Petri Net Model of a Dynamically Partitioned Multiprocessor System

Manish Madhukar †
Computer Science Dept.
Vanderbilt University
Nashville, TN 37235

Michael Leuze †
Mathematical Sciences Section
Oak Ridge National Lab.
Oak Ridge, TN 37831

Larry Dowdy †
Computer Science Dept.
Vanderbilt University
Nashville, TN 37235

Abstract

A multiprocessor system can be subdivided into partitions of processors, each of which can be dedicated to the execution of a parallel program. The partitioning of the system can be done statically at system configuration time, adaptively prior to the execution time, or dynamically during execution time.

Since, in a dynamically partitioned multiprocessor system, partitioning can occur anytime during the execution of a program, designing an analytical model for such a system is a difficult task. In this paper a Petri net model of a dynamically partitioned multiprocessor system is presented. The workload consists of parallel programs which are characterized by their execution signatures. Repartitioning overhead is an important parameter and is modeled explicitly.

The model is used to perform a series of sensitivity analysis experiments which give insight into the behavior of such systems. Several dynamic processor allocation policies have been implemented. Equal Slope and Shortest Job First Preemptive Resume achieve the best performance of the policies considered.

1 Introduction

Often the number of processors available in a multiprocessor exceeds the needs of a single parallel program. It is possible to get better performance by having more than one program execute simultaneously. The multiprocessor can be divided into smaller partitions, each of which can be dedicated to one of the jobs.

The partitioning can be done statically, adaptively, or dynamically. In static partitioning, the partitioning is done at system configuration time and it remains fixed [4], [12]. The partitions are allocated to the jobs as they arrive. For example, a multiprocessor with 64 processors can be divided into two partitions with 16 processors, and one partition with 32 processors. A job arriving at the multiprocessor will execute in one of the three partitions if there is one available, otherwise it will wait for a partition to become free.

In adaptive partitioning, the partitioning is done just prior to the execution of the jobs, and the number and sizes of the partitions may depend on the number of jobs present [11], [13]. A job may receive a large processor allotment if there are few other jobs waiting in the queue, or it may receive a small allotment if there are several jobs waiting. In both static and adaptive partitioning, once a partition is allocated to a job, it is not released until the job terminates. In dynamic partitioning, the partitions depend on the current state of the system. The processors are reallocated every time: 1) a new job arrives and needs to be scheduled, 2) an executing job terminates, releasing its processors, or 3) the requirement of an executing job changes [3], [5], [9], [10], [13]. A job does not necessarily execute in the same partition during its lifetime.

In this paper, a dynamically partitioned system with non-overlapping partitions is considered. This technique of multiprogramming a multiprocessor is also called dynamic space sharing [3], [9], [13].

In a dynamically partitioned system, partitioning can occur anytime during the execution of a job. This makes the modeling of such systems difficult. Previous efforts at modeling a dynamically partitioned system include [5], [10]. In these studies, a simple system with two jobs is modeled as a closed queuing network. A Markov model for the network, based upon the possible states, is constructed and solved. Since the number of states increases significantly as the multiprogramming level is increased, the model quickly becomes intractable.

The contributions of this paper include the following:

- a Petri net model of a dynamically partitioned multiprocessor system, and
- a sensitivity analysis of the model based on various model parameters.

In the system considered here, repartitioning occurs whenever a new job arrives, or an executing job finishes. For the results presented in this paper, it is assumed that a repartitioning can occur only when all the executing jobs have reached a break point. Break points are defined as points in the execution of a job when repartitioning can be carried out efficiently (*e.g.*, at the beginning of a loop, at an execution phase

†Supported in part under sub-contract 19X-SL131V administered through the Mathematical Sciences Section of Oak Ridge National Laboratory.

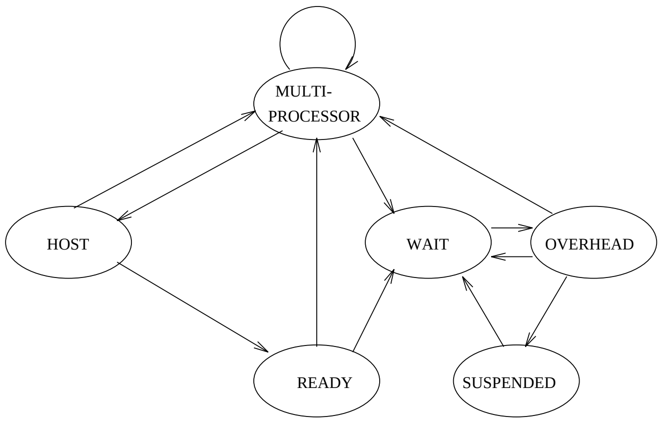


Figure 1: State Diagram

boundary). However, a modified model is also implemented in which this constraint is relaxed. In the modified model, repartitioning can occur as soon as idle processors become available and there are jobs waiting for them. The results from the comparison of the performance predictions of the basic and the modified model are also presented here. The models are constructed using the Stochastic Petri Net Package (SPNP) [2]. An exact steady state analysis of the model is done using SPNP.

The paper is organized as follows. The model is described in Section 2. In Section 3, the processor allocation strategies used in the modeling experiments are described. Section 4 describes the workload. The experiments and results are presented in Section 5. In Section 6, various processor allocation policies are compared. In Section 7, the performance predictions of the basic and the modified models are compared. Section 8 discusses future work.

2 The model

In this section, a description of the program flow model, on which the Petri net model is based, is provided. Following that, the Petri net model is described.

2.1 The program flow model

The execution of a job is divided into a fixed number of sequentially executable segments. The segment boundaries represent break points. A job executes uninterrupted between break points. At the break point the job is a candidate for repartitioning.

At any given instant, a job can be in one of six possible states (see Figure 1):

- *Host* - The job is initially executing at the host (which represents the arrival process). It arrives at the multiprocessor with a fixed rate, configured to use all the processors.
- *Ready* - The job is ready to use the multiprocessor. If all the executing jobs terminate, the job is scheduled and starts execution without undergoing repartitioning. Otherwise, it undergoes repartitioning with the other jobs.
- *Wait* - The job is waiting to undergo repartitioning. Repartitioning occurs when all the executing jobs have reached their next break point.
- *Overhead* - The job is experiencing repartitioning overhead.
- *Multiprocessor* - The job is executing at the multiprocessor in an assigned partition.
- *Suspended* - The job is suspended after having undergone repartitioning. If a priority based processor allocation strategy is used, it is possible for a job to be allocated a partition of size 0. When this happens, the job is moved to the suspended state.

A job is initially at the host. On arrival, it goes directly to the multiprocessor and starts execution if it is the only job present at the system. Otherwise, it is moved to the ready state. Other state transitions will be explained when the Petri net is described in the following section.

2.2 The petri net model

The model is designed as a multi-layered Petri net. There is one layer for each job in the system. Each place in a layer represents a state of a job. Place and state may be used interchangeably in describing the model. Since the number of layers in the model is fixed, the model assumes a fixed multiprogramming level. Figures 2 and 3 together represent the model.

Each job consists of a fixed number of segments, which execute serially. The jobs are characterized by their execution signatures, which specify the rate at which a segment of the job executes as a function of the number of allocated processors.

The segment boundaries represent break points. A job executes uninterrupted between break points. However, when an executing job reaches a break point and there are other jobs waiting to execute, the executing job is required to undergo repartitioning with the waiting jobs. The actual repartitioning is started only when all the executing jobs have reached their break points and are ready to be repartitioned.

The overall model is multi-layered. Consider a typical "basic" layer. There is one basic layer for each job. Moderately thick places represent places occurring only in the basic layer. There is one instance of each such place for each layer in the net. The thick places represent the instances in all layers except the basic layer. The thin places represent places for which there is only one instance in the Petri net.

The arrival process:

A token at the place called 'Host' (P1) indicates that the job is neither executing at the multiprocessor nor contending for processors.

In the initial marking only one transition, 'DepartHost' (T1), is enabled (Figure 2). 'DepartHost' is a timed transition, and the inter-firing time is assumed to be exponentially distributed with a user specified mean. When this transition fires, the token moves to 'NewlyArrived' (P6). This can enable 'GoDirectlyToMulti' (T3) if the multiprocessor is completely idle. Otherwise, the token gets placed in 'Ready' (P2), indicating that the job goes to the ready state, if at

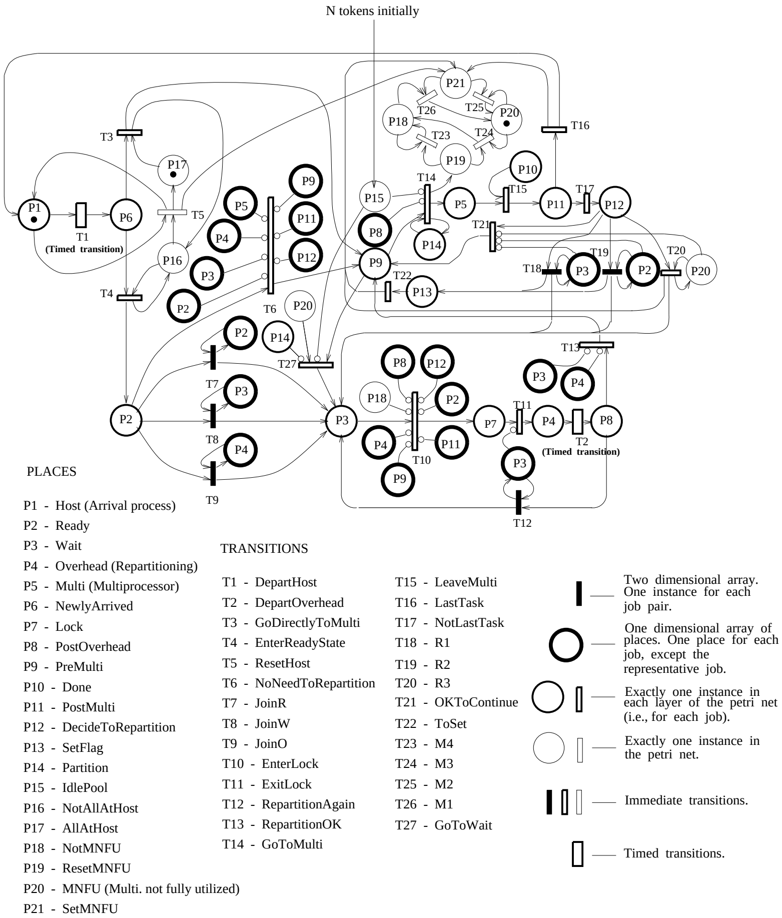


Figure 2: Process Behavior

least one other job is at the multiprocessor. If 'GoDirectlyToMulti' fires, the job goes to 'PreMulti' (P9) state, where it is allocated all the processors. If 'EnterReadyState' fires, a token gets placed in 'Ready' (P2), indicating that the job is ready for execution but has not yet been allocated processors.

The ready state:

From the 'Ready' state, a job can go directly to the 'PreMulti' state if it finds that all the other jobs are at 'Host' (due to the departure of a previously executing job). If there are other jobs at 'Ready' (P2), 'Wait' (P3) (waiting for repartitioning), or 'Overhead' (P4) (undergoing repartition), the token is moved to 'Wait' (P3). This indicates that the job must wait for the next repartitioning cycle before being allocated processors.

The wait state:

The jobs at 'Wait' arrive at this place either from 'Ready' (new arrivals) or from 'DecideToRepartition' (P12) (an executing job suspended for repartitioning). Unless there are no processors available, or there is a job at the 'Ready' state, the token is allowed to proceed to 'Lock' (P7), and then to 'Overhead' (P4). 'DepartOverhead' (T2) is a timed transition representing the repartitioning process. It has an exponentially distributed inter-firing time, with a user defined mean. Firing of 'DepartOverhead' places the token in 'PostOverhead' (P8). The repartitioning process is over when 'RepartitionOK' (T13) fires, placing a token in 'PreMulti' (P9) for every job that undergoes repartitioning.

Job execution:

A token can arrive at 'PreMulti' (P9) from 'NewlyArrived' (P6), 'Ready' (P2) or 'PostOverhead' (P8). For the token to have reached 'PreMulti', there must be exactly one token at 'MNFU' (P20) and one or more tokens at 'IdlePool' (P15). Once a job reaches the 'PreMulti' stage, it becomes eligible to receive processors. Initially, the processors are assumed to be held at 'IdlePool'. There is a token for every processor at 'IdlePool'. This enables the transition 'AllocatePEs' (Figure 3). 'AllocatePEs' remains enabled as long as the number of tokens in 'Partition' is less than the number of processors to which the job is entitled (determined using the processor allocation table). This controls the number of tokens that get placed in each job's 'Partition' (Figure 3). When all the proces-

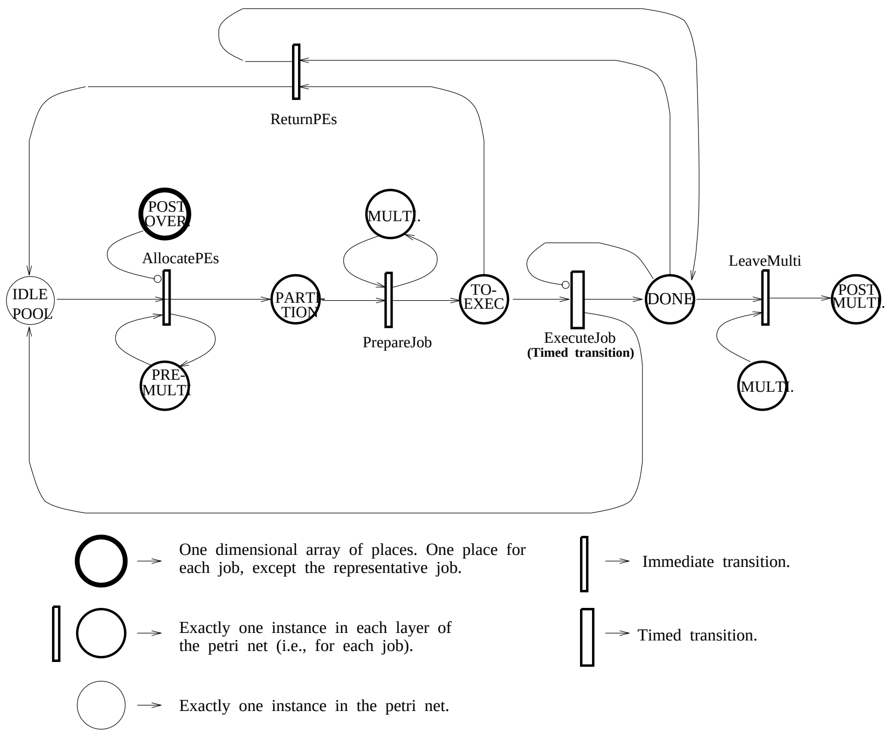


Figure 3: Allocation of Processing Elements

sors are allocated and the 'IdlePool' is empty, the job is allowed to proceed to 'Multi' (P5) (Figure 2).

'ExecuteJob' (Figure 3) is a timed transition with an exponentially distributed firing rate. The mean of the distribution depends on the job's execution signature and the number of tokens in 'ToExec', which represents the number of processors allocated to the job. Firing of 'ExecuteJob' represents completion of a segment of the job. After execution, the processors are returned to the 'IdlePool'.

When the job reaches the 'PostMulti' (P11) state (after execution), the transitions 'LastTask' (T16) (last segment) and 'NotLastTask' (T17) (not the last segment) are enabled. Assuming that there are S segments, 'LastTask' will fire with a probability of $1/S$ and 'NotLastTask' will fire with a probability of $1-(1/S)$. If 'LastTask' fires, it represents the termination of the job. Otherwise, the job remains at the multiprocessor. If there are any other jobs waiting for processors when the execution of a segment completes, the job is moved to 'Wait' and is forced to undergo repartitioning. If there are no jobs waiting for processors, the job is allowed to begin execution of its next segment without undergoing repartitioning.

The execution rate:

The firing rate for 'ExecuteJob' (Figure 3) is exponentially distributed with a mean that depends on the number of processors allocated to the job. The processor allocation policy is used to build a processor allocation table, which is then used to determine the allocations for each job. This table is built once, at the beginning of the model's execution, and is used whenever repartitioning occurs. Once the processors are allocated, the execution rate for a job is determined by its execution signature [4]. The execution signature of a parallel program is defined as the rate at which it executes in the absence of other programs, as a function of the number of processors allocated. The execution signatures of the 4 jobs used in the experiments are shown in Figure 4.

The allocation table consists of one row for every job. The columns correspond to the various possible groups of jobs that could be waiting at the multiprocessor for processors. For example, the column labeled *01__* corresponds to the state where the jobs *0* and *1* are contending for processors, and jobs *2* and *3* are at the host. Each entry in the Table specifies the number of processors that should be allocated to each job when the corresponding group of jobs is waiting at the multiprocessor. For instance, table 1 shows an example of the processor allocation table, obtained using equal partition allocation policy on a multiprocessor with 60 processors. The entries in the column labeled *01__*, are 30 in the rows *0* and *1* and nothing in rows labeled *2* and *3*. The processor allocation policy allocates 30 processors each to jobs *0* and *1*, when these are the only jobs contending for processors.

For the experiments conducted in this paper, the multiprocessor is assumed to have 60 processors. The assumption that repartition occurs only when all the

JOB	0___	_1__	01__	__2_	0_2_	_12_	012_	___3	0__3	_1_3	01_3	__23	0_23	_123	0123
0	60		30		30		20		30		20		20		15
1		60	30			30	20			30	20			20	15
2				60	30	30	20					30	20	20	15
3								60	30	30	20	30	20	20	15

Table 1: The Processor Allocation Table for an Equal Partition Strategy

executing jobs have reached their respective break points implies that processor allocation begins only when all the processors become idle. This avoids very large and very small partitions from co-existing and ensures a fair distribution of processors.

3 Processor allocation strategies

Various processor allocation policies are implemented and compared in this study.

- **Eq**ual **P**artition allocation strategy (EqP) [9] - The processors are distributed equally among the contending jobs. The allocation table is shown in Table 1.
- **Eq**ual **S**lope allocation strategy using the execution signatures (**R**ates) (EqSR) [10] - Processors are allocated to the jobs one at a time. The next processor is allocated to the job for which the increase in the rate of execution is maximum. The equal slope allocation strategy results in allocations such that the slopes of the execution signature curves for all jobs at the processor allocation point are identical.
- **Eq**ual **S**lope allocation strategy using the execution **T**imes (EqST) - This strategy is identical to EqSR, except that it uses the execution time instead of the execution signature curves. The next processor is allocated to the job for which the decrease in its execution time is maximum.
- **Eq**ual **R**ate allocation strategy (EqR) - The next processor is allocated to the job that has the lowest execution rate. This strategy attempts to keep the execution rates for the contending jobs equal, minimizing wait times at the segment boundaries and increasing processor utilization.
- **S**hortest **J**ob **F**irst **P**reemptive **R**esume (SJFPR) [8] - In this strategy all the processors are allocated to the fastest job that is waiting for processors. If a job with a higher execution rate arrives, the currently executing job is preempted.

3.1 The model parameters

The list of model parameters is given below.

- M - The number of parallel programs in the system. The index i, $0 \leq i < M$, is used to refer to a particular program.
- $\lambda_H[i]$ - The rate at which program i completes its host phase and arrives at the multiprocessor.
- $\lambda_O[i]$ - The rate at which program i completes its reconfiguration overhead (repartitioning overhead) phase.

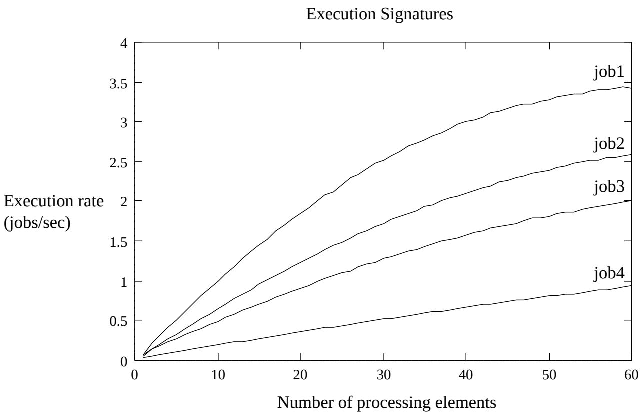


Figure 4: The Workload

- $A[i,j], 0 < j < 2^M$ - The size of the partition allocated to program i when the group of jobs present at the multiprocessor is the one represented by column j. Thus, $A[i,j]$ is simply the entry in the ith row and jth column of the allocation table. A is called the *allocation* or *mapping table*.
- $\lambda_M[i,j], 1 < j \leq N$ - The rate at which program i executes on the multiprocessor in a partition of size j. The array $\lambda_M[i]$ is program i's *execution signature*.
- $T[i]$ - The average number of segments in program i. The execution rate for one segment of the ith job when j processors are allocated to it is $\lambda_M[i,j]/T[i]$.

4 The workload

The workload used for experimentation consists of four parallel jobs. These programs are based on parallel wavefront algorithms [6] for solving triangular systems of linear equations. In these algorithms messages consist of segments of the solution vector.

The programs execute in two alternating phases, the host phase and the multiprocessor phase. In the host phase a program executes a delay loop to mimic the arrival process. In the multiprocessor phase, the program solves a number of triangular systems in its partition.

Figure 4 shows the execution signatures for the jobs used in the experiments. The execution signature specifies the execution rate for a job for a given partition size. Since each job has T segments of equal size, the execution rate for each segment of a job is T times that of the job itself. The execution rate for each segment is determined at the beginning of the segment, and could be different from that of the previous segment if the partition size is changed at the segment boundary.

5 Sensitivity analysis

The Petri net model (Section 2.2) is used to predict the throughput of the system as several parameters are varied. The parameters varied are:

- Multiprogramming level
- Number of segments per job (granularity)
- Arrival rate
- Repartitioning overhead
- Number of processing elements

The model is also used to compare various processor allocation policies. The results are presented in the following sections.

5.1 The multiprogramming level

In the following experiments, the multiprogramming level is increased successively from 1 to 4 and the model is executed to observe the throughput of the system as a function of various multiprogramming levels. The four jobs shown in Figure 4 are used. Job1, job2, job3, and job4 are added to the system in that order to increase the multiprogramming level. That is, when the multiprogramming level is three, jobs 1, 2, and 3 are used. Each job consists of 10 segments and the repartitioning overhead is assumed to be negligible. The results are shown in Figure 5.

Figure 5(a) shows the results obtained using the EqP strategy, and Figure 5(b) shows results obtained using the EqSR strategy. In each of these graphs, throughput is plotted for three different arrival rates. As the multiprogramming level increases, the performance of the equal slope allocation strategy, relative to that of equal partition allocation strategy, improves.

For higher arrival rates, as the multiprogramming level is increased from 2 to 3, the throughput decreases. This can be explained as follows. When job3 is added to the system to increase the multiprogramming level to 3, it reduces the overall average rate of execution of the jobs, since job 3's speedup is less than that of job 1 or job 2. Also, the variance in the execution rates of the job segments increases. These reasons, combined with the fact that an increase in variance also increases the average wait times at the break points, contribute to a decrease in throughput. A further increase in the multiprogramming level does not affect the throuhgput in case of EqSR because job4, which is the slowest, is not allocated processors while there are other jobs present.

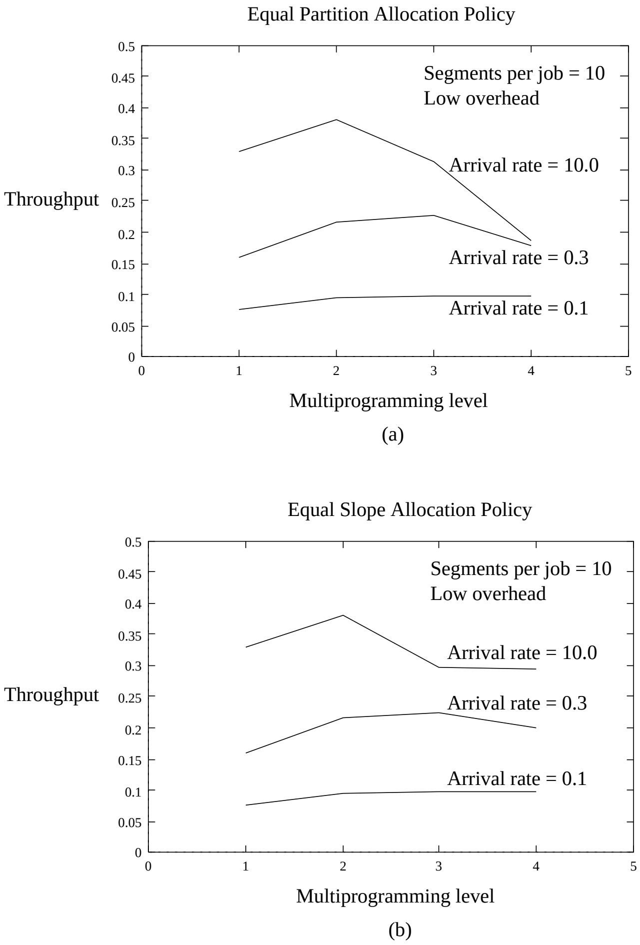
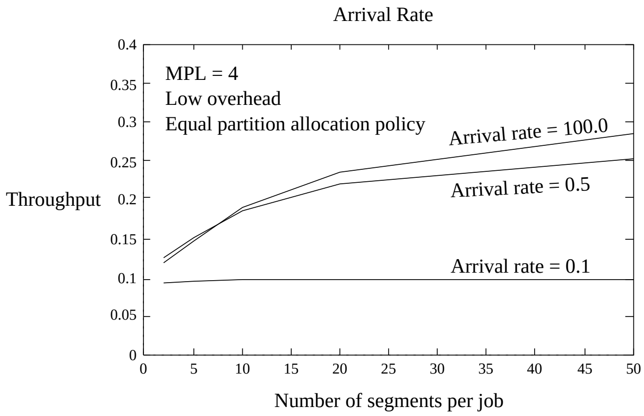


Figure 5: Throughput vs Multiprogramming Level (multi-class workload)

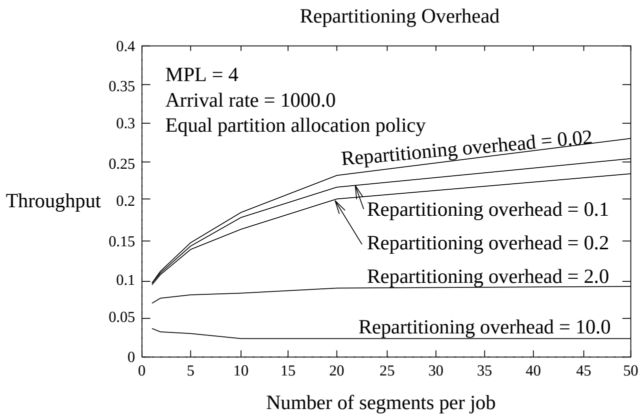
5.2 The granularity

The results in the previous section indicate that the wait time experienced by the jobs at the break points can significantly affect the system throughput. In the following experiments, the effect of the number of segments per job (or granularity) is observed as the repartitioning overhead and the arrival rate parameters are varied. Increasing the number of segments increases the repartitioning frequency since the average (and the variance in) execution time between break points decrease. A decrease in the average and the variance in segment execution times results in a reduced wait time for all jobs reaching the next repartitioning instant.

Arrival rate:
In Figure 6(a), the granularity (i.e., the number of segments per job) is varied from 2 to 50. System throughput is plotted for three different job arrival rates. The multiprogramming level is held constant at 4 and consists of job1, job2, job3, and job4. The processor allocation strategy is EqP. The system through-



(a)



(b)

Figure 6: Throughput vs Granularity

put increases with finer granularity. This is a result of increasing the repartitioning frequency when the repartitioning overhead is minimal.

Repartitioning overhead:
In Figure 6(b), throughput is again plotted against granularity. The model parameters and the setup are identical to those of the previous section, with the exception that the repartitioning rate varies from 0.1 to 50.0. The repartitioning overhead is the time spent in each repartitioning phase.

The throughput remains almost unaffected by the granularity when the repartitioning overhead takes more than 2 seconds (jobs 1, 2, 3, and 4 take 12.782, 16.632, 15.744, and 33.698 seconds respectively to execute on 1 processor), since the number of repartitionings increase as granularity becomes finer and the repartitioning overhead dominates the total execution time. When the overhead is lower, the system throughput increases as the granularity becomes finer because the idle time of a job waiting for all other jobs to reach their break points decreases.

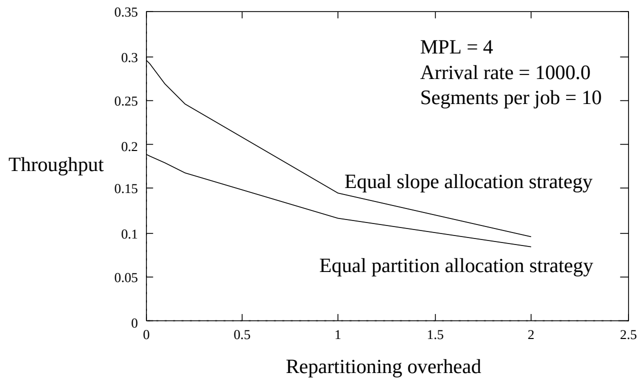


Figure 7: Throughput vs Repartitioning Overhead

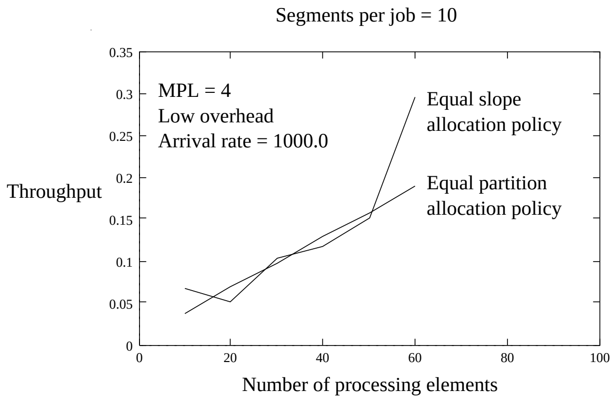


Figure 8: Throughput vs Number of Processing Elements (coarse granularity)

5.3 The repartitioning overhead

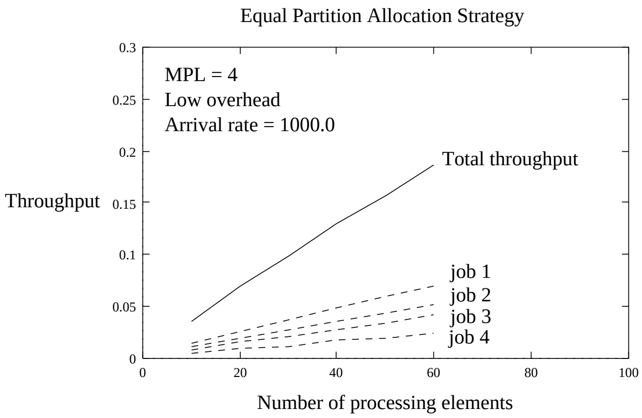
Figure 7 shows the effect of repartitioning overhead on system throughput. For these experiments the multiprogramming level is 4, the time spent by each job at the host is 1ms, and each job had 9 break points. As the repartitioning overhead is increased, and system throughput decreases.

5.4 Number of processing elements

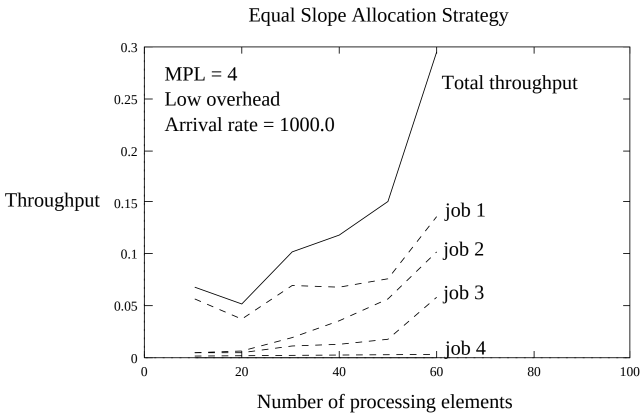
In the following set of experiments the performance of the system is observed as the number of processing elements in the system varies. Two sets of results are presented, one for coarse granularity and one for fine granularity. For the coarse granularity case, each job is assumed to have 10 segments, and for the fine granularity case, each job has 1000 segments.

Coarse granularity:
The results shown in Figure 8 show the effect of the number of processing elements on system throughput, for the EqP and EqSR allocation policies. For these experiments, the time spent at the host is 1ms, the multiprogramming level is 4, and the repartitioning overhead is negligible.

The throughput of the system increases steadily us-



(a)



(b)

Figure 9: Throughput vs Number of Processing Elements (coarse granularity)

ing EqP, and in a somewhat more erratic fashion using EqSR, as the number of processing elements increases. This can be explained as follows. A slower job's segment executes longer than that of a faster job. Since break points act as global synchronization points, slower jobs sometime become the bottleneck. In case of EqSR, it is possible that a faster job receives several processors before the first processor is allocated to the slower one. This results in an erratic increase in the overall system throughput.

Although EqSR can be theoretically proven to yield maximum throughput in the absence of synchronization points, the presence of break points lowers its performance. Since EqSR favors faster jobs, it allocates more processors to the faster jobs. This causes a bigger disparity between the execution times of a fast and a slow job, aggravating the bottleneck situation. As a result, EqSR performs worse than EqP at certain points in Figure 8.

The throughputs of the individual jobs are shown in Figure 9. Figure 9(a) shows results obtained using the EqP allocation strategy, and Figure 9(b) shows results using the EqSR strategy. The throughput for job 4 is close to 0 in the case of EqSR strategy because it is not allocated any processors when other jobs are present at the system, due to its poor use of processors.

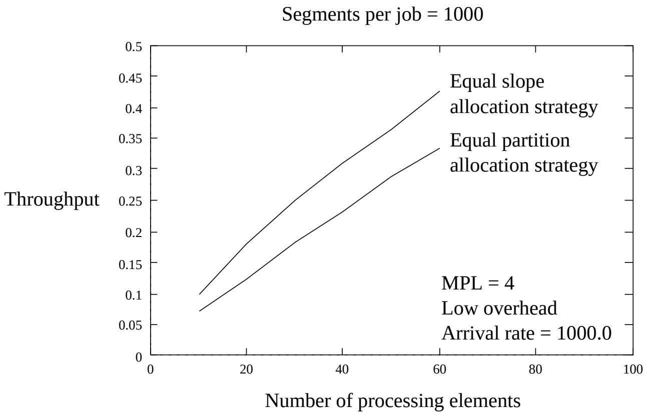


Figure 10: Throughput vs Number of Processing Elements (fine granularity)

Fine granularity:

The experiments performed in the previous section are repeated with 1000 segments per job. Figure 10 shows the results.

With fine granularity, throughput increases smoothly for the EqSR strategy (as well as for the EqP strategy) for the following reason. Since each job has 1000 segments, the execution time for each segment in case of fine granularity is 100 times less compared to that in the case of coarse granularity. This reduces the synchronization wait time spent by the jobs at the break points. Although, the slower jobs still act as bottlenecks, the absolute difference between the segment execution rates for the fast and slow jobs is much less.

Figure 11 shows the throughputs for the individual jobs. When equal slope allocation strategy is used, the first few processors are allocated to job 1, causing its throughput to increase with the total throughput. When the slope of the execution signature of job1 becomes less than that of job2, the next processor is allocated to job2. This continues for job3 and job4. This results in the "stair-step" allocation of processors seen in Figure 11.

6 Processor allocation strategies

The following results demonstrate the relative performances of the processor allocation policies that were described in Section 3. Figures 12 (a) and (b) show the results for multiprogramming levels 3 and 4. Job1, job2, job3 and job4 are added to the system in that order to increase the multiprogramming level. For these experiments, the repartitioning overhead is assumed to be negligible, and the arrival rate is 10 jobs per second. Throughput is graphed as a function

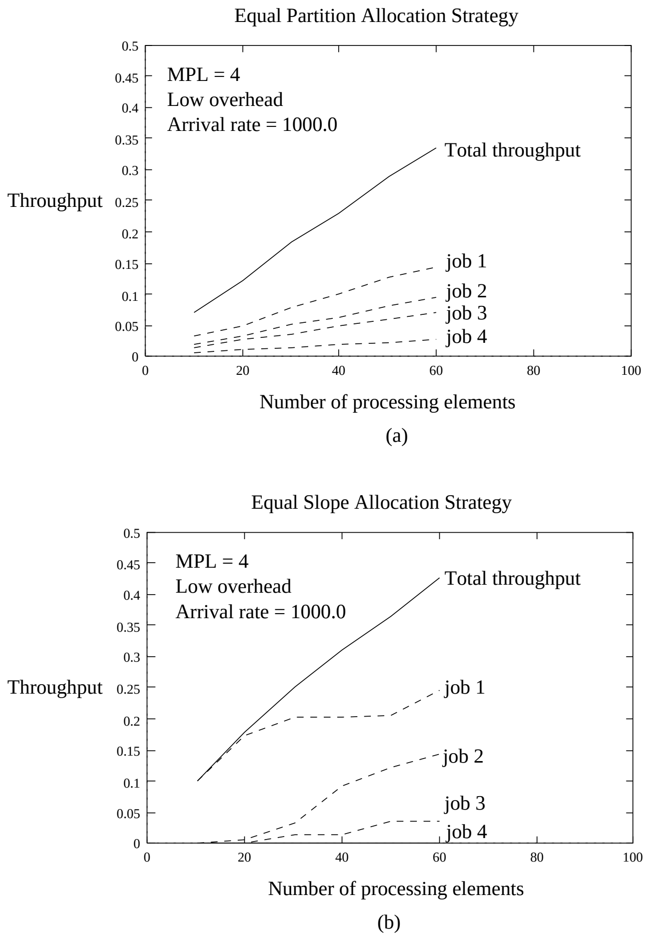


Figure 11: Throughput vs Number of Processing Elements (fine granularity)

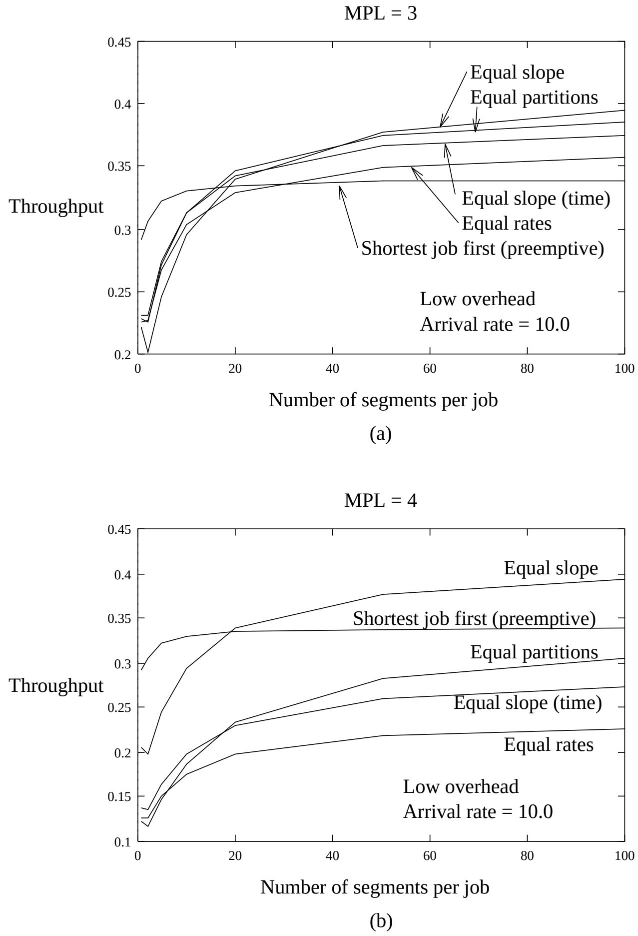


Figure 12: Throughput vs Granularity with Different Allocation Strategies

of the granularity.

The differences in the relative performances are apparent in the figures. The performances of the EqSR and SJFPR policies remain unchanged as the multiprogramming level is changed from 3 to 4. In Figure 12 (b), the performances of EqR, EqST and EqP policies are affected significantly, and are worse than those of EqSR and SJFPR. Performance of EqR, EqST, and EqP policies deteriorate as a result of adding a slower job (*i.e.*, job 4) to the system. This is due to the break points, and the fact that EqR, EqST, and EqP policies result in fairer (*i.e.*, more evenly distributed) processor allocations. The EqSR and SJFPR policies favor the faster jobs and are only mildly affected by the addition of slower jobs to the system.

7 When to repartition

In obtaining the results, the repartitionings are assumed to occur only when all the executing jobs have reached their break points. This means that idle processors are not allocated to the waiting jobs until all the executing jobs have reached their break points. This constraint reduces the number of possible states of the Petri net model, but it also increases processor idle time and lowers system throughput.

The model can be modified to eliminate this constraint. In the modified model, repartitioning occurs as soon as there are idle processors available and jobs waiting for them. When a new job arrives to the system, it waits until the next job (*i.e., not all jobs*) reaches a break point. The processors allocated to the executing job are split between the executing job and the newly arrived job. The other jobs execute undisturbed.

In the following experiments, the original and the modified models are compared using EqP. Due to the limited memory sizes of the machines on which the model is executed, the modified model is used to predict the performance of relatively small systems only.

Multiprogramming level:

The throughput of the system is plotted against the multiprogramming level. The results obtained using 8 processors are shown in Figure 13.

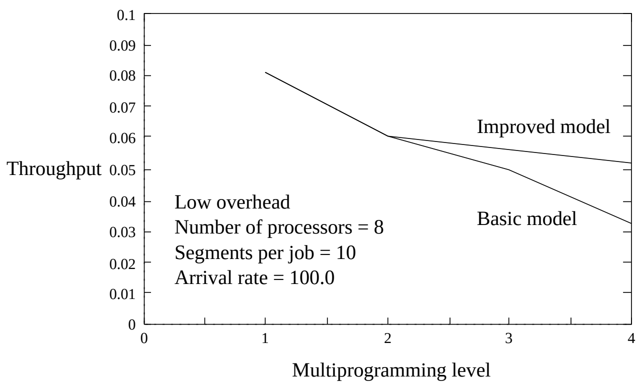


Figure 13: Throughput vs Multiprogramming Level (multi-class workload)

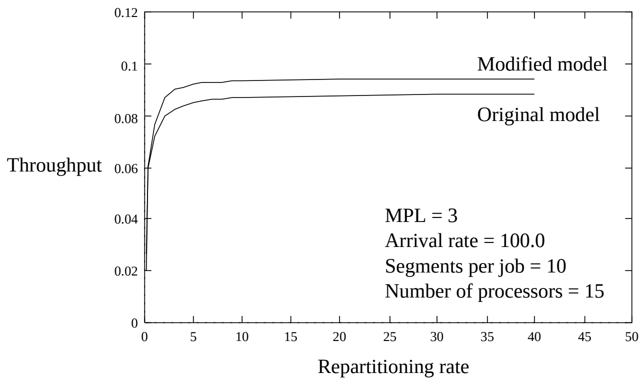


Figure 14: Throughput vs Repartitioning Overhead

When the multiprogramming level is 1 or 2, there is no difference in the two approaches. As the multiprogramming level is increased, the difference between the performances of the two approaches increases. This figure indicates that the modified approach is effective.

Repartitioning overhead:
In Figure 14, the system throughput is plotted against the repartitioning rate, which is the reciprocal of the repartitioning overhead (*i.e.*, the time required for each repartitioning). In the modified approach, the wait time at a break point decreases since a job waits for only one of the executing jobs to reach its next break point. Also, repartitioning can proceed simultaneously with the execution of other jobs. The modified model predicts a marginally better performance compared to the original model for the given values of the various parameters.

8 Comments and future work

This paper demonstrates the feasibility of and usefulness in constructing Petri net models of a dynamically partitioned multiprocessor system. By varying the model parameters, the effective range of several partitioning strategies can be identified.

The model is based on the assumption that the rate at which a job executes depends only on the number of processing elements allocated to it and not on the physical locations of the processors. This may not be valid for NUMA machines, on which the execution rate also depends on the physical locations of the allocated processors. An extension of this work is to take this into consideration.

References

[1] R. M. Chamberlain, "An algorithm for LU factorization with partial pivoting on the hypercube", *Technical Report CCS 86/11*, Chr. Micelson Institute, Bergen, Norway, June 1986.

[2] G. Ciardo, J. K. Muppala, "Manual for the SPNP package (Version 3.1)", 1991.

[3] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, E. Markatos, "Multiprogramming on multiprocessors", in *Proc. of the 3rd IEEE Symposium on Parallel and Distributed Processing*, December 1991, pp. 590-597.

[4] L. W. Dowdy, M. R. Leuze, "On modeling partitioned multiprocessors systems", *Journal of High Performance Computing*, 6,1(1994), pp. 31-53.

[5] K. Dussa, B. M. Carlson, L. W. Dowdy, K. H. Park, "Dynamic partitioning in a transputer environment", in *ACM SIGMETRICS*, May 1990, pp. 203-213.

[6] M. T. Heath, C. H. Romine, "Parallel solution of triangular systems on distributed-memory multiprocessor", *SIAM Journal on Scientific and Statistical Computing*, 9 (1988), pp. 558-588.

[7] Intel Scientific Computers, *iPSC/2 User's Guide*, Beaverton, Oregon, 1988.

[8] S. T. Leutenegger, M. K. Vernon, "The performance of multiprogrammed multiprocessor scheduling policies", in *ACM SIGMETRICS*, May 1990, pp. 226-236.

[9] C. McCann, R. Vaswani, J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed, shared memory multiprocessors", *Technical Report 90-03-02*, Department of Computer Science and Engineering, University of Washington, 90-03-02, February 1991.

[10] K. H. Park, L. W. Dowdy, "Dynamic partitioning of multiprocessor systems", *International Journal of Parallel Programming*, Vol. 18 No. 2 (1989), pp. 91-120.

[11] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, B. M. Carlson, "Robust partitioning policies for multiprocessor systems", *Performance Evaluation 1994*, Special Issue on Parallel Systems, pp. 141-165.

[12] S. K. Tripathi, D. A. Menascé, "Scheduling issues in heterogeneous multiprocessor systems", Transputers 1992.

[13] J. Zahorjan, C. McCann, "Processor scheduling in shared memory multiprocessors", in *ACM SIGMETRICS*, May 1990, pp. 214-225.