughput may require that two or more independent programs be executed simultaneously.

It is, therefore, worthwhile considering how a multiprocessor system can be multiprogrammed, that is, how it can be used to manage simultaneously two or more independent

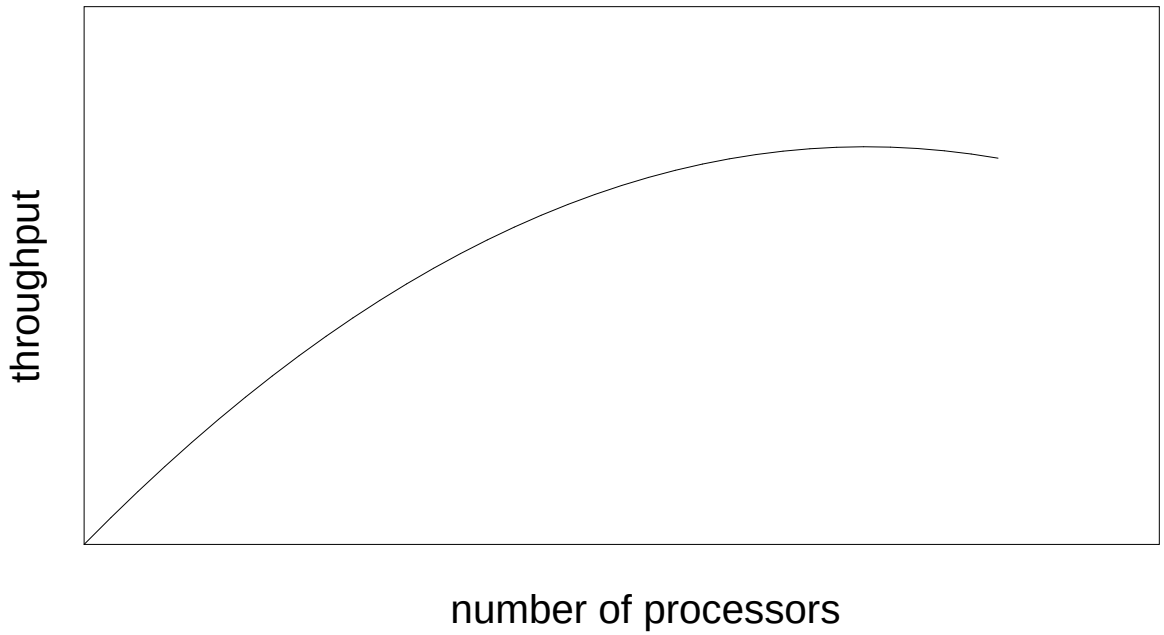

Figure 1: *Typical speedup curve for a parallel program. The horizontal axis indicates the number of processors allocated to the program; the vertical axis indicates throughput per unit time.*

sets of related processes. An important question is how to partition the processors among the active programs. Processor partitioning can be implemented in a number of ways analogous to the partitioning of a computer system's primary memory. The simplest approach is to divide the processors into a number of fixed-size partitions. At any time, a processor partition executes at most a single program, i.e., a single collection of related processes. Each program is designed to run using a specific partition and can be executed only by that partition. A somewhat more restrictive approach may also be taken in which partitions are *private*; i.e., a partition and a program are paired, and as long as the program remains in the system, no other program may use its partition. An alternative approach is to allow programs to execute in any available fixed-size partition. Finally, partitions may be variable. As the processing needs of a workload change, the size and configuration of processor partitions may change in response to these needs. In this paper, we consider modeling systems with two fixed-size, private partitions.

Section 2 contains a description of a simple model of a multiprocessor system with a two-program workload. In section 3, the model is adapted to an Intel iPSC/2 hypercube multiprocessor. Section 4 contains results from numerical experiments performed on the Intel iPSC/2 and compares throughputs obtained empirically to those predicted by the model.

2. The Model

In [2], a high-level model of a simple multiprocessor system whose processing elements (PEs) can be partitioned into two sets is described. The model assumes that program scheduling is done via a separate host processor, and that the parallel workload consists of two programs. The model description uses the following notation:

N—the total number of PEs in the multiprocessor.

n_i $(i = 1, 2)$—the number of PEs assigned to partition P_i. $(n_1 + n_2 = N)$.

p_i $(i = 1, 2)$—the fraction of PEs assigned to partition P_i. $(p_i = n_i/N.\ p_1 + p_2 = 1)$.

(A, B)—the current system state. "A" represents the location of program 1, either at the host (H) or in partition P_1. "B" represents the location of program 2, either at the host (H) or in partition P_2. The set of allowable 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. Service times are assumed to be exponentially distributed.

$S_M(A, B)$—a state-dependent "potential factor." A parallel program assigned to a partition may not operate at its full potential due to contention for shared resources.

X—throughput of the multiprocessor system.

This model can be represented by the closed queuing network model of Figure 2.

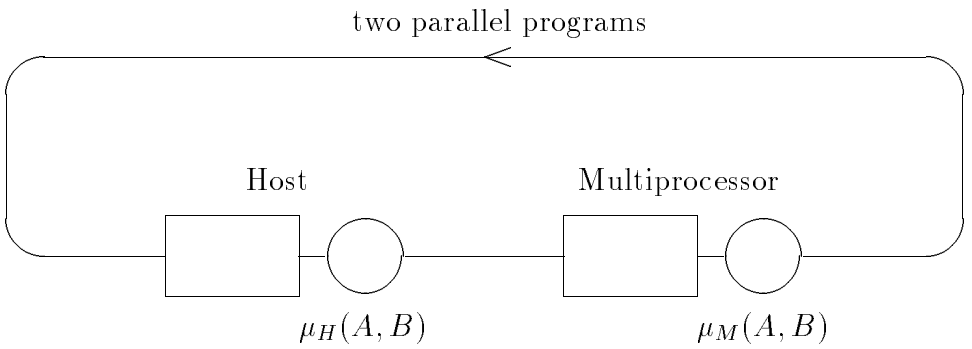

Figure 2: *Closed queuing network model of a multiprocessor system with its parallel workload.*

The host processor is included to model the workload sent to the multiprocessor. Each program experiences a delay at the host to reflect the program's cost of compilation, input/output, and accounting. The host also allows a variable multiprogramming level to be placed on the multiprocessor. Due to the action of the host, the multiprocessor is sometimes

idle, sometimes executing a single program, and sometimes executing multiple programs. By varying the host's speed, different multiprocessor workloads can be modeled.

The following assumptions are made with regard to the model:

$\mu_H(A, B) = \lambda$. Service time at the host is state independent. The host treats both programs identically; i.e., the demands each program places on the host are statistically identical. Processor sharing at the host is assumed.

$\mu_M(P_1, H)$ and $\mu_M(H, P_2)$ are the "execution signatures" of the two parallel programs. Each term represents the processing rate of a program when it alone executes on the multiprocessor. Execution signatures depend not only on a program's characteristics, but also on the specific architecture and on the size of the program's partition. Expressions for execution signatures are suggested below. Execution signatures are analogous to speedup curves (see Figure 1).

$S_M(H, H) = S_M(P_1, H) = S_M(H, P_2) = 1$. When at most one program is executing on the multiprocessor, no performance degradation is experienced, since there is no contention for the shared resources of the multiprocessor.

$S_M(P_1, P_2) = s$. When both programs execute on the multiprocessor, the performance of each is degraded to a constant potential factor s.

$\mu_M(P_1, P_2) = (\mu_M(P_1, H) + \mu_M(H, P_2)) \times S_M(P_1, P_2)$. When both programs are executing on the multiprocessor, the processing rate is the sum of the individual processing rates times the potential factor $S_M(P_1, P_2)$.

Execution signatures of the form

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

have been suggested [2], where D_{i1} and D_{i2} are parameters dependent on characteristics of the specific architecture and of the specific program i. If $D_{i1} = 0$, program i experiences a linear speedup; if $D_{i2} = 0$, the execution rate of program i is constant, independent of the number of processors used.

The model is solved by applying standard techniques [5] to the balance equations derived from the Markov diagram shown in Figure 3.

Throughput as a function of the system parameters is

$$X(\lambda, D_{11}, D_{12}, D_{21}, D_{22}, p_1, p_2, s) =$$

$$\frac{\lambda^2(p_2 s D_{12} + p_1 p_2 s D_{11} + p_1 p_2 s D_{21} + p_1 s D_{22}) + \lambda(2 p_1 p_2 s)}{\left(\begin{array}{l}\lambda^2(p_2 D_{12} D_{21} + p_1 p_2 D_{11} D_{21} + p_1 D_{11} D_{22} + D_{12} D_{22}) + \\ \lambda(p_2 s D_{12} + p_1 p_2 s D_{21} + p_1 p_2 s D_{11} + p_1 s D_{22}) + 2 p_1 p_2 s\end{array}\right)} \tag{2}$$

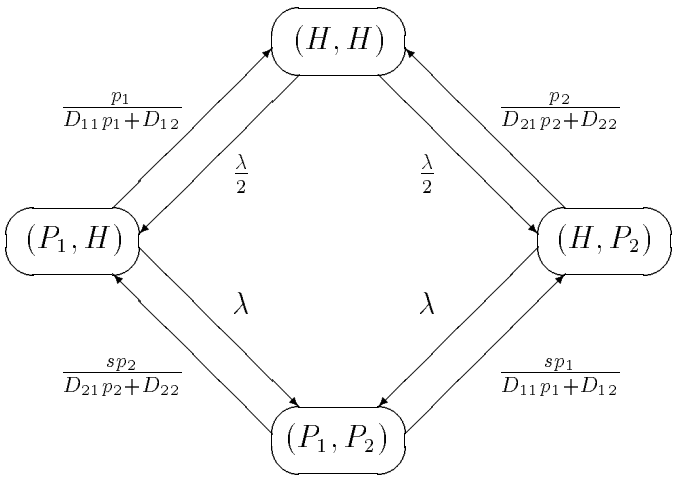

Figure 3: *Markov diagram of the two-program multiprocessor system model. States correspond to locations of the programs; arcs are labeled with the transition flow rates.*

3. Adaptation of the Model

3.1 The multiprocessor system

Experiments were performed on a 32-node Intel iPSC/2 multicomputer [4]. The iPSC/2 is a distributed-memory machine, connected in a binary n-cube configuration. Communication in the iPSC/2 is handled by a separate communications processor on each node, known as a "Direct-Connect^{TM} module." The Direct-Connect module supports two modes of message-passing: short messages (100 bytes or less) are passed using a datagram technique; long messages (more than 100 bytes) are passed using a virtual circuit technique known as "wormhole routing" [1]. Consequently, there is a jump in the cost of transmitting messages at the 100-byte boundary. The Direct-Connect module network requires only slightly longer to pass multi-hop messages than to pass single-hop messages. Consequently, the iPSC/2 appears to the user to be a fully-connected machine.

Adaptation of the model to the iPSC/2 is straightforward. There are only two major considerations: the manner in which the 32 nodes of the iPSC/2 are partitioned and the value used for the potential factor s. Preliminary experiments investigating possible ways to partition the 32 nodes of the iPSC/2 were conducted. The results of these experiments indicate that the critical factor in determining a program's performance is the number of processors allocated to that program; the specific processors assigned to a program have little impact on its behavior, due to the "fully-connected" nature of the iPSC/2. Therefore, for convenience, program 1 is always run in a partition consisting of nodes numbered 0 through $i-1$, and program 2 is always run in a partition consisting of nodes numbered i through 31 ($0 \leq i \leq 32$).

Experiments were also performed to determine the value of the potential factor s. The results of these experiments indicate that a program running in one partition has negligible impact on a program running in another partition. Consequently, s is set to one.

3.2 The parallel workload

The application programs used in these experiments are based on parallel wavefront algorithms [3] for solving triangular systems of linear equations. The wavefront idea produces both a column-oriented vector-sum algorithm and a row-oriented scalar-product algorithm. For these experiments, the row-oriented algorithm was chosen. In this algorithm, messages consist of segments of the solution vector. Smaller segments increase the potential for parallelism but also increase the number of messages that must be passed. A performance trade-off exists, therefore, between potential parallelism and communication volume. Heath and Romine [3] have examined the issue of choosing an optimal segment size. The effect of varying the segment size (σ) is illustrated in Figures 4 and 5.

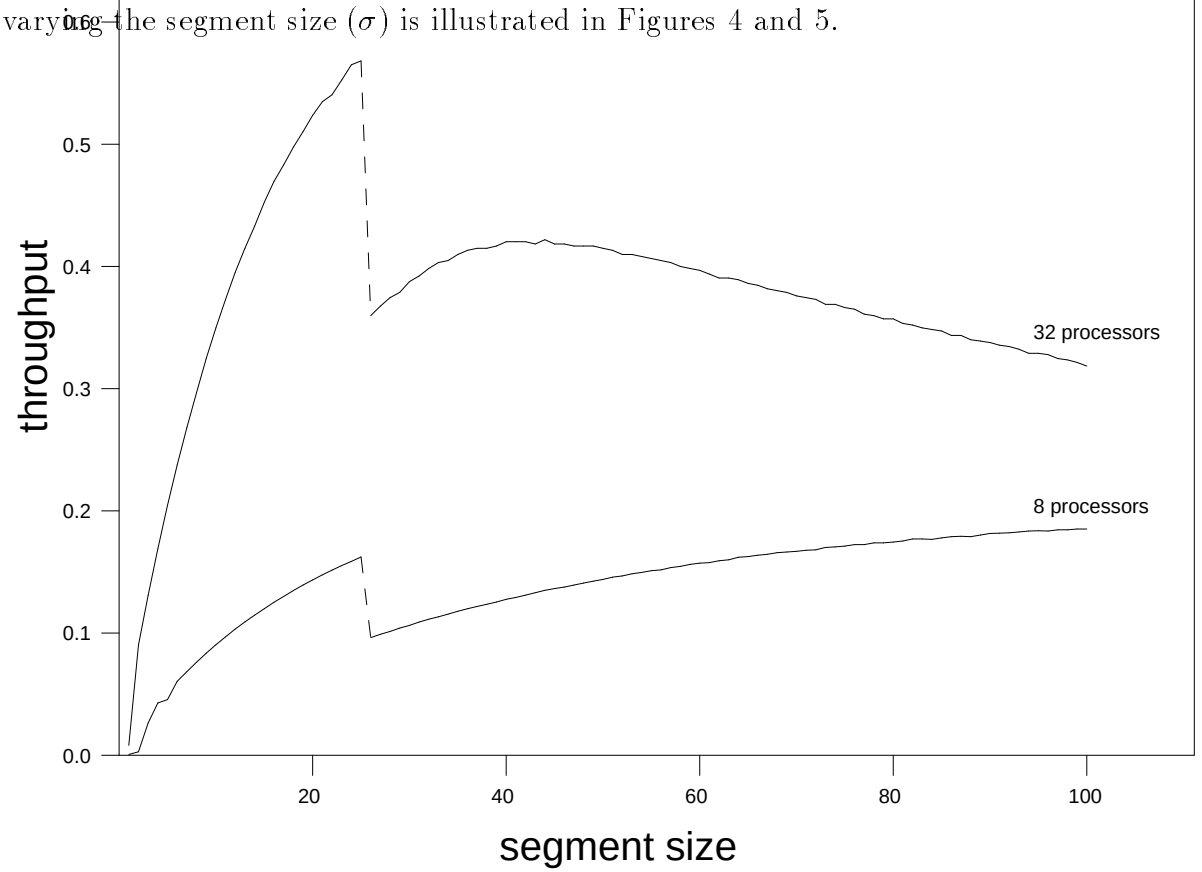

Figure 4: *Throughput as a function of segment size for 8 and for 32 processors solving a triangular system of order 1500 using the row-oriented wavefront algorithm. The dashed lines indicate the point at which the iPSC/2 switches message-passing modes from datagram service to virtual circuit service.*

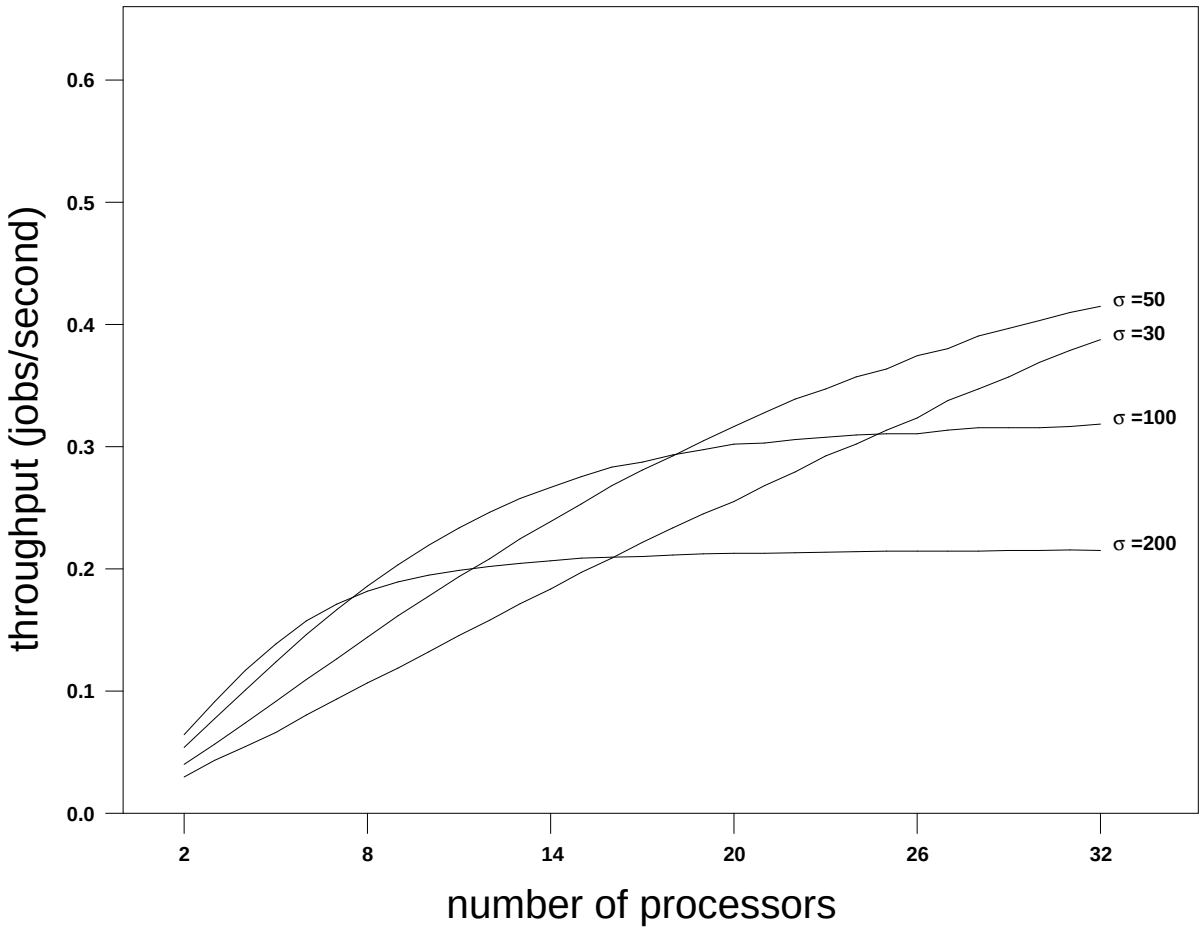

Figure 5: *Execution signatures for the row-oriented wavefront algorithm on the iPSC/2 hypercube with a matrix of order 1500.*

Each program was written to run in two modes, a multiprocessor mode and a host mode. When in multiprocessor mode, a program solves a single triangular system in its private partition of the iPSC/2; when in host mode, a program executes a delay loop on the host machine (see Figure 6). The duration of a delay is randomly chosen from a predetermined probability distribution. The delay loop is used to simulate various host speeds, various amounts of required activity on the host, or various "loadings" placed on the multiprocessor by the host. A single experiment consists of the simultaneous execution of the two test programs for a predetermined period of time, during which each program cycles between its multiprocessor and host modes.

To study empirically the throughput of a partitioned system, two programs based on the row-oriented wavefront algorithm were selected. Program 1 solves a triangular system of order 1500 with a segment size of 10, and program 2 solves a triangular system of order 1800 with a segment size of 100. The empirically determined execution signatures of these two programs are illustrated in Figure 7. Corresponding execution signatures of the form of

```
k = 222500.0 * tmin
  + 222500.0 * (tmax - tmin) * random() ;
j = 0 ;
for (i = 0; i < k; i++)  j = (j+1)%1000 ;
```

Figure 6: *Sample delay loop executed by a program in host mode. The routine* `random()` *generates a uniform random deviate between 0 and 1. The constant* `222500.0` *was empirically determined so that in this example, a delay of from* `tmin` *to* `tmax` *seconds would occur.*

equation (1) were determined by a non-linear least-squares fit. These execution signatures

$$\mu_M(P_1, H) = \frac{p_1}{0.1702p_1 + 2.6947} \quad \text{and} \quad \mu_M(H, P_2) = \frac{p_2}{2.4615p_2 + 1.1839} \tag{3}$$

are also illustrated in Figure 7.

Adapting the model (Section 2) to a specific architecture (Section 3.1) and to a specific workload (Section 3.2) results in parameter values listed in Table 1. The expression for

parameter	*value*
D_{11}	0.1702
D_{12}	2.6947
D_{21}	2.4615
D_{22}	1.1839
s	1.0000

Table 1: *Parameters for the theoretical model.*

throughput as a function of the partition size and host speed becomes

$$X(\lambda, p_1, p_2) = \frac{\lambda^2(2.6317p_1p_2 + 1.1839p_1 + 2.6947p_2) + \lambda(2p_1p_2)}{\left(\begin{array}{c}\lambda^2(0.4189p_1p_2 + 0.2015p_1 + 6.6330p_2 + 3.1903) + \\ \lambda(2.6317p_1p_2 + 1.1839p_1 + 2.6947p_2) + 2p_1p_2s\end{array}\right)} \tag{4}$$

4. Numerical Experiments

Two test cases, chosen to illustrate two sources of increased throughput from multiprogramming, were examined. In Test Case 1, program 2 makes more efficient use of earlier processors than does program 1, while program 1 makes more efficient use of later processors (See Figure 7). Throughput can be increased through multiprogramming by assigning

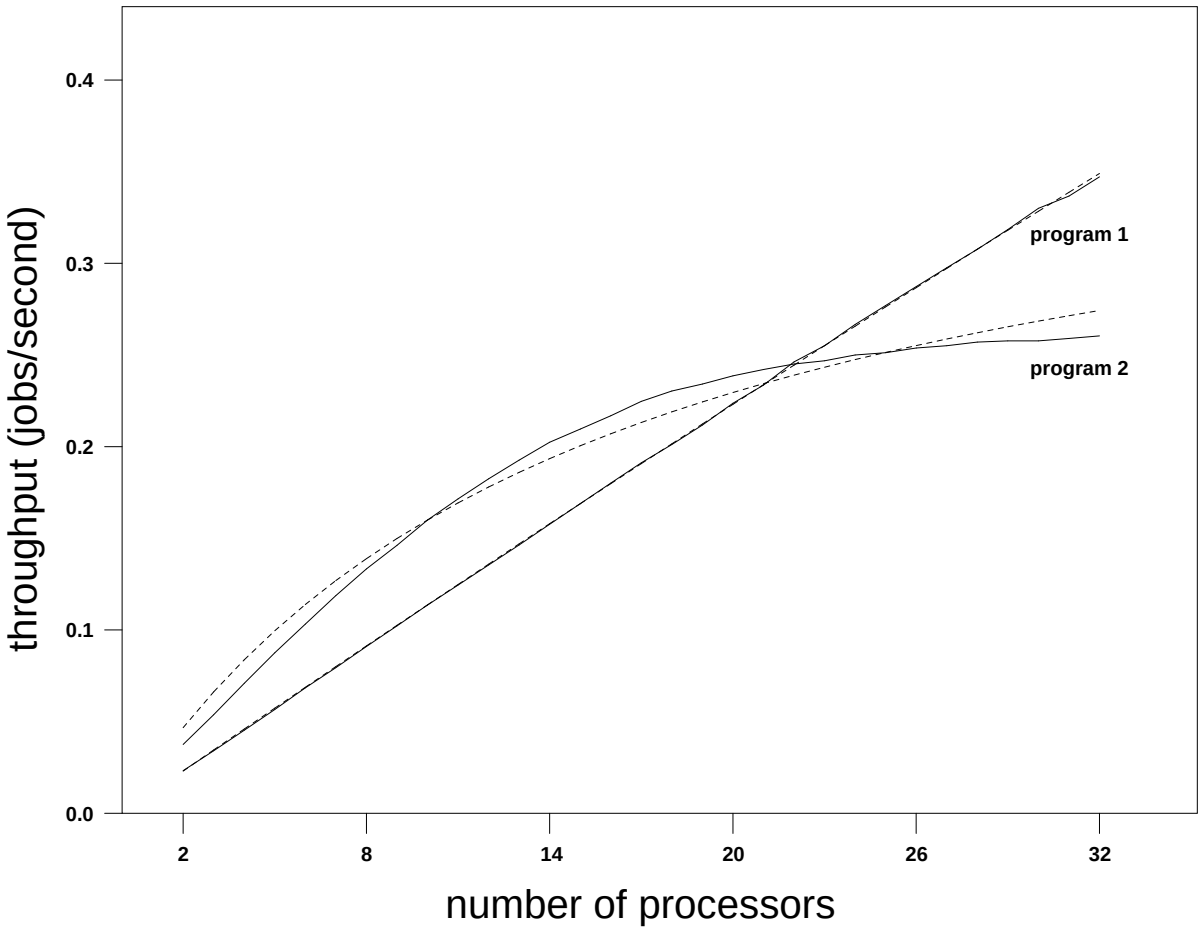

Figure 7: *Execution signatures of the two programs from Test Case 1. The solid curves are empirically determined signatures; dotted curves are least-squares fits of signatures of the form of equation (1) to the empirical signatures.*

processors to programs so as to optimize processor utilization. In Test Case 2, program 1 uses *each* processor it is assigned more efficiently than program 2 uses *any* processor (see Figure 11). It would seem that maximum throughput could be achieved by uniprogramming the multiprocessor, so that program 1 would be allowed to use all processors during its multiprocessor phase. However, program 1 would then be forced on occasion to await the completion of program 2's multiprocessor phase, during which time the multiprocessor would not be used at maximum efficiency. Multiprogramming increases throughput by allowing the more efficient program to use a portion of the multiprocessor any time it is able.

4.1 Test Case 1

The 32 nodes of the iPSC/2 were allocated so that program 1 was executed in partitions of size 0, 2, 4, ..., 32, with program 2 executing in the complementary partitions. Host delays were uniformly distributed between 2 and 4 seconds. The value for the speed of the host (λ) was 0.3268, the reciprocal of the sum of the average time spent in the delay

loop (3 seconds) and a small measurable host–multiprocessor communication time (0.06 seconds). Each experiment was run for 10 minutes. System throughputs were both measured empirically and predicted using equation (4). The empirically determined and theoretically predicted throughputs are illustrated in Figure 8. Throughputs as high as 0.2134 jobs per second were observed. This throughput is 36.5% greater than the throughput for the same job mix in the corresponding uniprogramming environment, 0.1563 jobs per second.

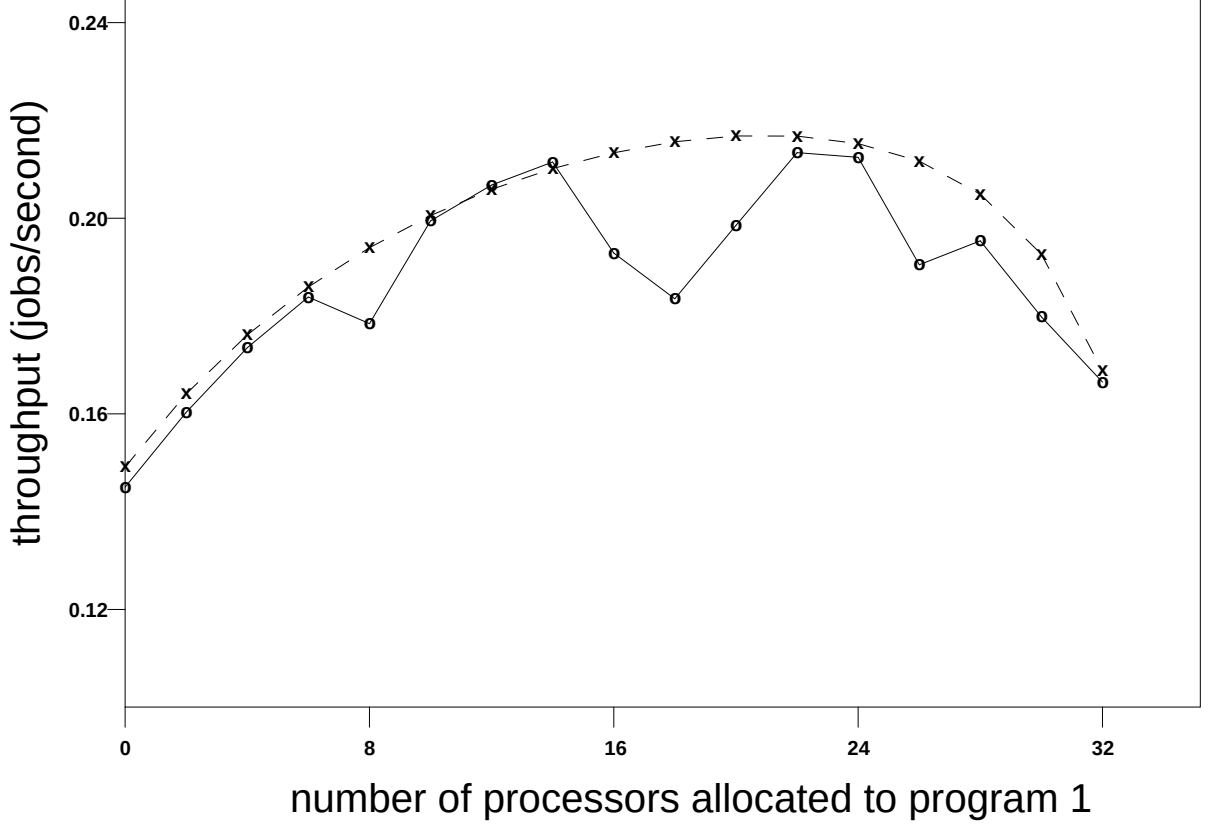

Figure 8: *Throughput for Test Case 1 as a function of the number of processors allocated to program 1. The solid line connects observed values. The dashed line connects values predicted by the theoretical model using the input values of Table 1.*

The experimental and theoretical throughput values match quite well. The greatest discrepancy in throughput values occurs when 18 processors are assigned to program 1 and 14 processors are assigned to program 2. When allocated 18 processors, program 1 requires approximately 4.97 seconds to solve a triangular system; when allocated 14 processors, program 2 requires approximately 4.94 seconds to solve a triangular system. Thus, since each program places the same demands on the host, the average multiprocessor-host cycle times for both programs are nearly identical in length for this particular partitioning. When the program traces for this partitioning were examined, it was discovered that, despite the ran-

dom delays incurred at the host, the programs tended to execute in lock-step, i.e., both programs were together at the host and then both programs were together at the multiprocessor. This tendency toward synchronized behavior resulted in increased contention at the host and in a consequent decrease in throughput.

Other discrepancies between experimental and theoretical throughput values occur at the 8–24 and at the 26–6 partitionings. For these partitionings, the cycle time of one program was approximately twice the cycle time of the other. Again, a tendency toward synchronization was observed, with each multiprocessor phase of the slower program and every second multiprocessor phase of the faster program beginning at approximately the same time. Thus, every second host phase for the faster program was without contention; no host phase of the slower program was ever without contention.

Experiments were conducted to determine whether increasing the variance in the host delay, while maintaining a mean delay of 3.0 seconds, would counteract the tendency toward synchronization. Using a partition of 18 processors for program 1 and 14 processors for program 2, the minimum host delay was varied in 0.5 second intervals from 0.0 seconds to 3.0 seconds. Experimental results are summarized in Figure 9. As the variance in the host delay increases, the behavior of the two programs becomes more asynchronous. Consequently, contention at the host decreases, and throughput increases. With a host delay between 0 and 6 seconds, the observed throughput is less than 4% below the value predicted by the model.

Other experiments were performed to determine whether the observed synchronization phenomenon was due to the duration of earlier experiments. A number of 100–minute experiments, using a variety of partitionings and host–delay variances, were conducted. In no instance did the observed throughput from a 100–minute experiment differ by more than 0.73% from the observed throughput from the corresponding 10–minute experiment. Thus, the synchronization phenomenon is repeatable and is not an artifact of the duration of the experiments performed. Rather, the tendency toward synchronization appears to result from specific characteristics of the iPSC/2 system.

Experiments in which the speed of the host was changed were performed to determine the sensitivity of the model to changes in hardware. Average delay at the host ($1/\lambda$) was varied from 0.5 to 10.0 seconds in increments of 0.5 seconds. All host-speed sensitivity experiments were conducted with 24 processors assigned to program 1 and 8 processors assigned to program 2. The empirically determined and theoretically predicted throughputs are illustrated in Figure 10. There is excellent agreement between experimental and model throughput values.

4.2 Test Case 2

The sensitivity of the model to changes in software was examined by experimentation on a test case qualitatively different from Test Case 1. Unlike Test Case 1, Test Case 2 involves programs with execution signatures that do not cross (cf. Figures 7 and 11). In Test Case 2, program 1 solves a triangular system of order 1500 with a segment size of 10, and program 2 solves a triangular system of order 1800 with a segment size of 10. The empirically

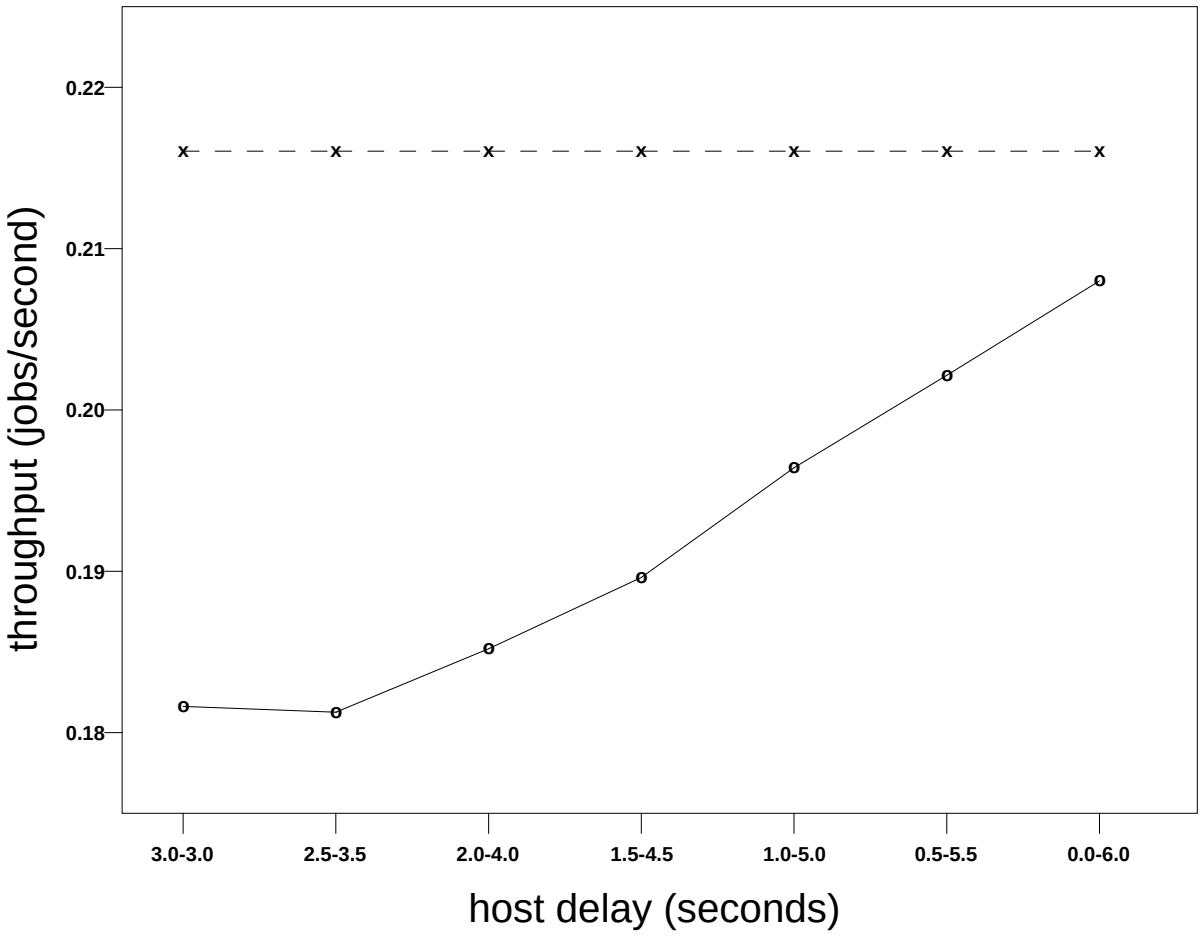

Figure 9: *Throughput for Test Case 1 as a function of variation in host delay. Program 1 was allocated 18 processors and program 2 was allocated 14 processors. Mean delay at the host was maintained at a constant 3.0 seconds. The dashed line represents throughput predicted by the theoretical model.*

determined execution signatures of these two programs are illustrated in Figure 11.

As in Test Case 1, the 32 nodes of the iPSC/2 were allocated so that program 1 was executed in partitions of size 0, 2, 4, ..., 32, with program 2 executing in the complementary partitions. However, in Test Case 2, host delays were uniformly distributed between 0 and 6 seconds in order to minimize the effects of the synchronization phenomenon. Again, the experimental and theoretical throughput values, as illustrated in Figure 12, match quite well. In Test Case 2, the synchronization phenomenon observed in Test Case 1 is not apparent, due perhaps to the increased variance in the host delay.

5. Summary

Although the development of most multiprocessor systems has been motivated by the need to execute single programs quickly, once an actual multiprocessor is in operation, sys-

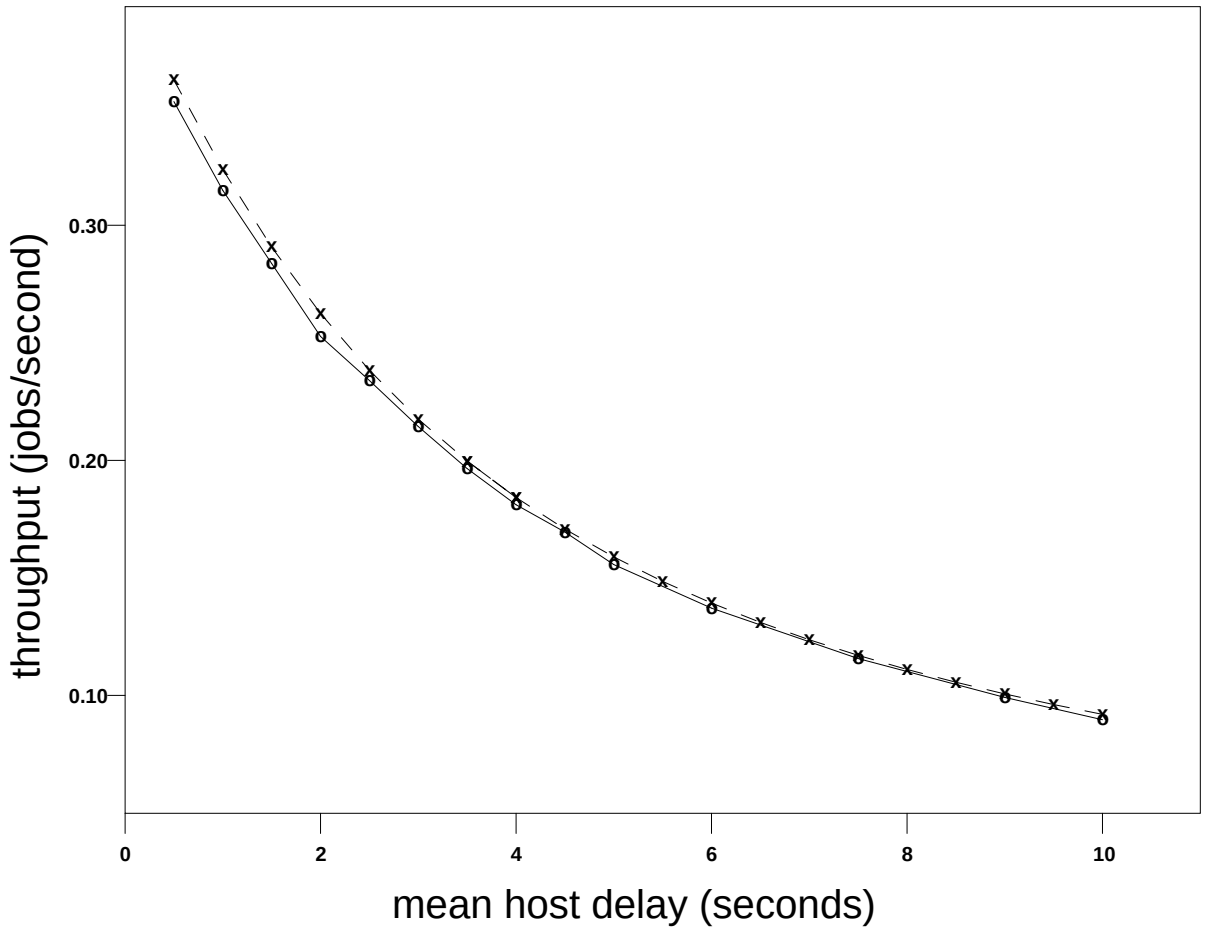

Figure 10: *Throughput for Test Case 1 as a function of the mean delay at the host. The solid line connects observed values. The dashed line connects values predicted by the theoretical model using the input values of Table 1.*

tem goals may change. For example, it may be important to complete the execution of a diverse workload as efficiently as possible, minimizing *average* response time or maximizing throughput. Pursuit of this goal leads to consideration of multiprogramming the multiprocessor system, which leads in turn to consideration of issues related to the partitioning of processors among the set of active programs.

In this paper, the adaptation of a model of a multiprocessor system with two fixed-size private partitions and a two-program workload to an actual system and an actual workload has been described. Validation experiments show a good match between theoretically predicted and empirically observed values. It is demonstrated that multiprogramming a distributed-memory multiprocessor can substantially improve performance. For the example selected, multiprocessor throughputs under multiprogramming that were as high as 0.2134 jobs per second were observed. For the same job mix in the corresponding uniprogramming environment, throughput would be 0.1563 jobs per second. Thus, an increase in throughput of 36.5% due to multiprogramming is observed.

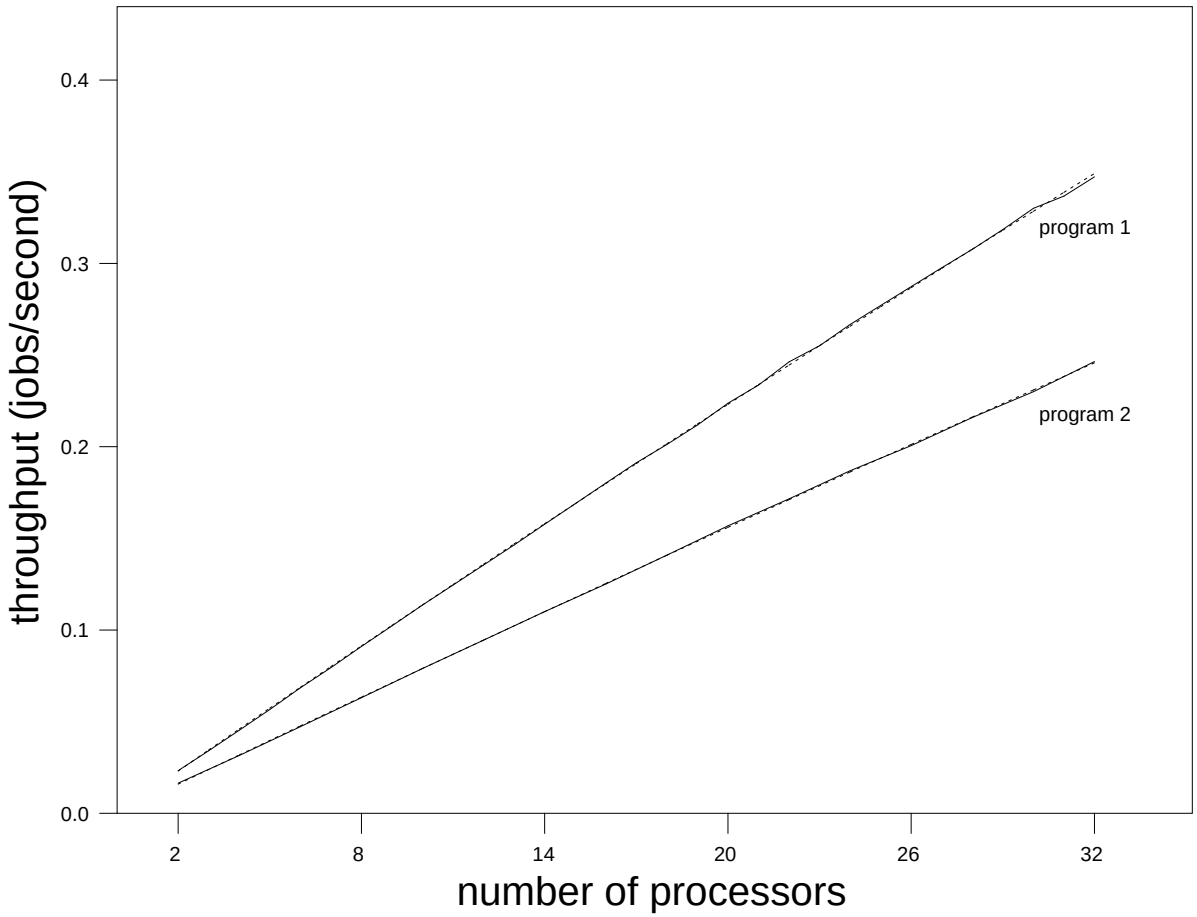

Figure 11: *Execution signatures of the two programs from Test Case 2. The solid curves are empirically determined signatures; dotted curves are least-squares fits of signatures of the form of equation (1) to the empirical signatures.*

The model presented here is a framework for simple analytic models of multiprocessor systems. The model is general enough to represent the primary components of multiprocessor systems, including the arrival process of parallel programs (via the host), program speedup characteristics (via execution signatures), overhead due to contention for shared resources (via the model parameter s), and the allocation of processors to programs (via the model parameter p). The model has been successfully applied to an Intel iPSC/2 hypercube. Preliminary results of applying the model to a 14–node transputer system are also encouraging.

A number of extensions to this work are possible. The model of Section 2 can be generalized to account for more partitions and more programs. Increasing the number of partitions and programs will necessitate the use of heuristics in solving the model, since standard Markov analysis quickly becomes intractable. Other partitioning schemes, such as dynamic partitioning, can be incorportated into the model. The parallel workload can also be generalized. Different forms for the execution signatures can be considered, and the effects of dif-

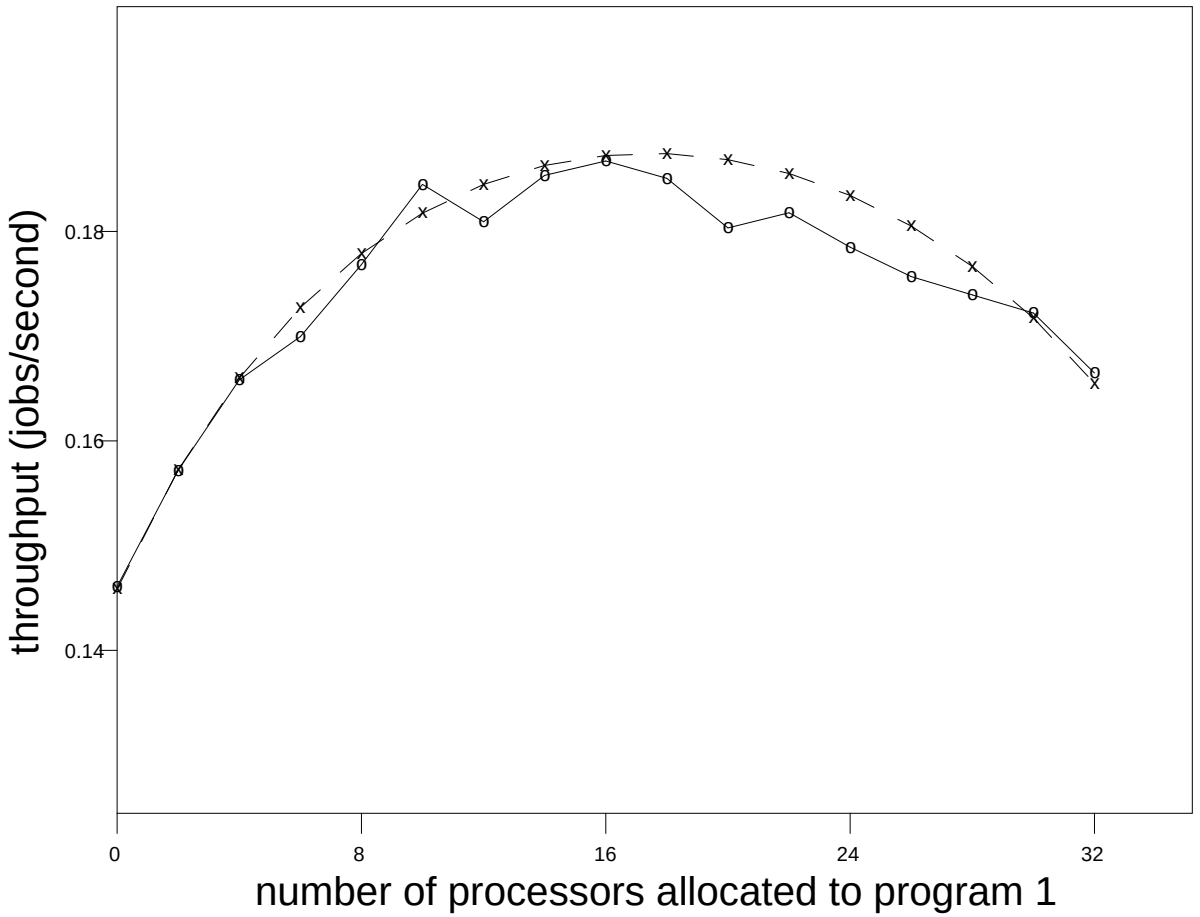

Figure 12: *Throughput for Test Case 2 as a function of the number of processors allocated to program 1. The solid line connects observed values. The dashed line connects values predicted by the theoretical model.*

ferent processing requirements at the host can be explored. For example, host requirements might vary from program to program, and the number of programs in the multiprogramming set may change as new programs enter the system and old programs complete and exit. All of these generalizations of the model and of the workload should be validated experimentally. Because of the performance improvements that may be achieved through multiprogramming of multiprocessor systems, research in this area is important.

Acknowledgements

The authors wish to express their gratitude to Chuck Romine and to Pat Worley for helpful discussions. Their efforts have significantly improved the quality of this work.

M. R. Leuze is supported by the United States Department of Energy under contract DE–AC05–84OR21400 with Martin Marietta Energy Systems, Inc.

L. W. Dowdy is supported in part by the Alexander von Humboldt Foundation while on

leave to the University of Erlangen-Nürnberg, Erlangen, FRG.

References

[1] William J. Dally and Charles L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, *IEEE Trans. on Comput.* **C-36**(5), May 1987, 547–553.

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

[3] Michael T. Heath and Charles H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Statist. Comput.* **9**(3), May 1988, 558–588.

[4] iPSC/2 User's Guide, Intel Scientific Computers, Beaverton, Oregon (1988).

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