Preemptive versus Non-Preemptive Processor Allocation Policies: An Empirical Comparison

Jitendra Padhye

August 31, 1995

Abstract

When a parallel job arrives at a space-sharing multiprocessor system, a decision has to be made regarding the number and the specific identities of the allocated processors. A preemptive partitioning policy may preempt one or more of the currently running jobs to accommodate the new arrival. A non-preemptive policy may consider the state of the system at arrival time, however, it does not allow preemption of a job once the execution begins. In this paper preemptive and non-preemptive policies are presented and investigated experimentally. The performance of these policies are compared on the Intel Paragon. The workload model is based on matrix multiplication applications commonly found on large systems used for scientific applications. Results are reported for single and multiclass cases. A sensitivity analysis with respect to workload speedup characteristics is presented.

Keywords: multiprocessor, preemptive scheduling policies, adaptive scheduling policies, message passing architecture, Intel Paragon

1 Introduction

Multiprogramming is a common way of improving performance of large multiprocessor systems. Most common parallel applications have sublinear speedup curves and hence can not take full advantage of all system processors. For a heavily used system it is not uncommon to have several parallel jobs waiting to use the multiprocessor. In such cases, performance can be improved by sharing the multiprocessor among all or some of the waiting jobs [SRSDS94]. This can be achieved via either space sharing or time sharing.

Under space sharing scheduling polices, an incoming job is assigned to a subset of the total available processors. Thus, multiple jobs can be active within the multiprocessor at the same time. A running job releases its partition only after it finishes execution and it is not possible for multiple jobs to be active on the same processor simultaneously. Under time sharing scheduling policies, the entire multiprocessor, or some subset of it, can be allotted to one of the ready jobs, but only for a fixed time period (quanta). At the end of the time period, if the job has not completed execution, it is returned to the ready queue. Thus, several jobs can be assigned (active) to the same processor concurrently. Combining of space sharing and time sharing policies is also possible. Several issues such as the application speedup characteristics, the underlying multiprocessor architecture, and special requirements of certain application workloads are among the factors affecting the effectiveness of the processor allocation strategies for multiprogramming parallel systems. Several processor allocation policies have been proposed in the literature [Oust82, MEB88, Sev89, TG89, PD89, LV90, DCDP90, ZM90, GST91, GTU91, MEB91, ZB91, MVZ93, SST93, RSDSC94, Sev94, CMV94, MZ94]. Each of these policies is designed to perform well under certain conditions. In this paper preemptive and non-preemptive space sharing policies are considered.

Under preemptive policies, parallel programs can be stopped during execution to allow for resource redistribution according to changing system loads. Time-sharing polices are inherently preemptive in nature. Several space-sharing preemptive policies have also been proposed [TG89, MZ94]. Performance analysis of preemptive space sharing scheduling policies has been largely based on simulation studies and Markovian analysis of small systems. For a simulation study to be accurate and realistic, detailed knowledge of various parameters of the system under consideration is necessary. Also, validation of the the simulation models is difficult. Detailed Markovian analysis to verify simulation models is possible only for small systems (i.e. less than 10 processors) [SRSDS94]. For larger systems, simplifying assumptions have to be made, resulting in a loss of accuracy.

Experimental studies on preemptive processor partitioning policies have been done mainly for shared memory architectures [GTU91]. In this paper, experimental analysis of preemptive processor partitioning policies for a message passing architecture is presented. The primary objective of this study is an empirical analysis and comparison of performance of the preemptive and non-preemptive processor partitioning polices on a message passing architecture. To this end, two preemptive processor partitioning policies based on techniques proposed in [MZ94] and one non-preemptive robust adaptive policy presented in [RSDSC94] are implemented on the Intel Paragon. Two workload programs are used to compare the behavior of the implemented scheduling policies. One is a synthetic workload program designed to emulate various speedup curves and other characteristics of common scientific applications. The flexibility of the synthetic workload allows easy emulation of a wide variety of load conditions for comparing the scheduling policies. The other workload program is based on a parallel, preemptable, distributed memory version of a matrix conjugate gradient program. This program is representative of typical scientific workloads and has been used for similar purposes in the past [BMSD95]. Both the workload programs interface with the scheduling programs via a set of standardized subroutines. This allows for a fair and uniform comparison of the performance of the scheduling polices under consideration.

The rest of the paper is organized as follows. Section 2 describes the experimental setup, including the Paragon architecture, the implementation of the preemptive and non-preemptive scheduling policies and the two workload programs. The results of the study are presented in Section 3. Section 4 presents conclusions and directions for future work.

2 Experimental Setup

2.1 The Intel Paragon

This section briefly describes the architecture of the Paragon Supercomputer. A detailed description may be found in [Int93]. An Intel Paragon Supercomputer consists of several nodes[1] connected in a mesh configuration. The computer used for this study has 66 nodes connected in a 11 by 6 matrix. Several other nodes are dedicated to special tasks such as disk and network control. Each node consists of two Intel i860 processors sharing 16 MB of memory. One of the processors runs the application program while the other acts as a communication co-processor. This arrangement is transparent to the user who sees the node as a single computing entity. To route messages between the nodes, the wormhole routing protocol is used. The operating system confirms to OSF/1 standards. Although the machine is capable of supporting MIMD (or MPMD) computing models, most applications use the SPMD model for computing. The workloads for this study use the SPMD model.

2.2 Non-Preemptive Policies

The Robust Adaptive (RA) scheduling policy is proposed and discussed in [RSDSC94]. RA actually represents a suit of scheduling policies rather than a single scheduling algorithm. The version chosen here is representative and has been used in subsequent studies [SRSDS94]. RA is a non-preemptive scheduling policy. Once a job is scheduled on a partition, it executes on the same partition without any further interruption from the scheduler until completion. The policy is called an adaptive policy because it adapts to load changes over a period of time. That is, the number of allocated processors depends on, (i.e. adapts to) the current system state. Two identical jobs may be allocated different sized partitions, depending upon the system state when each job arrives. When a new job arrives in the system, it is placed at the end of a queue of jobs awaiting service. The scheduler calculates a "target" partition size as follows:

$$\text{Target partition size} = \ max\left(1,\ \left\lfloor\frac{\text{Total number of processors in the system}}{\text{Number of jobs waiting in the queue}}\right\rfloor\right)$$

As soon as the target number of processors are available, the scheduler starts allocating processors to the jobs from the waiting queue in FCFS manner. The target is not recomputed during this time. Once no more jobs can be scheduled, either due to an empty queue or due to a lack of sufficient free processors, the scheduler stops scheduling new jobs and recomputes a new target. The target recalculation allows RA to adapt to changes in the load condition. A higher load would result in a longer queue, resulting in a smaller target which in turn means that more jobs are scheduled and allocated partition size for each job is reduced.

Figure 1 shows the RA policy in operation for a job arrival sequence consisting of 4 jobs, on a system with 16 processors. The timeline shows job departures and arrivals. For each of these events, the state of multiprocessor and the value of the target are also indicated. The multiprocessor

[1] Lees that 2000 in current models.

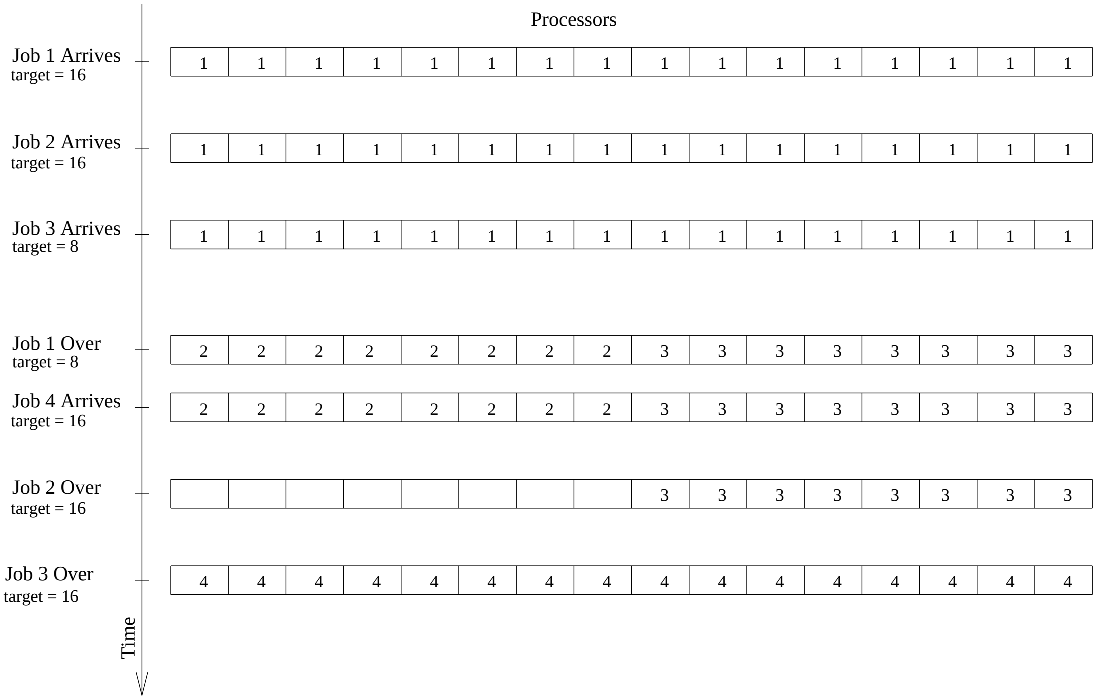

Figure 1: The RA policy

is represented by 16 rectangles, each of which represents a processor. The number within each rectangle indicates the the job executing on that processor. An empty rectangle indicates an idle processor. When job 1 arrives at the multiprocessor, it is the only one in the queue, so the target is calculated to be 16, and the job is scheduled on all 16 processors. When job 2 arrives, the queue length is still 1 (as job 1 has been scheduled) and the target remains at 16. When job 3 arrives, there are two jobs waiting in the queue and the target drops to 8. When job 1 is completed, 8 processors are allocated to both jobs 2 and 3. When job 4 arrives, it is the only one waiting in the queue, so the target rises to 16. When job 2 finishes, the target is still 16, and there are only 8 processors free. Hence, these 8 processors are kept idle and job 4 continues to wait. When job 3 finishes, 16 processor become free and job 4 is scheduled on all of them.

2.3 Preemptive Partitioning Policies

Preemptive partitioning policies can stop currently executing jobs in order to redistribute processors among the jobs because of job arrivals and/or departures. Thus, preemptive scheduling policies can adapt quickly adapt to transient changes in the workload flow. The two preemptive processor partitioning policies considered in this study are based on the policies proposed in [MZ94]. Both the preemptive scheduling policies are implemented around a standard interface. The interface consists of six reserved message id numbers, two data structures and three library routines that are incorporated into the workload programs. The interface is independent of the workload and the scheduling algorithm involved. The same interface, with minor modifications, is also used to implement the non-preemptive RA policy.

When a job is preempted and restarted on a new partition, the data set of the job has to be redistributed so that each processor in the new partition has the correct subset of the data. Careful design of the preemption sequence is necessary to minimize the preemption overhead. The data redistribution routine is unique for each workload program since it depends on the data structures used by the workload program as well as its logical flow. Two steps were taken to minimize the data redistribution overhead during preemption.

1. The processor allocation routine observes the *subset* restriction. The *subset* restriction is defined as follows. If P_1 and P_2 represent the sets of processors in two successive partitions allotted to a job then:

$$\begin{aligned} P_1 \bigcap P_2 &= P_1 \quad \text{if } |P_1| \leq |P_2| \\ &= P_2 \quad \text{otherwise} \end{aligned}$$

   This restriction ensures that the application data does not have to be stored on any nodes that do not belong to the application, and helps minimize the data transfers required during redistribution.

2. The preemption process is divided into four distinct steps.

   (a) The scheduler requests that the job be preempted. Whenever a job reaches its next breakpoint, it informs the scheduler that it is ready to be preempted.

(b) Once the scheduler receives such replies from all the jobs to whom such requests were sent, it calculates the new processor allocations. All the preempted jobs are informed of the new partitions.

(c) Depending on the size of the new partition, the job executes one of the following three steps.

i. The size of the old and the new partitions are the same. In this case, the *subset* restriction implies that the job will be restarted on exactly the same processors as before. Hence, no data redistribution is necessary in this case. [2]

ii. The size of the new partition is smaller than the size of the previous partition. Such a job is termed as *LOSER*. A *LOSER* job determines the subset of its old partition that forms the new partition. The data is redistributed so that all the application data is stored on the processors in this subset.

iii. The size of the new partition is larger than the old one. Such a job is termed a *WINNER*. A *WINNER* job has to wait until the extra processors are allocated to it to do its data redistribution.

(d) Once the data redistribution is finished, the jobs release their nodes or acquire new nodes as may be required, and wait for a *RESTART* message from the scheduler. The *WINNER* jobs redistribute their data during this wait period. The *LOSER* jobs, on the other hand, have already finished their data redistribution. After receiving the *RESTART* message, the jobs resume their normal execution.

2.3.1 Equipartitioning

Equipartitioning is the most natural and intuitive preemptive scheduling policy. The policy is proposed in [MZ94]. The policy tries to equally allocate the available processors among all jobs that are present in the system. On each new arrival or departure, all the currently executing jobs are preempted. The new partition size for each job is calculated as follows:

$$\text{New partition size} = \; max\left(1, \lfloor\frac{\text{Total number of processors in the system}}{\text{Number of ready jobs}}\rfloor\right)$$

The number of ready jobs is calculated as follows.

$$\text{Number of ready jobs} = \text{Number of preempted jobs} + \text{Number of jobs awaiting service}$$

Since at least one processor has to be allowed to each job, it may not be possible to restart all the ready jobs. However, the scheduler always restarts all the preempted jobs. Figure 2 shows equipartitioning policy in operation for a job arrival sequence consisting of 4 jobs, on a system with 16 processors[3]. Initially, as job 1 arrives, it is started on all 16 processors. When job 2 arrives,

[2] This is clearly inefficient, as the job should not have been preempted in the first place. The implementation of the Folding policy has been optimized to avoid this problem. However, this situation can not be avoided under Equipartitioning.

[3] This arrival sequence does not necessarily represent the same jobs as in the sequence for the RA policy. A similar time line has been maintained for easy comparison.

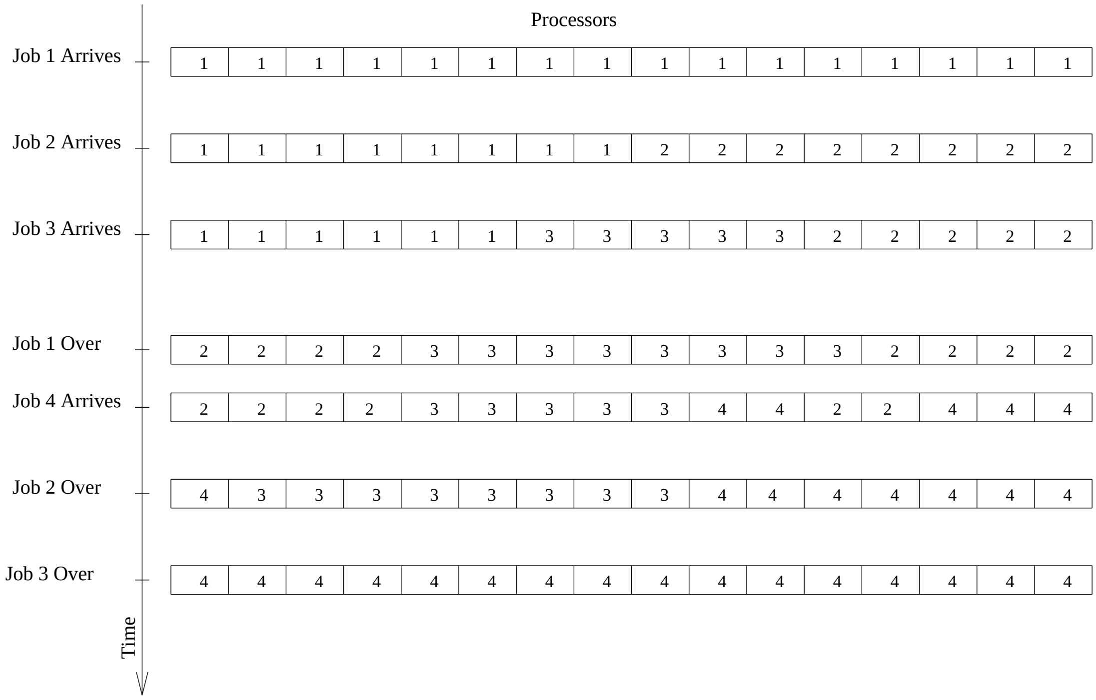

Figure 2: The Equipartitioning policy

job 1 is preempted and the new partition size is calculated to be 8. Both job 1 and 2 are started on 8 processors each. When job 3 arrives, both job 1 and 2 are preempted. The partition size is calculated to be 5. As the implementation allocates any extra processors (1 in this case) to jobs in the order of the job numbers, job 1 is allocated 6 processors while job 2 and 3 are allocate 5 processors each. When job 1 finishes, both jobs 2 and 3 are preempted and the new partition size is calculated to be 8. Job 2 and 3 are then started on 8 processors each. When job 4 arrives, it is allocated 5 processors while job 2 gets 6 processors and job 3 gets 5. When job 2 finishes, job 3 and 4 are preempted and restarted with 8 processors each. Finally, when job 3 gets over, job 4 is preempted and restated on all 16 processors.

The policy suffers from two major disadvantages.

1. A preemptive workload can theoretically be preempted many times under this policy. Even if individual preemptions are not very costly, the total cost can be prohibitive.

2. Every time the processors need to be reallocated, equipartitioning preempts all jobs running on the current partition. Some of these jobs can be in their last phase and hence may never reply to the preemption request. Thus, during each repartitioning, the scheduler has to wait until all the processors in the partition are released before calculating the new partition size for each job. Jobs that get preempted earlier are forced to wait until all the jobs running on the multiprocessors free their own partitions. This delay is called the *synchronization delay.*

This policy enforces an equal division of resources. In practice, the performance of the policy is quite poor, mainly because of the excessive number of preemptions caused by it. However, the policy can be used as a standard against which other preemptive policies can be measured. Three principle variations of the equipartitioning policy are possible.

1. The total number of preemptions for a job are restricted to an upper bound. After each new job arrives or departs, only those jobs that have not already reached their preemption limit are preempted and the processors are redistributed equally among them. This modification of the basic policy prevents an excessive amount of preemptions.

2. The number of active jobs at any given time is restricted to an upper bound. This ensures that each job is allotted at least a certain minimum number of processors. The variation is motivated by factors like the memory requirements of each job and the minimum number of tasks required to execute the job. After every arrival, the scheduler repartitions the multiprocessor if and only if the number of active jobs is less than the predefined limit. After each departure, the number of jobs admitted from the ready queue is limited so that the number of active jobs in the system does not exceed the predefined limit. As in the first variation, this has the effect of limiting the number of possible preemptions.

3. A combination of the two above. Preemptions are restricted such that each job is preempted no more than a certain predefined number of times and no more than a certain number of jobs are active at any time.

2.3.2 Folding

The Folding policy is proposed in [MZ94]. The policy, as described in the paper, is rather difficult to implement due to complicated partition movements such as rotation, in which partition sizes of neighboring jobs are compared and depending on the ratio, partition size for some jobs is doubled while it is halved for some others. Also, the results of [MZ94] suggest that a long interval between successive rotations leads to better performance. Hence, rather than implementing the policy as described in the paper, a variation is implemented as follows. At each new arrival, the scheduler checks to see if there are any free processors. If there are free processors in the system, the new job is started on all of them. No preemptions take place in this case. If there are no free processors, then the job running on the largest partition is preempted and half of the processors are given to the incoming job. This is called folding. If there are multiple candidates for folding then the implementation can select any one of them. The implementation used for this study preempts the job that arrived last in the event of such a tie. At any time, if the system has any idle processors, the job executing on the smallest partition is preempted and the idle processors are merged with the job's old partition and the job is restarted on the new partition thus formed. This is called unfolding. If there are several candidates for unfolding, the implementation may choose any one of them. The implementation used for this study unfolds the job that arrived the last. Three variations of this policy, similar to the one described for equipartitioning are implemented. The implementation of the algorithm is similar to that of equipartitioning, except for the fact that each time only one job is preempted instead of several.

Figure 3 shows the Folding policy in operation for a job arrival sequence consisting of 4 jobs, on a system with 16 processors. [4] When job 1 arrives at the multiprocessor, it is allocated all 16 processors. When job 2 arrives, partition belonging to job 1 is folded and 8 of its 16 processors are allocated to job 2. Thus, job 1 and 2 both have 8 processors. When job 3 arrives partition belonging to job 2 is preempted. Four of the eight processors are allocated to job 3 and job 2 and 3 are started on four processors each. When job 1 finishes, there are no waiting jobs in the queue. Partition belonging to Job 3 is then unfolded and job 3 starts running on 12 processors. When job 4 arrives, partition belonging to job 3 is folded and 6 of its 12 processors are allocated to job 4. Job 3 and 4 now both have 6 processors. When job 2 finishes, partition belonging to job 4 is unfolded and job 4 starts running on 10 processors. Finally, when job 3 finishes, partition belonging to job 4 is unfolded and all 16 processors are allocated to it.

2.4 Workload

This section describes the two workloads used for the experimental study. The first is a synthetic workload, designed to have enough flexibility to emulate various speedup curves. Computation versus communication costs and preemption overheads are easily adjustable. The second workload is based on the Conjugate Gradient method for matrices. The program is a representative example of the scientific workload and has been used for similar purposes in other studies [BMSD95]. The

[4] This arrival sequence does not necessarily represent the same jobs as in the sequence for the RA or equipartitioning policies. A similar time line has been maintained for easy comparison.

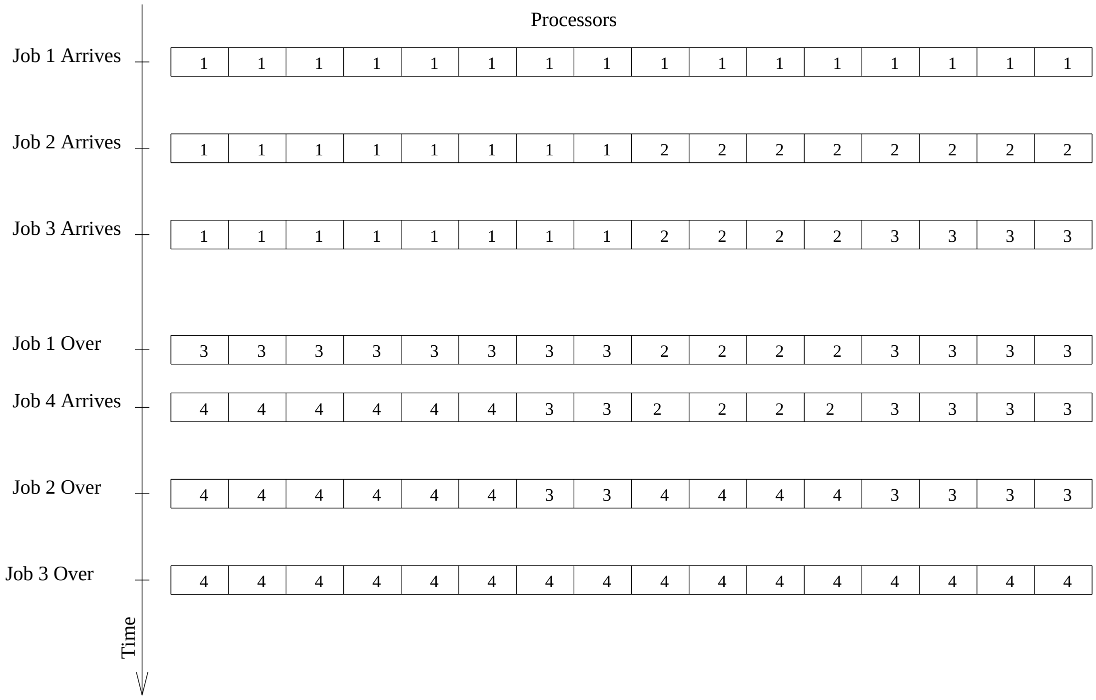

Figure 3: The Folding policy

program is modified to make it capable of running under a preemptive scheduler. The code change is restricted to the data initialization routines at the beginning of the program and synchronization routines at the end of every phase. The former are modified to accept initialization messages from the scheduler while the later are modified to check for preemption and call the standard preemption handler routines. In addition a data redistribution procedure is added to handle redistribution and re-alignment of data segments in the event of a preemption.

2.4.1 Synthetic workload

The synthetic workload is designed to be flexible enough to generate various different speedup curves by varying workload parameters. The workload is capable of of spawning as many tasks as the number of allocated processors [MZ94]. When the workload is started on a given partition, the lowest numbered node in the partition is selected as the "co-ordinator". The workload operates in phases. Each phase consists of three subphases, namely, the broadcast communication subphase, the compute subphase and the collect communication subphase. The execution of the synthetic workload is controlled by a table called configuration table, which lists the number of workload phases. Also, for each phase it lists:

- The amount of data to be transferred between the local controller and each node during the broadcast communication subphase.
- The amount of computations in the compute subphase.
- The amount of data transferred from each node to the local controller during the collect communication subphase.
- The total amount of data that needs to be redistributed if the job is preempted at the end of this phase.

The three subphases are described below.

1. Broadcast communication subphase

   The co-ordinator broadcasts a message to each node in the partition. This message contains initialization parameters for that phase and other problem related data. The size of the message is determined by the corresponding entry in the configuration table.

2. Compute subphase

   Each node in the partition does the amount of computations indicated in the corresponding entry in the configuration table. There is no communication among the nodes during this phase.

3. Collect communication subphase

   Each node in the partition sends back a message to the co-ordinator node. This message contains partial results and other synchronization data. The size of the message is determined by the corresponding entry in the configuration table. This saves the current state of the task at the co-ordinator node.

At the end of each phase the nodes undergo a barrier synchronization. At the end of the barrier synchronization, the local controller checks to see if a preemption request has arrived from the scheduler. If such a request is present, a preemption sequence is initialized. Otherwise, next phase of the communicate-compute-communicate cycle is started. The preemption sequence is as follows.

1. All the nodes undergo an additional barrier synchronization, to ensure that all the tasks of the job are in the same state before preemption. At the end of the synchronization, the local controller informs the scheduler that the job is ready to preempt.

2. Once the scheduler knows that each node is ready, it sends to each node information about the new partition assignment. The information consists of the partition size and the node numbers.

3. Depending on the size of the new partition, one of the two actions occurs:

   - When the new partition size is smaller than the current partition size:

     The nodes which no longer belong to the new partition transfer their data to the remaining nodes. The amount of data transferred is dependent on the phase in which the workload is executing. It might also be necessary to transfer data among the remaining nodes themselves, to achieve proper alignment of data.

   - When the new partition size is larger than the current partition size:

     The data is to be redistributed so that the newly allocated nodes receive the required problem data. This data transfer, however, can not be done at this time, since the required nodes are not yet under the control of the job. The data redistribution is done once the new nodes come under the control of the job.

4. The local controller sends back a message to the scheduler that the job is now ready to preempt. After sending this message, the nodes stop executing the workload program and wait for re-initialization.

Changing the ratio of computation to communication in each phase yields different speedup curves. The preemption cost can be varied by changing the data redistribution required after each phase or by changing number or the length of the phases. Note that the cost of a preemption is reflected in the waiting time of the jobs that are waiting in the arrival queue and synchronization delay for the jobs that have already been preempted.[5] The cost of preemption has a lower bound of approximately half the length of each phase of the job, since, on average, a job that is being preempted will have half a phase to go before it can act on the received preempt request. Thus, to lower preemption cost of a workload, a large number of small phases is preferable to a small number of large phases.

The speedup curves generated by the synthetic workload appear in Figure 4. The speedup curves are generated by executing the workload on different numbers of processors. The execution time is measured for various number of allocated processors and the speedup curves are derived. The

[5]This delay is observed in the case of Equipartitioning only. Folding preempts only one job at a time and thus does not suffer from this delay.

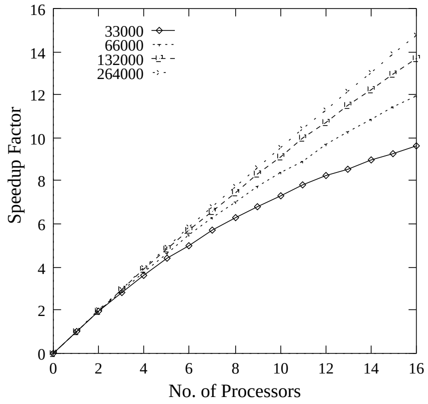

Figure 4: Speedup Curves for the Synthetic Workload

different speedup curves result from different number of computations performed during each workload phase. The other parameters like the size o messages in broadcast and collect communication phase are held constant for each run. Specifically, synthetic workload speedup curves are generated with the following parameters:

- For each curve, the number of *total* computations done per phase is indicated in the legend.
- For all the curves, the number of phases is restricted to 50.
- The message size during both broadcast as well as collect communication subphase is 32 bytes.
- For all the curves, the time taken by the program for a given partition size was averaged over 10 separate program runs.

2.4.2 Conjugate Gradient Workload

The task graph of SPMD version of the Conjugate Gradient (CG) program is shown in Figure 5. N represents the problem size (i.e. NxN matrix) and P represents the number of processors. The circles indicate the complexity of each stage. The values at the arrows represents the number of data items transmitted. The program consist of several phases in each iteration. A brief explanation of each phase follows. A more detailed description may be found in [BMSD95].

- *CP1:* Distributed computation of a scalar vector product.
- *CM1:* Global collect of a distributed vector using tree topology.
- *CP2:* No computations.
- *CM2:* Global broadcast of the collected vector using tree topology.

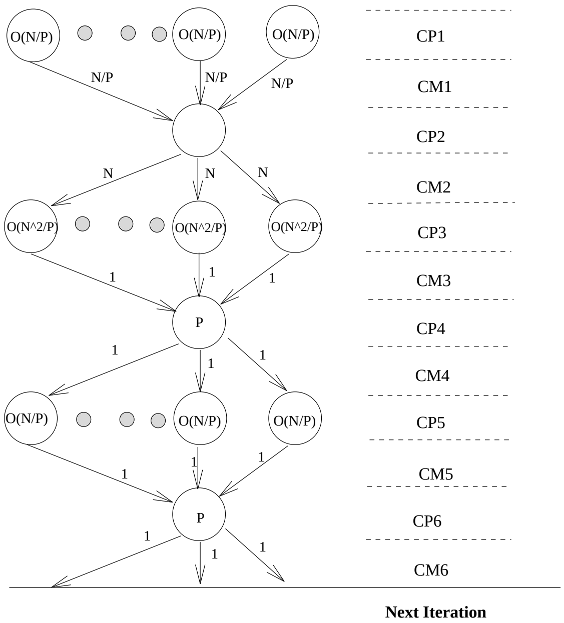

Figure 5: Task graph of the Conjugate Gradient program

- *CP3:* Distributed computation of a matrix vector product.
- *CM3:* Global sum using tree topology.
- *CP4:* Global sum built by one processor.
- *CM4:* Global sum broadcast using tree topology.
- *CP5:* Computation of two scalar-vector products and one scalar product. This step has been modified from the original algorithm to provide better speedup characteristics for smaller matrix sizes.
- *CM5:* Same as CM3.
- *CP6:* Same as CP4.
- *CM6:* Same as CM4.

The program executes a specified number of iterations or until the user specified accuracy is achieved. For test purposes the accuracy testing feature is disabled and the program continues for a specified number of iterations. No special sparse matrix algorithms are implemented. Preemption is allowed only at the end of each iteration. Data redistribution involves redistributing data contained in five different arrays or matrices. Thus the preemption overhead involved can be high. The speedup curves for the CG workload are shown in the Figure 6. The different speedup

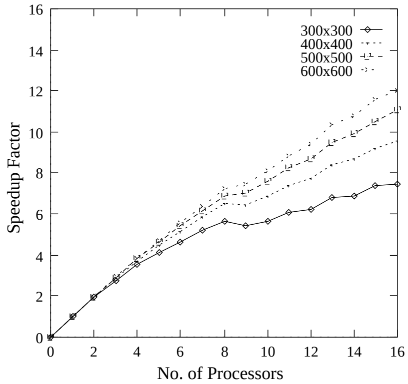

Figure 6: Speedup Curves for the Conjugate Gradient Workload

curves are generated by using different matrix sizes. Specifically, the CG workload speedup curves were generated with the following parameters:

- For each curve, the legend indicates the number of rows/columns of the square matrix used as data set for the program.
- Each CG program performs 250 iterations on the data matrix.
- Timings for each program are averaged over 20 separate runs for each partition size.

3 Results

The experiments described in this study are carried out on the 66 node Intel Paragon System at Oak Ridge National Laboratories. The results presented here constitute a representative sample of several experiments carried out over a period of one year. To make for fair comparisons, all the experiments were run during a time period in which the system software/hardware did not undergo changes. Also, only the most general version of Equipartitioning and Folding policies are used for comparison purposes. Under the general version of Equipartitioning and Folding, a job may be preempted any number of times (i.e., once per phase) and the smallest allowable partition is a single processor. The effect of changing these parameter values on the performance of the scheduling policies depends strongly on the speedup characteristics of the workload being scheduled. To minimize the effects of such implications, comparisons are restricted only to the most general version of the policies.

The rest of this section is organized as follows. In Section 3.1, the performance of the Equipartitioning policy is compared against that of the Folding policy. In 3.2, sensitivity analysis results of the relative performance of RA and Folding policies are presented. Factors considered include constant versus exponential demand, speedup curves, and multiclass workloads. In Section 3.3, the performance of the RA and the Folding policies is compared on a larger multiprocessor to show that the results are scalable.

3.1 Equipartitioning versus Folding

Figure 7 compares the performance of the Equipartitioning and the Folding scheduling policies. The workload used is the CG program with a 500x500 matrix. The program iterates through 250 phases. Preemption requires redistribution of approximately 2 megabytes. The interarrival times are exponentially distributed, while the workload demand is constant, i.e., each program in the workload stream is identical to all others.

The primary results are:

- The Folding policy exhibits a better response time than Equipartitioning under all arrival rates.
- The difference between the response times is large and increases with increasing arrival rates.

The increase in response time under Equipartitioning is a result of the following factors:

- Folding allocates more processors to each job than Equipartitioning at all arrival rates.
- The average number of preemptions per job is higher under Equipartitioning than under Folding.
- The above two factors, taken together, result in a higher execution time per job under Equipartitioning than under Folding.
- At high arrival rates, jobs experience higher waiting times under Equipartitioning than under Folding.

It is seen from these results that Folding significantly outperforms Equipartitioning. This trend is also observed for various other speedup curves and workload mixes. Hence, the remaining results in this section compare only the RA and the Folding policies.

3.2 RA versus Folding

3.2.1 Constant versus Exponential Workload Demand

To compare the performance of the policies under constant and exponential workload demands, synthetic workload is used instead of the CG program. With synthetic workload it is easy to vary the demand imposed by the workload on the multiprocessor by manipulating the number of computations. The workload demand for synthetic workload varies linearly with respect to the number of computations performed. Figure 8 compares the performance of the RA and Folding scheduling policies. The workload used is the synthetic workload with 50 phases. The workload performs 66000 floating point multiplication per phase. When preempted, 0.5 megabytes of data

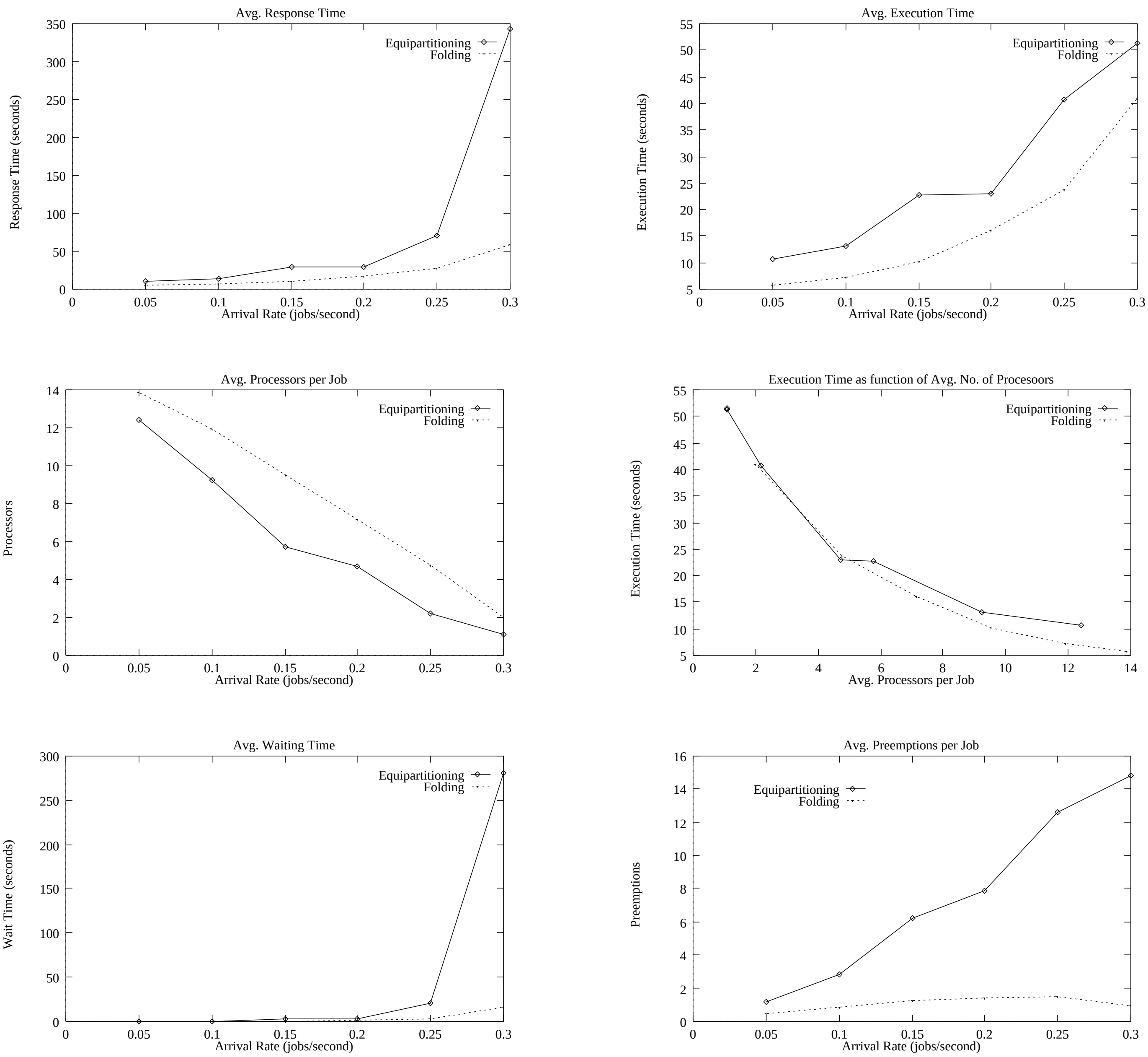

Figure 7: Equipartitioning versus Folding

requires redistribution. The interarrival times are exponentially distributed. The workload demand, however, is constant. In other words, each program in the workload stream is identical to all others. The primary results are:

- The Folding policy gives a better response time than RA across all arrival rates.
- The maximum improvement of Folding over RA in terms of response time is approximately 17%, achieved at higher arrival rates.
- The performance advantage of Folding over RA is attributed to the reduced waiting time under Folding. The maximum improvement of folding over RA with respect to the waiting time is over 90% at moderate and higher arrival rates. Folding schedules a newly arrived job as soon as it can either find free processors or preempt one of the existing jobs. RA makes a newly arrived job wait until it can allocate a partition of current target size. Except at very low arrival rates, the target partition size usually can not be allocated until one or more of the currently executing jobs get over. Thus the jobs experience higher waiting time under RA.
- As arrival rate increases, Folding is able to allocate more processors to each job than is RA.
- Job execution times are higher under Folding than under RA. This is due to the preemption overheads. This can be seen from two graphs in Figure 8, first, when execution time is plotted as a function of average no. of processors per job and second, when execution time is plotted as a function of the arrival rate. The maximum increase in the execution time is approximately 17%, observed at lower arrival rates. This does not coincide exactly with the maximum number of preemptions. The preemption overhead also depends on the size of partitions before and after preemptions. These sizes determine *how many* processors will participate in the data redistribution, and hence affect the data redistribution overhead. Since the average partition sizes are higher at lower arrival rates, higher preemption overheads are observed at these arrival rates.
- The average number of preemption per job is small under Folding. At most, approximately, 1.4 preemptions per job occur at the higher arrival rates. Thus Folding is effectively able to use preemptions in a conservative manner. This keeps the preemption overhead low.

Figure 9 compares the performance of the RA and Folding scheduling policies when the workload demand is exponentially distributed. The synthetic workload program is used for this comparison as well. All the workload parameters are same as in the previous comparisons, except for the number of computations performed per phase. This number is drawn once per workload from a negative exponential random distribution with a mean of 66000 and then rounded to the next higher integer. Since the bulk of the time spent by the synthetic workload is spent in the compute phase, this technique provides a close approximation of an exponentially distributed workload demand. The key observations are:

- The Folding policy exhibits a better response time than RA across all arrival rates.

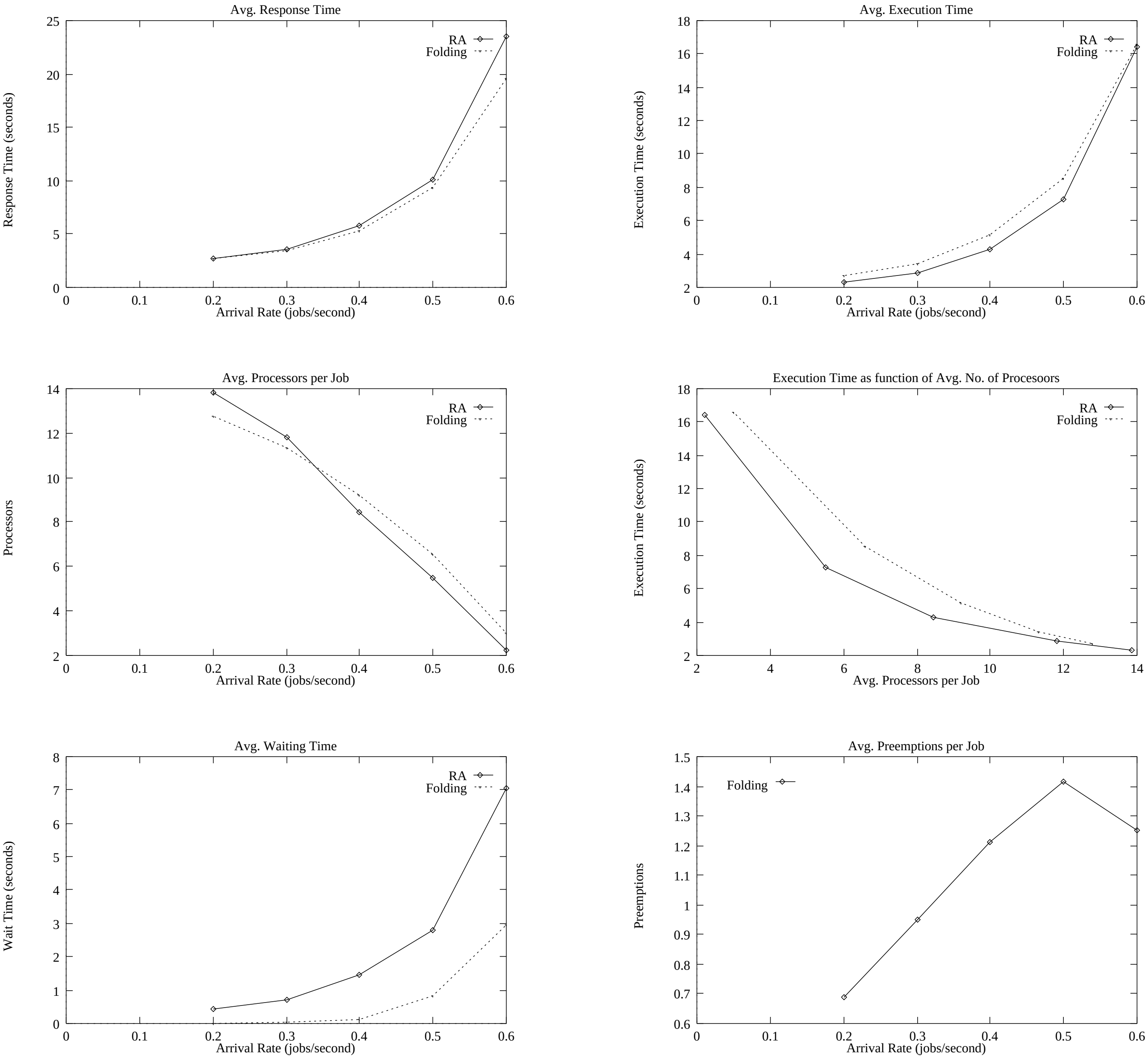

Figure 8: RA versus Folding: Synthetic workload with Constant Demand

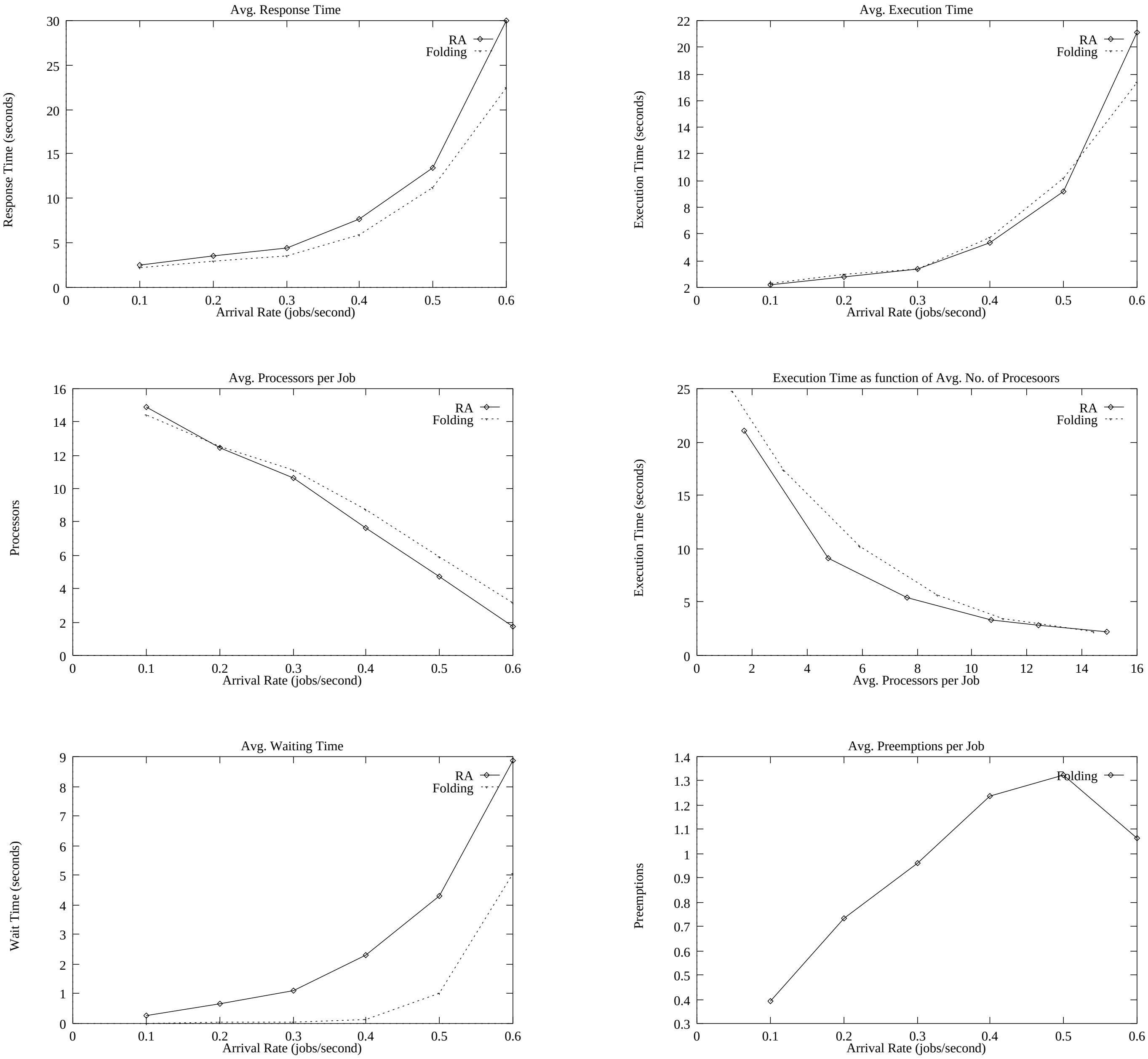

Figure 9: RA versus Folding: Synthetic Workload with Exponential Demand

- The response time improvement under Folding policy is more marked when the demands are exponentially distributed than when the demands are constant. The maximum improvement of about 25% is achieved at high arrival rates.

- An improvement of approximately 97% in the wait time under Folding policy is achieved at low arrival rates.

- As the arrival rate increases, Folding allocates more processors to a job than RA. As RA does not preempt currently executing jobs, average waiting time (and hence average queue length) is higher under RA than under Folding. RA calculates the target partition size by dividing the total number of processors by number of waiting jobs. As the queue length increases, the target size gets smaller and smaller [6]. Under Folding policy, the average waiting time is lower. Folding also unfolds currently executing jobs to execute them on larger partitions whenever possible.

- Execution time of job is higher under Folding than under RA due to preemption overheads. The maximum increase in the execution time is about 10%, at moderate-to-high arrival rates. However, it is interesting to observe that at the highest arrival rate considered, the execution time is actually better under Folding. This is due to the increased numbers of processors allotted to a job under Folding. Allocating more processors results in a decreased execution time and this improved performance offsets the cost of preemption overhead.

- The average number of preemption is small under Folding. The maximum is 1.32 preemptions per job.

3.2.2 Performance under Different Speedup Curves

Figure 10 shows the performance of the RA and the Folding policies when scheduling a conjugate gradient workload with a 300x300 matrix. Each program iterates through 250 iterations. When preempted, approximately 0.72 megabytes of data require redistribution. The interarrival times are exponentially distributed, while the workload demand is constant. The key observations are:

- The Folding policy exhibits a better response time than RA across all arrival rates.

- The maximum improvement in the response time of Folding over RA is approximately 19%.

- The maximum improvement in the wait time under Folding over RA is approximately 99%.

- As in the synthetic workload case, the execution time is better under RA policy than under Folding across all arrival rates. The maximum difference is approximately 20%.

- The average number of preemptions is small, under 1.7 per job. The number of preemptions is small at both lower as well as higher arrival rates, while it peaks in the middle of the range. At lower arrival rates, an executing job is less likely to be preempted during its run time, as new jobs arrive at a slower rate. At higher arrival rates, several of the currently executing

[6] The minimum possible target size is one.

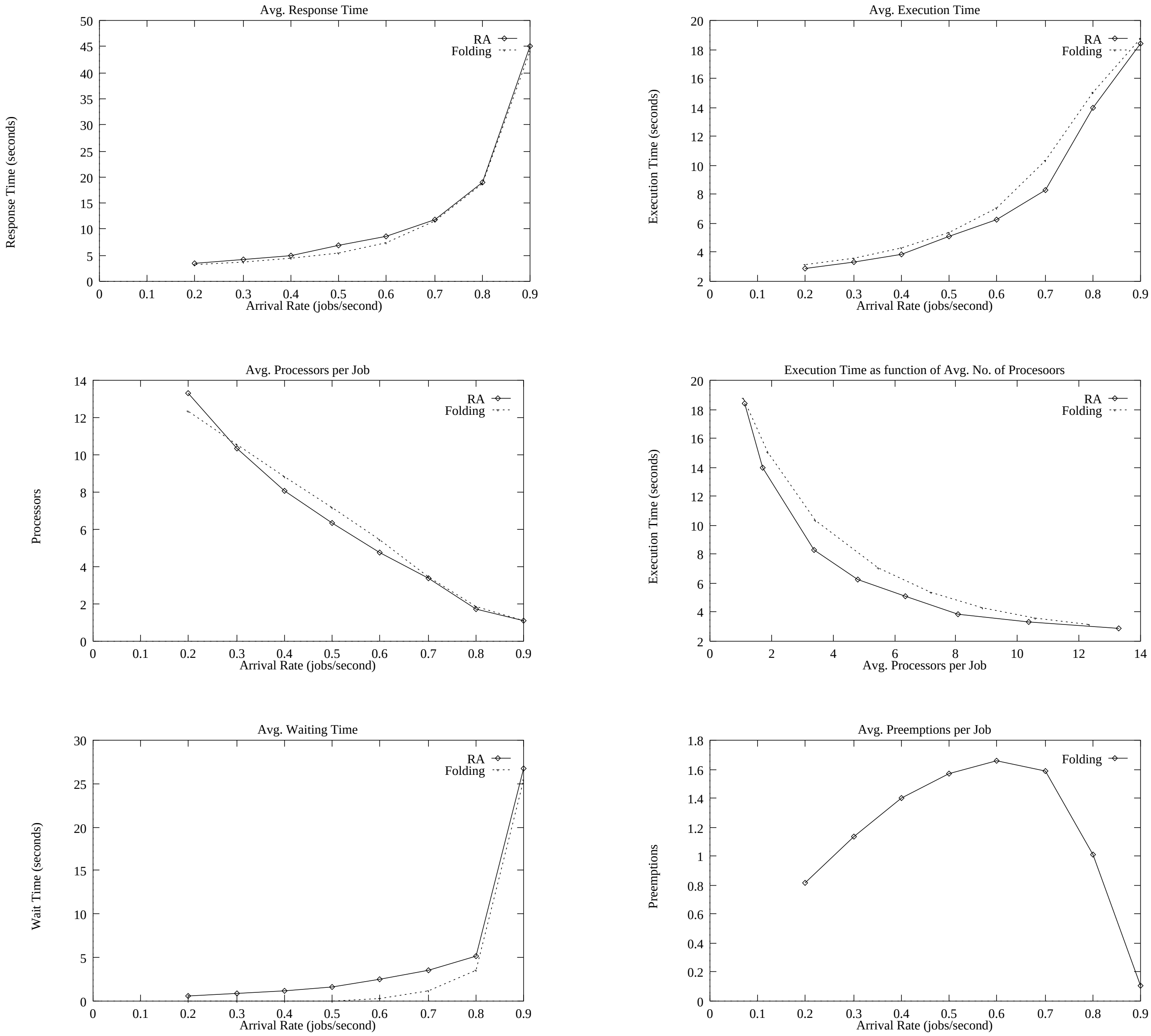

Figure 10: RA versus Folding: Conjugate Gradient Workload with a 300x300 Matrix

jobs are likely to have partitions of size 1, and these can not be preempted for folding any further. Thus, the average number of preemptions goes down at both higher and lower arrival rates.

Figure 11 shows the performance of the RA and the Folding policies when scheduling a conjugate gradient workload with a 500x500 matrix. All other parameters are identical to the 300x3000 case, except that a preemption requires redistribution of approximately 2 megabytes. The primary observations are:

- The Folding policy gives a better response time than RA across all arrival rates.
- The maximum improvement in the response time of Folding over RA is approx. 8%, achieved at relatively high arrival rates.
- The maximum improvement in the wait time of Folding over RA is approximately 99%, achieved at lower arrival rates.
- Other than at low arrival rates, Folding is able to allocate more processors to a job than RA.
- In general, the execution time is lower under RA than under Folding. The maximum difference is approximately 20%. However, the improved wait time under Folding compensates for this.
- The maximum number of preemptions occur at relatively high arrival rates, however, no more than 1.5 preemptions/job occur at any arrival rate.

It is seen from these results that as speedup curve gets more flat (i.e. 300x300 matrix as opposed to 500x500 matrix), the improvement in the response time under Folding as compared to RA becomes more pronounced. This can largely be attributed to the improved in waiting time under Folding as can be seen from the waiting time graphs in figures 10 and 11.

3.2.3 Performance under Multiclass Workload

Figure 12 compares the performance of the RA and the Folding policies when scheduling a two-class workload. Each class constitutes approximately 50% of the total workload. The first class consists of conjugate gradient programs with a matrix size of 300x300, while the second class has a matrix size of 500x500. In the event of preemption, the first class requires redistribution of approximately 0.72 megabytes of data while the second class requires redistribution of approximately 2 megabytes of data. The key observations are:

- The Folding policy exhibits a better response time than RA across all arrival rates.
- The maximum improvement in the response time is approximately 19%, achieved at relatively high arrival rates.
- The response time improvement under Folding is due to the significant gains in the waiting times of the jobs under Folding. The maximum improvement of approximately 99% is achieved at low arrival rates.

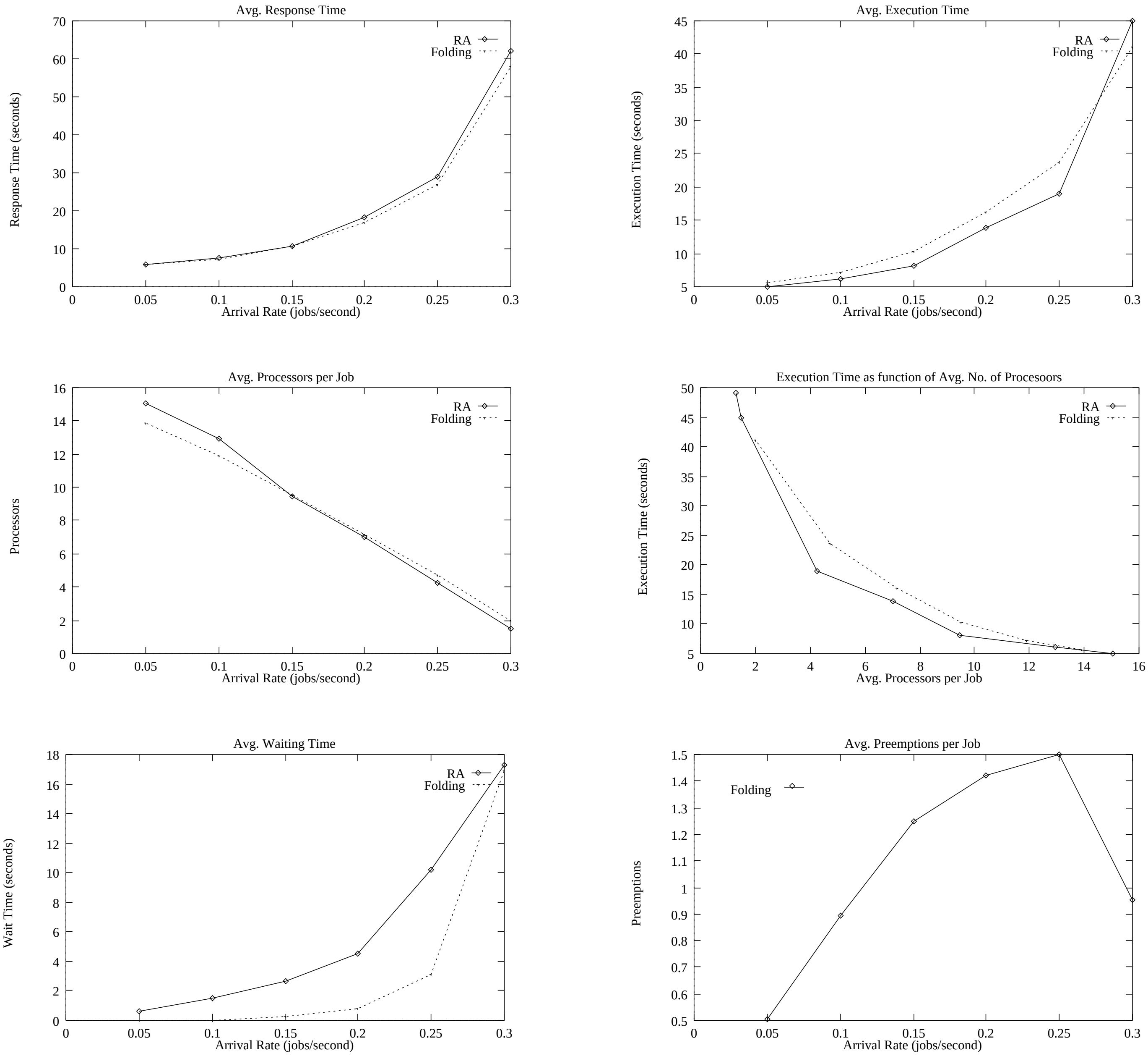

Figure 11: RA versus Folding: Conjugate Gradient Workload with 500x500 Matrix

- Except under very low arrival rates, Folding allocates more processors to a job than does RA.
- Execution times of jobs are higher under Folding than under RA. The maximum increase in the execution time under Folding as compared to RA is approximately 17% at moderate arrival rates.

Folding policy performs better than RA while scheduling a multiclass workload. Being a preemptive policy, Folding can adapt to transient changes in the workload flow better than RA. This behavior, as seen previously, is also observed when the workload demands are exponentially distributed rather than being constant.

3.3 Comparison under Higher Numbers of Processors

The general behavior observed in the previous sections remains consistent as the number of processors in the multiprocessor increase. Figure 13 presents comparison of RA and Folding policies with conjugate gradient workload with 500x500 matrix size. The workload demand is constant while the interarrival times have a negative exponential distribution. All the other parameters are the same as in the previous analysis (e.g. Section 3.3.2). It can be seen that better response times are obtained under Folding policy than under RA policy for most arrival rates. As the number of processors in a multiprocessor increases, good workload speedup characteristics are necessary to take advance of preemptive scheduling provided under Folding. It is possible that under Folding, a job may be unfolded to run on the entire multiprocessor. If the job has poor speedup characteristics, or if the job has negative speedup (likely for very large number of processors) characteristics, unfolding may prove detrimental to the overall performance.

4 Conclusions

This paper presents an experimental comparison between a non-preemptive, adaptive scheduling policy (RA) and a preemptive scheduling policy (Folding). Both the policies are implemented on a parallel computer with a message passing architecture. An open system model of the workload flow is implemented and the observed response time is the primary metric used for comparison. It is possible to draw following general conclusions from the experimental observations.

- It is possible to achieve good improvements in the average response time using a preemptive policy instead of an adaptive policy in many workload environments. This conclusion can be refined further as follows:
  - In most cases, the improvement in the response time is a direct result of the improvement in the time a job spends waiting in the ready queue.
  - As the job speedup curves become flatter, (i.e. the workloads don't scale as well), benefits of the Folding policy become more apparent. The Folding policy is able to dynamically capture processors from a currently executing job (where they are not providing any significant improvement in the execution time) and allocate them to a new job to take advantage of the efficient use of a few processors allocated to a newly arrived job. The

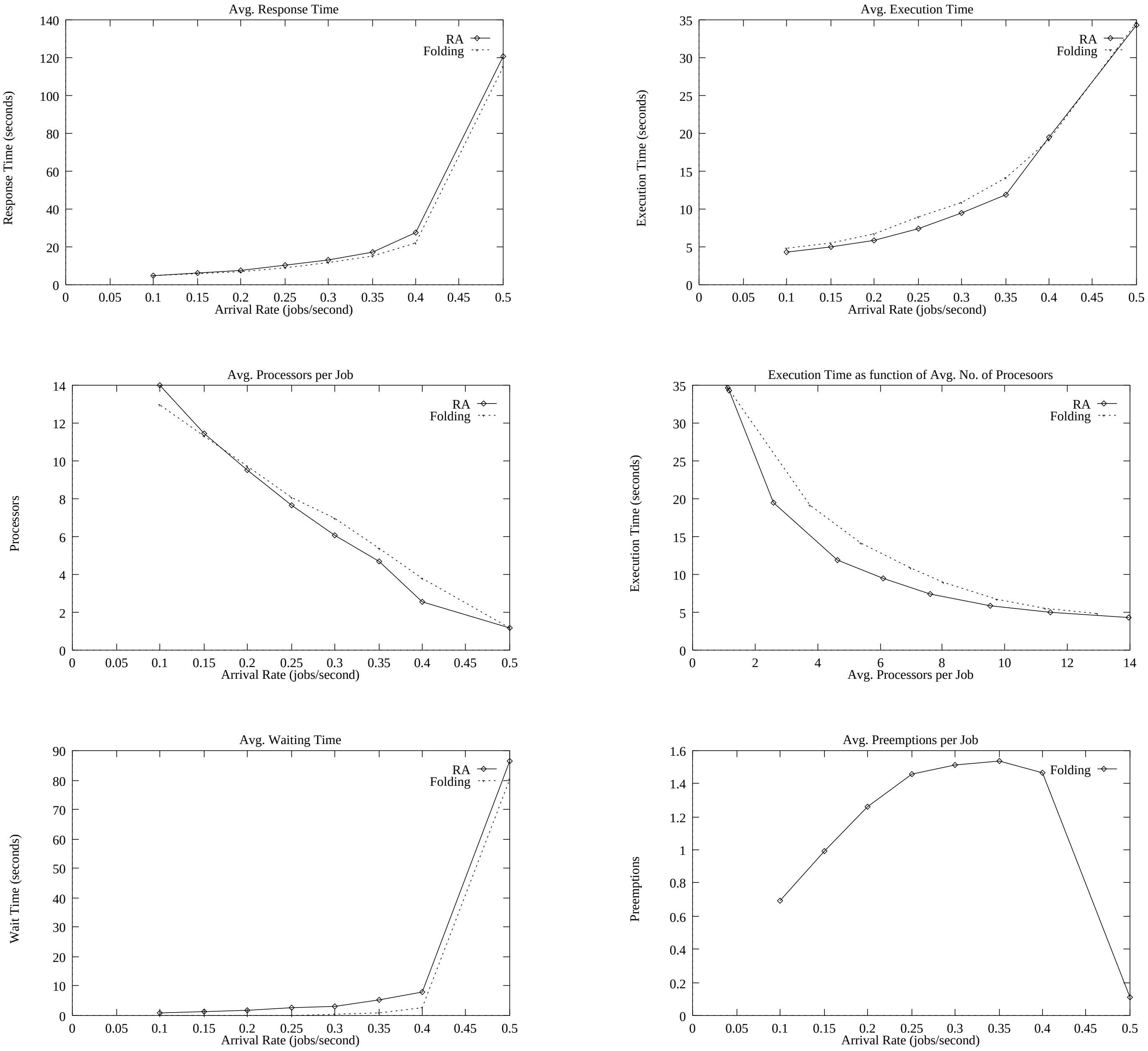

Figure 12: RA versus Folding: Multiclass Workload

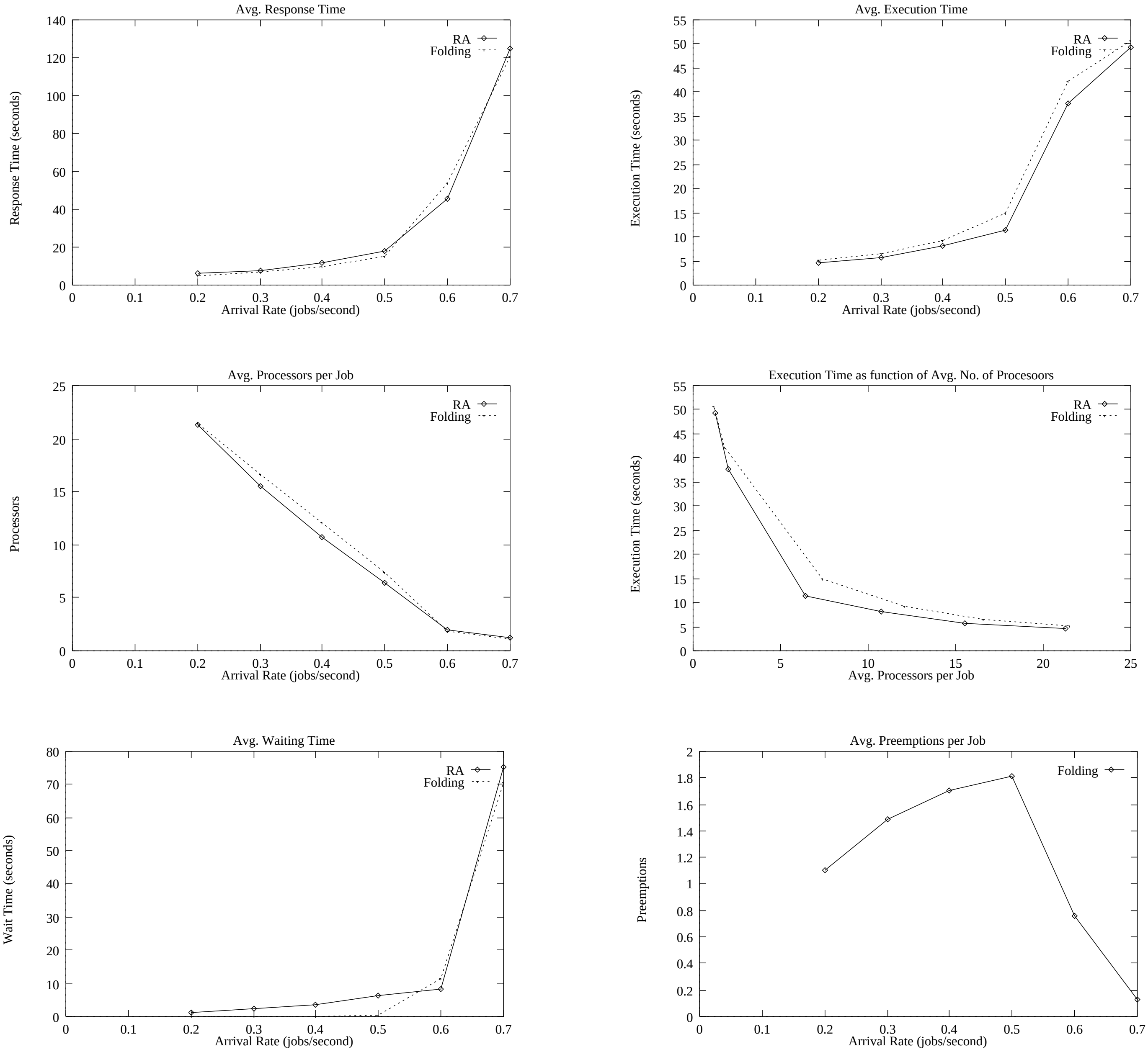

Figure 13: RA versus Folding: 32 Processors

non-preemptive policy may require the new arrivals to wait until at least one of the executing jobs completes.

  - As the workload flow becomes more *random* (exponential demand as opposed to constant) Folding performs better than RA. The preemptive nature of the Folding policy allows it to adapt quickly to transient changes common under exponentially distributed workload demands. The same is true in case of a multiclass workload.

- In almost all cases, Folding also allocates more processors to a job than does RA. It is interesting to note, however, that, except in few cases, this increase does not offset the preemption overhead of Folding. However, workloads with more linear speedup curves are able to take better advantage of the extra processors.

- The overhead incurred by the workload while executing under a preemptive policy (e.g. Folding) can be relatively high. Careful implementation of the workload-scheduler interface, and conservative processor allocation policies can help reduce this overhead.

- Although preemptive policies can improve performance, excessive use of preemptions to reallocate partitions can prove detrimental. This is clear from the way Equipartitioning suffers due to excessive preemptions.

It can thus be recommended that a non-preemptive, adaptive policy like RA be used to schedule jobs on a multiprocessor if the workload demand is constant, the workload is single-class and the workload has good speedup characteristics. On the other hand, a preemptive policy like Folding should be considered if the workload speedup characteristics are non-linear or if the workload demand is not constant (i.e either exponentially distributed or multi-class workload).

Possible directions for the future work include:

- Validating these results on a larger number of processors.

- Compare the policies under various arrival rates distributions, like constant, hyper-exponential and hypo-exponential.

- Compare the policies under various demand distribution as well as various multiclass workloads.

- Improve the scheduler-workload interface as well as the data redistribution routines to minimize the preemption overheads.

- Perform similar experiments on other computers with message passing architectures, with different interconnection networks and/or different message routing algorithms.

- Solve analytical models of small systems running preemptive scheduling policies and compare the results with experimental observations. Simulation models can also be created.

5 Acknowledgments

Any experimental project like this requires help and support from several people to be successful. First and foremost, the project would have been impossible without the support of Dr. Dowdy, who is not just a very good advisor, but is an even better *person.* Evgenia Smirni and Emilia Rosti provided invaluable help and suggestions throughout the project. They also provided the original code for the scheduler based on the RA policy. Several members of the PEG group provided a lot of support during the implementation phase. I also wish to thank the staff members of the Center for Computational Sciences at Oak Ridge National Labs, who made their computers available for this study and readily helped with any hardware or software problems.

References

[AMV93] R. Agrawal, R.K. Mansharamani, M.K. Vernon, "Response time bounds for parallel processor allocation policies," Technical Report # 1152, Computer Science Dept., Univeristy of Wisconsin, Madison, WI, June 1993.

[CMV94] S.-H. Chiang, R.K. Mansharamani, M.K. Vernon, "Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies," *Proc. ACM SIGMETRICS*, 1994, pp. 33-44.

[EZL89] D.L. Eager, J. Zahorjan, E.D. Lazowska, "Speedup versus efficiency in parallel systems," *IEEE Transactions on Computers*, Vol 38(3), March 1989, pp. 408-423.

[DCDP90] K. Dussa, B.M. Carlson, L.W. Dowdy, K.-H. Park, "Dynamic partitioning in a transputer environment," *Proc. ACM SIGMETRICS*, 1990, pp. 203-213.

[GST91] D. Ghosal, G. Serazzi, S.K. Tripathi, "Processor working set and its use in scheduling multiprocessor systems," *IEEE Transactions on Software Engineering*, Vol 17(5), May 1991, pp. 443-453.

[GTU91] A. Gupta, A. Tucker, S. Urushibara, "The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications," *Proc. ACM SIGMETRICS*, 1991, pp. 120-132.

[Int93] Intel Corporation, **Paragon OSF/1 User's Guide**, 1993.

[Klei75] L. Kleinrock, **Queueing Systems**, Vol 1, Wiley Interscience, 1975.

[LV90] S.T. Leutenegger, M.K. Vernon, "The performance of multiprogrammed multiprocessor scheduling policies," *Proc. ACM SIGMETRICS*, 1990, pp. 226-236.

[MEB88] S. Majumdar, D.L. Eager, R.B. Bunt, "Scheduling in multiprogrammed parallel systems," *Proc. ACM SIGMETRICS*, 1988, pp. 104-113.

[MEB91] S. Majumdar, D.L. Eager, R.B. Bunt, "Characterization of programs for scheduling in multiprogrammed parallel systems," *Performance Evaluation*, Vol 13(2), 1991, pp. 109-130.

[MVZ93] C. McCann, R. Vaswani, and J. Zahorjan, "A dynamic processor allocation policy for multiprogrammed shared memory multiprocessors," *ACM Transactions on Computer Systems*, Vol 11(2), February 1993, pp. 146-178.

[MZ94] C. McCann, J. Zahorjan, "Processor allocation policies for message-passing parallel computers," *Proc. ACM SIGMETRICS*, 1994, pp. 19-32.

[Oust82] J. Ousterhout, "Scheduling techniques for concurrent systems," *Proc. 3rd International Conference on Distributed Computing Systems*, 1982, pp. 22-30.

[PD89] K.-H. Park, L.W. Dowdy, "Dynamic partitioning of multiprocessor systems," *International Journal of Parallel Programming,* Vol 18(2), 1989, pp. 91-120.

[RSDSC94] E. Rosti, E. Smirni, L.W. Dowdy, G. Serazzi, B.M. Carlson, "Robust partitioning policies for multiprocessor systems," *Performance Evaluation*, Vol 19(2-3), March 1994, pp. 141-165.

[SST93] S.K. Setia, M.S. Squillante, S.K. Tripathi, "Processor scheduling in multiprogrammed, distributed memory parallel computers," *Proc. ACM SIGMETRICS*, 1993, pp. 158-170.

[Sev89] K.C. Sevcik, "Characterization of parallelism in applications and their use in scheduling," *Proc. ACM SIGMETRICS*, 1989, pp. 171-180.

[Sev94] K.C. Sevcik, "Application scheduling and processor allocation in multiprogrammed multiprocessors," *Performance Evaluation*, Vol 19(2-3), March 1994, pp. 107-140.

[SRDS93] E. Smirni, E. Rosti, L.W. Dowdy, G. Serazzi, "Evaluation of Multiprocessor Allocation Policies," Tech. Report, Computer Science Dept., Vanderbilt University, Nashville, TN, August 1993.

[TG89] A. Tucker, A. Gupta, "Process control and scheduling issues for multiprogrammed shared-memory multiprocessors," *Proc. of the 12th ACM Symposium on Operating Systems Principles*, 1989, pp. 159-166.

[ZM90] J. Zahorjan, C. McCann, "Processor scheduling in shared memory multiprocessors," *Proc. ACM SIGMETRICS*, 1990, pp. 214-225.

[ZB91] S. Zhou, T. Brecht, "Processor pool-based scheduling for large-scale NUMA multiprocessors," *Proc. ACM SIGMETRICS*, 1991, pp. 133-142.

[SRSDS94] E. Smirni E. Smirni, E. Rosti, G. Serazzi, L. W. Dowdy, K. C. Sevcik "Performance Gains from Leaving Idle Processors in Multiprocessor Systems" *Proc. International Conference on Parallel Processing*, 1995.

[BMSD95] J. Brehm, M. Madhukar, E. Smirni, L. W. Dowdy, "PerPreT - A performance prediction tool for massively parallel systems," To appear in *Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, September 1995.

[PICL] G. Geist, M. Heath, B. Peyton, P. Worley, "PICL, A PORTABLE INSTRUMENTED COMMUNICATION LIBRARY, C REFERENCE MANUAL" Oak Ridge National Laboretory.