A Robust Scheduling Strategy for Moldable Scheduling of Parallel Jobs

Sudha Srinivasan Savitha Krishnamoorthy P. Sadayappan

Department of Computer and Information Science
The Ohio State University, Columbus, OH-43210
{srinisud, savitha, saday}@cis.ohio-state.edu

Abstract

Moldable job scheduling has been proved to be effective compared to traditionaljob scheduling policies. It is based on the observation that most jobs submitted to a space-shared parallel system can actually reduce their response times if they were allowed to take any number of processors in a user-specified range. Previous approaches to scheduling of moldable jobs focused on when and how to choose the number of processors for a moldable job. Careful experimental evaluations show that these techniques are not robust. This paper proposes a new strategy for scheduling moldable jobs that outperforms not only the traditional rigid scheme, but also the previous moldable scheduling policies, by doing uniformly well under different load conditions and for jobs of different scalabilities.

1. Introduction

supercomputing centers require that a job's specification specify the number of processors that the job should execute on. However, most parallel applications are not hardwired to run on a particular number of processors. Usually, the user chooses the number of processors for a job that is expected to result in the best response time for the job. Hence there has been interest in another model of parallel job scheduling called moldable job scheduling. A moldable job is one which is submitted with multiple alternate processor choices and corresponding estimated run times. Such a scenario allows the system to select a processor choice for each job and schedule it so that the response time for the user is minimized.

The simplest approach of scheduling jobs on a First-Come-First-Served (FCFS) basis has been shown to result in severe fragmentation and poor system utilization [8]. To overcome these problems, several backfilling policies [11, 15, 10, 12] have been proposed and have been implemented in several production schedulers [7]. There are two common approaches to backfilling - conservative and aggressive (or EASY) [11, 12]. In conservative backfill, every job is given a reservation when it enters the system. A later arriving job is moved forward in the queue only if it can use idle unreserved processors without affecting the already reserved jobs. In aggressive backfilling, only one job has a reservation at any point in time. A later arriving job is allowed to leap forward only if it does not delay the reserved job.

Several policies for scheduling rigid jobs have evolved over the recent times, beginning with a simple First-Come-First-Served policy [5] to more advanced policies like multiple-queue backfilling policies [9] and selective reservation with backfilling [13]. Recently, there have been a few studies addressing a moldable job scheduling model [3, 14]

Cirne [3] proposed a moldable job scheduling scheme in which an application level scheduler makes this choice for the user before job submission. In [14], an assessment was made of the benefits of leaving this choice to the scheduler, which could delay the selection of processor count and use more dynamic information about the state of the system. Two schemes were proposed, based on a 'fairshare' allocation of processors and a 'lazy' selection of partition size for each job, and were shown to perform better than the submit-time selection strategy originally proposed by Cirne. However, when we carried out a more extensive evaluation of these schemes under varying conditions of load and job scalability, we found that they were not very robust. Where as performance improved over the rigid scheduling schemes under many circumstances, there were situations under which performance degraded quite significantly for some categories of jobs. In this paper, we propose enhancements to the moldable scheduling strategy that render it more robust under different load and scalability characteristics of jobs.

The rest of this paper is organized as follows: In Section 2, we provide the background and workload characteristics pertinent to the rest of the paper. Section 3 describes and evaluates the previously proposed approaches to moldable scheduling of jobs. In Section 4, we propose and evaluate three strategies to overcome the shortcomings of the previous approaches. In Section 5, we evaluate the performance of a combined strategy that incorporates the three proposed mechanisms. Section 6 describes the evaluation of the new scheme using empirically measured scalability profiles for NAS Parallel Benchmark applications. Section 7 concludes the paper.

Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'03)
0-7695-2066-9/03 $17.00 © 2003 IEEE

2. Background and Workload Characterization

Since the moldable job model developed by Downey [4] is used for most of our experiments, it is first discussed in this section. We also describe the workload parameters, approach to moldable workload generation, performance metrics for comparisons and our criteria for robustness.

2.1. Downey's Moldable Job Model

The speedup of a job on n processors is defined as the ratio of the job's run time on a single processor to the job's run time on n processors:

$$S(n) = L/T(n) \tag{1}$$

where S is the speedup function, L is the *effective* sequential run time and $T(n)$ is the run time of the job on n processors. Downey's speedup model [4] characterizes a job by two parameters:

- σ ($sigma$) is an approximation of the coefficient of variance in parallelism within the job. It determines how close to linear the speedup is. A value of 0 indicates linear speedup and higher values indicate greater deviation from the linear curve.
- A denotes the average parallelism of a job and is a measure of the maximum speedup that the job can achieve.

Downey's speedup function is defined as follows:

$$S(n, A, \sigma) = \begin{cases} \frac{An}{A + \sigma(n-1)/2} & (\sigma \leq 1) \wedge (1 \leq n \leq A) \\ \frac{An}{\sigma(A - 1/2) + n(1 - \sigma/2)} & (\sigma \leq 1) \wedge (A \leq n \leq 2A - 1) \\ A & (\sigma \leq 1) \wedge (n \geq 2A - 1) \\ \frac{nA(\sigma+1)}{\sigma(n + A - 1) + A} & (\sigma \geq 1) \wedge (1 \leq n \leq A + A\sigma - \sigma) \\ A & (\sigma \geq 1) \wedge (n \geq A + A\sigma - \sigma) \end{cases} \tag{2}$$

Three parameters were varied to test the robustness of the scheduling strategies:

1. Load factor (LF) - This represents the percentage of the actual job load used for simulation . We used LF values of 100, 125 and 150 for our experiments. At $LF = 150$, the utilization of the system is as high as about 95% under a rigid conservative backfilling policy and the system is quite saturated. Hence higher values of LF are not considered.
2. Sigma (σ) - It characterizes the scalability of a job. A value of 0.0 indicates perfect scalability and higher values indicate poor scalability. We varied σ from 0.0 to 2.0 in our experiments, to represent the common scalability characteristics of most parallel jobs [3].
3. Range factor (RF) - This represents the range of processors specified by the user. If the total number of processors in the system is n, and the number of processors requested for a job is p in the job log, a range factor of RF is used to calculate the minimum and maximum of the user specified range as follows:

$$min = ((1 - \frac{1}{RF}) * p) + 1$$

$$max = \frac{(n-p)}{RF} + p$$

We experimented with range factors of 1, 2, 3 and 4. Higher values of range factors produce results very close to a range factor of 4. We found the values of 1, 2, 3 and 4 to be sufficient to enumerate the behavior of various schemes under different conditions and hence higher range factors are not reported.

2.2. Moldable Workload Generation

We used trace based simulation to evaluate the various schemes using a subset of 5000 jobs from the SDSC (San Diego Supercomputer Center) workload log and a subset of 5000 jobs from the CTC (Cornell Theory Center) workload log from Feitelson's archive [6]. Because of the similarities found between using both the logs under various schemes, due to space limitations, we present the results of our proposed strategy for the SDSC log, along with a subset of the results for the CTC log.

A real-world job log contains information about scheduling of jobs submitted to a supercomputer over a period of time. For each job scheduled on the system, the information found in a job log generally includes the job ID, submission time, start time, completion time, allocated number of processors, requested maximum time, memory requirements, used CPU hours, etc.

In evaluating moldable job scheduling strategies, in addition to the information available from rigid-scheduling job logs, the following additional information is needed for each job:

1. Range of processor requests.
2. Estimated execution times corresponding to the processor requests.
3. Scalability Information.

We perform two kinds of experiments to measure the performance with our schemes. In the first kind, which is mainly used for testing various extremities and for analyzing improvements, we assume identical scalability (σ) and range factor (RF) for all jobs in any single run. In the second kind of experiment, we assume the scalability of jobs to correspond to the scalability of one of the applications from the NAS Parallel Benchmarks [1], and also assume the ranges of processor requests to correspond to the available experimental data for the NAS benchmarks.

For the first kind of experiment, we use the Downey model [4] to determine the speedup and the execution time for a job for any desired processor-request size. Cirne [2, 3] also used Downey's model of job scalability, but used statistical distributions to determine scalabilities and user-specified ranges on a synthetic workload.

Every job in the job log is assumed to have an average parallelism (A) equal to the maximum of an assumed user-specified range, and an assumed value of σ. From these, the speedup on n processors is found using Downey's model. We can then find the *effective* sequential run time

$L = T(n) * S(n)$. From L, the execution time on x processors, denoted $T(x)$, can be found using the formula:

$$T(x) = L/S(x, A, \sigma)$$

It should be noted that though all jobs have an effective sequential time, they might not be actually able to run sequentially because of several factors like memory requirements, algorithmic constraints and amount of parallelism [3]. For the second kind of experiments in which NAS Parallel Benchmark applications are involved, interpolation is used to directly find the run time corresponding to any particular processor request in the specified range. Hence in this case, the sequential run times are never computed.

2.3. Performance Metric

The average turnaround time of all the jobs in the system is not an adequate measure of performance of a user-centric scheduling strategy since there could be significant differences in the relative performance for different categories of jobs. An ideal moldable scheduling scheme should perform better than or comparable to traditional scheduling schemes for jobs of different categories. Hence we categorize the jobs based on their workload-based weights (i.e., the product of their requested processors and actual run time in seconds) and analyze the average turnaround time for jobs in each category, in addition to the overall turnaround time.

We are particularly interested in assessing the *robustness* of moldable scheduling schemes, i.e. that the achieved response time is better than or comparable to the rigid conservative backfilling scheme for jobs of different categories, without significant degradation to any category, for different values of the three parameters defined previously (LF, RF and σ).

3. Previous Schemes

In this section, we describe Cirne's greedy strategy [3] and the two fairshare strategies proposed in [14]. The following is a brief description of these three strategies:

1. *Submit-time greedy strategy* - In this scheme [3], every job is allowed a range of processor choices, specified by the user. Among these partition sizes, the one resulting in the best turnaround time for the job is chosen. This choice is made at the time of job submission and is not subsequently changed.
2. *Submit-time fair share strategy* - This scheme [14] is similar to the submit-time greedy scheme but the maximum fraction of the total system processors allowed for a job is calculated based on the following formula:

$$Weight\ fraction\ of\ job\ i = \frac{Weight\ of\ job\ i}{\sum_{all\ jobs} Weight\ of\ job\ currently\ in\ system}$$

   where $Weight$ refers to the $effective$ sequential run time of a jobs, and all currently running and queued jobs in the system are considered. In [14], an additional multiplicative weight-factor (WF) of 2 is used to avoid over-restriction of this value for jobs. A limit is imposed on the maximum percentage of the system's processors any job can be assigned. This factor is called gap-factor and has been taken to be 90% in [14].
3. *Schedule-time aggressive fair share strategy* - This scheme [14] is a variant of the submit-time fair share strategy in that the choice of partition size for a job is deferred until its start time and aggressive backfilling is used instead of conservative backfilling.

3.1. Performance of Fair-share Schemes

Since the greedy strategy has already been shown to be ineffective in [14], we restrict our analysis in this section to the two fair-share strategies. Figures 1 and 2 show the percentage change in the turnaround times of jobs in different categories and also the percentage change in the overall turnaround time obtained for the two schemes, with respect to the rigid conservative backfilling strategy, using the CTC log.

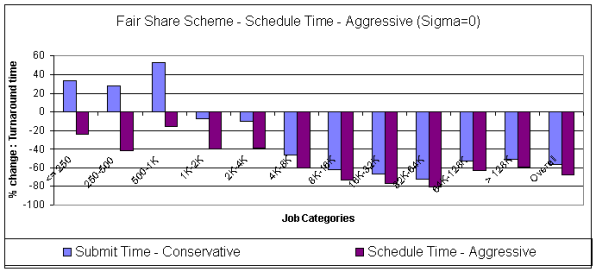

Figure 1. previous schemes w.r.t rigid conservative backfilling strategy, $LF = 100$, $\sigma = 0$ and $RF = 1$ (CTC log).

Figure 1 shows that under perfect scalability (sigma = 0), the overall turnaround time is better compared to the rigid scheme, but several category-wise turnaround times suffer greatly for the submit-time conservative fair-share strategy. But schedule-time aggressive performs better than the rigid scheme for all categories. However, with a sigma of 1 (Figure 2), although the overall turnaround time improves, category-wise turnaround times suffer greatly for both the schemes.

Figure 3 shows the results for the two fair-share schemes when SDSC log is used with $LF = 125$, $RF = 1$ and $\sigma = 2$. These clearly show the inefficiency of the schemes at high loads and poor scalabilities of jobs.

With a range factor of 1 when every job is allowed to take any number of processors from 1 to total system processors, a perfect fair share strategy attempts to split the available number of processors between all jobs. This indirectly tries to make all queued-up jobs finish at the same time. When a weight-factor of 2 is used as in strategies B and C,

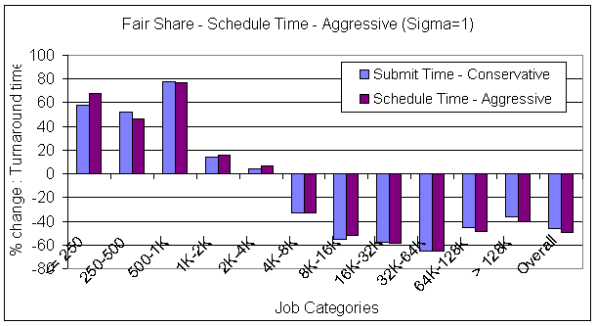

Figure 2. previous schemes w.r.t rigid conservative backfilling strategy for $LF = 100$, $\sigma = 1$ and $RF = 1$ (CTC log).

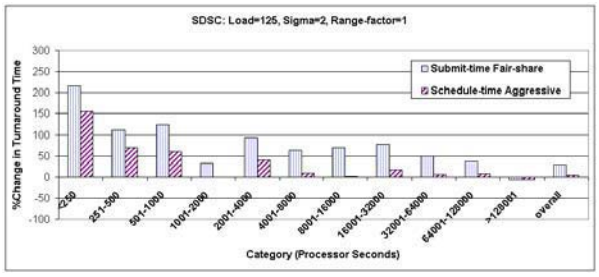

Figure 3. previous schemes w.r.t rigid conservative backfilling strategy $LF = 125$, $\sigma = 2$ and $RF = 1$ (SDSC log).

the large jobs tend to expand even further and take up more processors. Though such a behavior does not affect the performance of any category of jobs when σ=0, jobs in lower categories are affected when σ=1. This is because the large and medium jobs now take up even more time than before because of their poorer scalability. This results in huge delay for other jobs, especially those in the lower categories which now wait for a longer time. Hence strategies B and C suffer from poor performance for jobs in the light and medium categories.

For higher values of σ and some higher range-factors, there is even further deterioration in performance. The conclusion is that though the previously proposed strategies perform well under certain conditions, they suffer from poor category-wise performance under some other conditions or even poor overall performance in some other cases. For the purpose of comparisons in the rest of this paper, we restrict to just the schedule-time aggressive fair-share strategy since it gives the best performance among the three schemes.

4. Schedule-time Aggressive Fair-share Strategy

In this section, we propose strategies to improve the performance of the schedule-time aggressive fair-share scheme. In the rest of this paper, we use the schedule-time aggressive fair share strategy with a weight factor of 1 and number of processor choices equal to 12, as a base strategy for all comparisons.

4.1. Effect of Increasing the Number of partition sizes

We first consider the effect of the number of partition choices on the effectiveness of the schedule-time aggressive fair-share scheme. Evaluating all possible partition sizes for each job directly would be very expensive. However, assuming a monotonic function of runtime as a function of the number of processors, it is not necessary to evaluate all possible partition sizes, as discussed below.

The scheduler keeps track of the information about all the free-time blocks available. This forms the core part of the scheduler because it is used for all scheduling related activities. Each free-time block is represented as a triplet of the start time, end time and the number of processors available during the period between the start time and end time. For example, the free time list might consist of the following triplets: (100, 1000, 12), (1000, 1500, 20), (1500, 2000, 2), indicating that 12 processors are available between time=100 and time=1000, 20 processors between time=1000 and time=1500 etc.

A profile-based allocation scheme works as follows: To evaluate all possible partition sizes, the free-time blocks are considered starting from the first free-time block. In the example provided above, first a processor choice of 12 is considered and the turnaround time determined. Then the processor choice 20 is considered and evaluated. If any processor choice lies below the user-specified minimum, it is ignored and if it lies above the user-specified maximum, it is truncated to the maximum limit. The best turnaround time is kept track of at every stage and the process continues till we hit a block whose start time exceeds the best completion time seen so far. Thus the partition size which gives the best turnaround time is found. It is not necessary to evaluate the turnaround time for any other partition sizes than those corresponding to the time blocks in the schedule since they will either be infeasible or sub-optimal.

We found that when the number of processor choices is 12, large jobs not only conveniently find a good choice for the number of processors, but also tend to leave at least some holes in the schedule for jobs in lower categories to backfill. But when the full range of processor choices is evaluated, the large jobs expand and take up as many system processors as allowed by their fair share limit and thus severely limit backfilling opportunities for jobs belonging to lower categories. It was generally found that evaluating all possible processor choices had a detrimental effect on the performance of lighter job categories, when compared to use of only twelve processor choices.

4.2. An Effective Fair-share Allocation Policy

As seen in the previous section, a strict fair-share allocation allocates processors to jobs in direct proportion of the job's weight (total processor-seconds needed). This has the effect of seeking to equalize the run-times of all jobs - a job with twice the processor-seconds as another would tend to receive twice as many processors. The results show that heavy jobs tend to benefit significantly while the performance of light jobs deteriorates. Another possible allocation policy is an equi-partition policy, to allocate each job the same number of processors, irrespective of the job's weight. This can be expected to benefit light jobs at the expense of heavy jobs. These two alternatives represent two extremes with respect to the time and processor dimension for moldable jobs: the weight-based fair-share policy tends to equalize the time dimension of all moldable jobs, while an equi-partition strategy tends to equalize the processor dimension of the moldable jobs. We therefore evaluated a policy that sought balance between the space (processor) and time dimensions. Given two jobs, with one having 4 times the weight of the other, the idea is to give the heavier job twice as many processors as the lighter job, with the consequence that it would also have twice the runtime as the lighter job. Thus relative to the lighter job, the heavier job is reshaped to spread out in a roughly equal manner along both the space and time dimensions. Such an allocation is achieved by the following *modified weight-fraction* formula:

$$weight\ fraction\ of\ job\ i = \frac{\sqrt{weight\ of\ job\ i}}{\sum \sqrt{weight\ of\ job\ currently\ in\ system}}$$

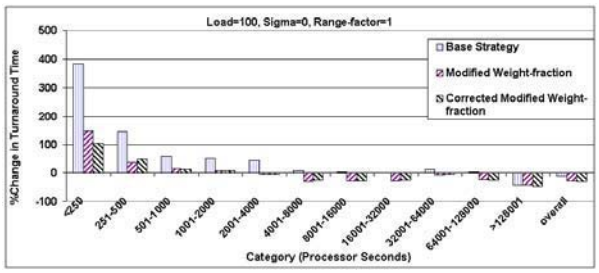

Figure 4. Performance of modified weight-fraction and corrected modified weight-fraction formulae, $LF = 100$, $RF = 1$ and $\sigma = 0$ (SDSC log).

Even though each sequential job can only take at most 1 processor, the modified weight-fraction formula would set aside a proportional amount of processors even for such jobs. To correct this inconsistency, we ignored the sequential jobs while calculating the weight fraction. The *corrected modified weight-fraction* formula is:

$$weight\ fraction\ of\ job\ i = \frac{\sqrt{weight\ of\ job\ i}}{\sum_{all jobs} \sqrt{weight\ of\ parallel\ jobs\ currently\ in\ system}}$$

Figure 4.2 shows the effect of using the corrected modified weight-fraction formula. Though the performance improvement with respect to the base strategy is high, only jobs belonging to certain categories improve with respect to the modified weight-fraction strategy. This is explained as follows: The plain modified weight-fraction formula includes the weights of sequential jobs and hence the fair-share limits of all the jobs are much lower. Though the corrected modified weight fraction formula allows a greater fair-share limit for all jobs and thus improves the overall performance, it works against jobs in some categories because of greedy occupancy of the extra processors by some heavy jobs.

4.3. Enhanced Reservation Policies

We propose two enhanced reservation policies, namely category-based reservation and Xfactor-based reservation and evaluate their individual and combined effects on the base strategy. The original fair-share formula is used throughout to enable direct comparison with the base strategy.

In a category-based reservation strategy, we partition the jobs into categories based on their weights and assign a reservation depth of one for each partition. Thus in each category of jobs, one queued job always has a reservation for starting at a future point in time. All further reservations or occupation of processors by other jobs should respect the constraint posed by the reserved jobs of all the categories. This is similar to the multi-queue backfilling policy proposed in [11].

We used 10 categories with exponentially increasing intervals. The results are shown in figure 5. Jobs from almost all categories improve with respect to the base strategy. Similar trends were seen for higher loads too.

To further reduce the possibility of starvation of certain jobs, we keep track of the Xfactor of each unreserved job in the system and reserve a job whenever its Xfactor has exceeded a threshold value k. The Xfactor for a job is calculated using its Effective Sequential Run Time (ESRT) as follows:

$$Xfactor = \frac{Queue\ time + ESRT}{ESRT}$$

We chose a value of 4 for k. This implies that a job is given a reservation as soon as its elapsed queue time exceeds 3 times its sequential wall-clock limit. Figure 5 evaluates the Xfactor-based reservation. When compared with the category-based reservation, performance improves only for jobs in lower categories and for most of the higher categories, the performance degrades. This is because the Xfactor-based reservation results in quicker reservations for light jobs. The Xfactor for large jobs increases at a very

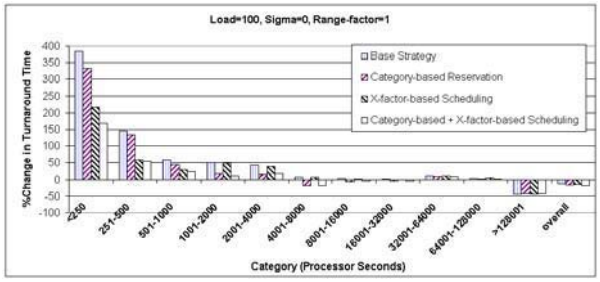

Figure 5. Evaluation of the individual and combined effects of the enhanced reservation policies $LF = 100$, $RF = 1$ and $\sigma = 0$ (SDSC log).

low rate and hence they are mostly scheduled because of their position in queue and availability of processors. But in category-based reservation, the processors can be occupied by jobs from any category and hence sometimes light jobs starve and large jobs gain. Figure 5 evaluates Xfactor-based reservation in combination with category-based reservation. All categories except the largest categories improve compared to just using the category-based reservation. The lighter categories improve as they reach their threshold Xfactor quicker than the large jobs. The overall turnaround time for this combined strategy outperforms the two strategies taken separately.

5. A Combined Moldable Scheduling Strategy

As seen in the previous section, evaluating all the possible partition sizes for each job only degrades the overall performance. So we limited the number of partition choices to be 12. This seemed to represent a good balance between allowing sufficiently many choices for each job, but at the same time preventing complete elimination of backfill opportunities for light jobs in small holes in the schedule.

We evaluated a schedule-time aggressive fair-share strategy with the following features:

1. Using corrected modified weight-fraction instead of the strict fair-share formula
2. Using Xfactor-based and Category-based enhanced reservation policies.

The overall results show that the combined scheme performs consistently better than the rigid scheme. For space reasons, we present a subset of the results in this paper. The performance of the combined scheme for SDSC log and CTC log with $LF = 1$, $\sigma = 0$ for different values of range-factors are shown in figures 6 and 7. Both the overall and category-wise turnaround times improve with respect to the rigid scheme, except for a very slight deterioration for the lowest weight category for $RF = 1$.

The performance results for SDSC and CTC log with $LF = 125$ and $\sigma = 0$ for different values of range-factors

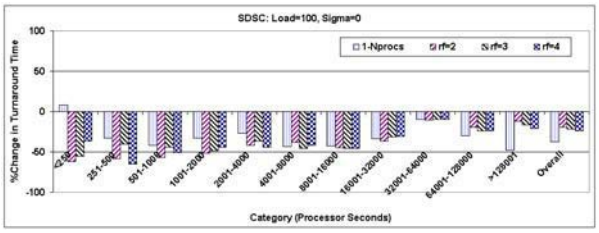

Figure 6. $\sigma = 0$ and $LF = 100$, SDSC log.

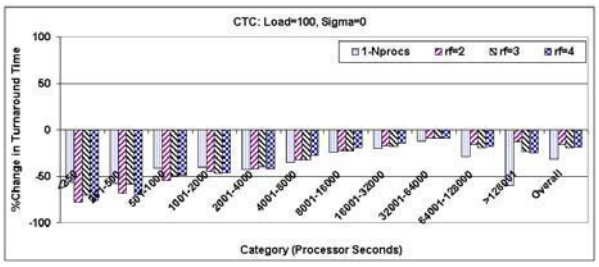

Figure 7. $\sigma = 0$ and $LF = 100$, CTC log.

are shown in figures 8 and 9 respectively and show similar improvements in turnaround time. At high loads, the combined scheme always provides significant improvement over using a rigid scheduling scheme.

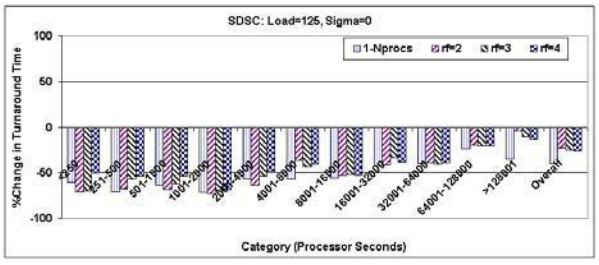

Figure 8. $\sigma = 0$ and $LF = 125$, SDSC log.

The performance results for SDSC and CTC log with $LF = 125$ and $\sigma = 1$ for different values of range-factors are shown in figures 10 and 11 and these too show uniformly good improvements in turnaround time. Thus, even when the scalabilities of jobs are poor, the combined scheme outperforms the rigid scheme for all categories of jobs.

6. Using NAS Parallel Benchmark Performance Data for Deriving the Scalability Characteristics of Jobs

In each of our previous experiments we assumed uniform scalability and range-factor for all the jobs. In a real sys-

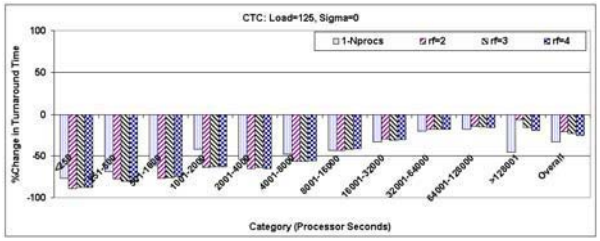

Figure 9. $\sigma = 0$ **and** $LF = 125$**, CTC log.**

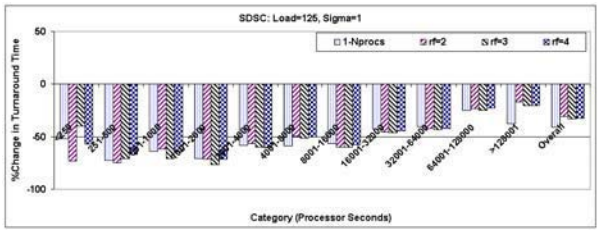

Figure 10. $\sigma = 1$ **and** $LF = 125$ **for SDSC log.**

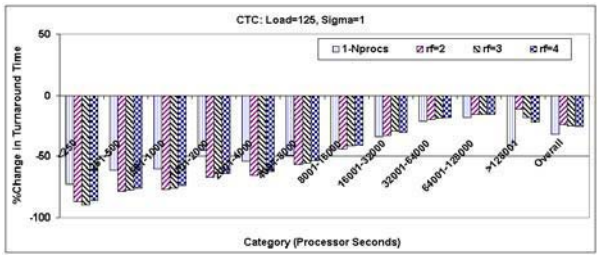

Figure 11. $\sigma = 1$ **and** $LF = 125$ **for CTC log.**

tem, this will clearly not be the case. Since we do not have any real moldable job log, a synthetic job log might seem like a practical option. But to better simulate a real moldable job log, we used the scalability characteristics of different applications in the NAS parallel benchmarks to characterize the scalability of jobs in a rigid job log. Each job in the log is randomly taken to exhibit the scalability characteristic of some application in NAS parallel benchmark. For a job that corresponds to some application in the NPB, we define the user-specified range to be the range of processors for which the execution times are available for the application. We then used the combined scheme to further verify its robustness.

Using the scalability characteristics of the results of NPB applications on the CRAY T3E for jobs in the SDSC log, experiments were performed for three load-factors. The results are shown in figures 12, 13 and 14. For all loads, the base strategy degrades some categories of jobs while the combined scheme performs better than or at least as well as the rigid scheme in all cases.

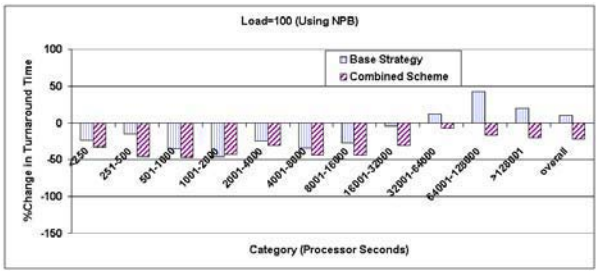

Figure 12. Performance of the combined scheme when $LF = 100$ **for SDSC log (Using NPB Performance Data).**

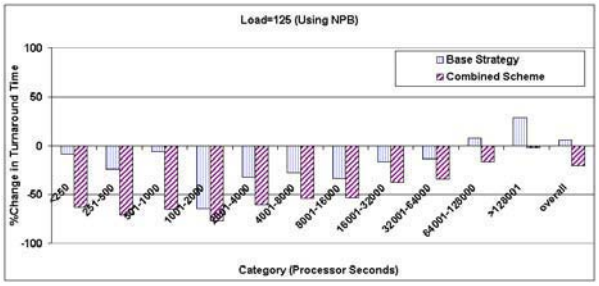

Figure 13. Performance of the combined scheme when $LF = 125$ **for SDSC log (Using NPB Performance Data).**

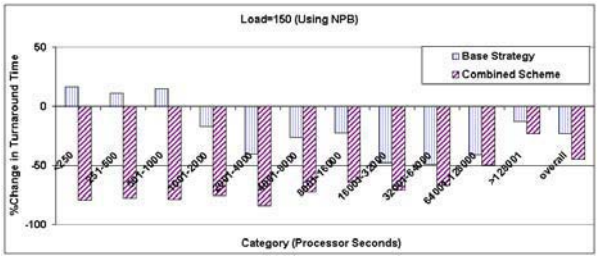

Figure 14. Performance of the combined scheme when $LF = 150$ **for SDSC log (Using NPB Performance Data).**

7. Conclusion

Moldable job scheduling schemes have been previously proposed and shown to improve average job slowdown and response time. But evaluation under different conditions of job scalability, system load, and extent of job moldability revealed that performance of some job classes could degrade significantly. In this paper, we proposed ways of improving moldable job scheduling, and showed that the en-

hancements improve performance as well as robustness.

References

[1] NAS parallel benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[2] W. Cirne and F. Berman. Adaptive selection of partition size for supercomputer requests. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 187–208, 2000.

[3] W. Cirne and F. Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. In *Journal of Parallel and Distributed Computing*, volume 10, pages 1571–1601, October 2002.

[4] A. B. Downey. A model for speedup of parallel programs. Technical Report CSD-97-933, 1997.

[5] A. B. Downey. Using Queue Time Predictions for Processor Allocation. In *3rd Workshop on Job Scheduling Strategies for Parallel Processing*, April 1997.

[6] D. G. Feitelson. Logs of Real Parallel Workloads from Production Systems. http://www.cs.huji.ac.il/labs/parallel/workload/logs.html.

[7] D. Jackson, Q. Snell, and M. J. Clement. Core Algorithms of the Maui Scheduler. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–102, 2001.

[8] J. Krallmann, U. Schwiegelshohn, and R. Yahyapour. On the Design and Evaluation of Job Scheduling Algorithms. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 17–42, 1999.

[9] B. G. Lawson and E. Smirni. Multiple queue backfilling scheduling with priorities and reservations for parallel systems. In *Workshop on Job Scheduling Strategies for Parallel Processing*, 2002.

[10] D. Lifka. The ANL/IBM SP scheduling system. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, 1995.

[11] A. W. Mu'alem and D. G. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. In *IEEE Transactions on Parallel and Distributed Computing*, volume 12, pages 529–543, 2001.

[12] J. Skovira, W. Chan, H. Zhou, and D. Lifka. The EASY - LoadLeveler API Project. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 41–47, 1996.

[13] S. Srinivasan, R. Kettimuthu, V. Subramani, and P. Sadayappan. Selective Reservation Strategies for Backfill Job Scheduling. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, July 2002.

[14] S. Srinivasan, V.Subramani, R.Kettimuthu, P.Holenarsipur, and P.Sadayappan. Effective selection of partition sizes for moldable scheduling of parallel jobs. In *Proceedings of the International Conference on High Performance Computing*, 2002.

[15] D. Talby and D. Feitelson. Supporting Priorities and Improving Utilization of the IBM SP Scheduler using Slack-Based Backfilling. In *Proceedings of the 13th International Parallel Processing Symposium*, 1999.