Scheduling Resizable Parallel Applications

Rajesh Sudarsan[†] Calvin J. Ribbens

Department of Computer Science
Virginia Tech, Blacksburg, VA 24061-0106
E-mail: {sudarsar, ribbens}@vt.edu

Abstract

Most conventional parallel job schedulers only support static scheduling thereby restricting schedulers from being able to modify the number of processors allocated to parallel applications at runtime. The drawbacks of static scheduling can be overcome by using scheduling policies that can exploit dynamic resizability in distributed-memory parallel applications and a scheduler that supports these policies. The scheduler must be capable of adding and removing processors from a parallel application at runtime. This ability of a scheduler to resize parallel applications increases the possibilities for parallel schedulers to manage a large cluster. Our ReSHAPE framework includes an application scheduler that supports dynamic resizing of parallel applications. In this paper, we illustrate the impact of dynamic resizability on parallel scheduling. We propose and evaluate new scheduling policies made possible by our ReSHAPE framework. Experimental results show that these scheduling policies significantly improve individual application turn around time as well as overall cluster utilization.

1. Introduction

In the last few years, low cost commodity clusters have emerged as a viable alternative to mainstream supercomputer platforms. A typical size of a cluster may range from a few hundred to thousands of processor cores. Although the increased use of multicore nodes means that node-counts may rise more slowly, it is still the case that cluster schedulers and cluster applications have more and more processor cores to manage and exploit. As the sheer computational capacity of these high-end machines grows, the challenge of providing effective resource management grows as well—in both importance and difficulty. A fundamental problem with existing cluster job schedulers is that they are static, i.e., once a job is allocated a set of processors, it continues to use those processors until it finishes execution. Under static scheduling, jobs will not be scheduled for execution until the number of processors requested by that job are available. Even though techniques such as backfilling [4] and gang scheduling [3, 2] try to reduce the time spent by an job in the queue, it is common for jobs to be stuck in the queue because they require just a few more processors than are currently available. A more flexible and effective approach would support dynamic resource management and scheduling, where the set of processors allocated to jobs can be expanded or contracted at runtime. This is the focus of our research—dynamically reconfiguring or (*resizing*) of parallel applications.

Dynamic resizing can improve the utilization of clusters as well as an individual job's turn around time. A scheduler that supports dynamic resizing can squeeze a job that is stuck in the queue onto the processors that are available and possibly add more processors later. Alternatively, the scheduler can add unused processors to a job so that the job finishes earlier, thereby freeing up processors earlier for waiting jobs. Schedulers can also expand or contract the processor allocation for an already running application in order to accommodate higher priority jobs, or to meet a quality of service or advance reservation deadline. More ambitious scenarios are possible as well, where, for example, the scheduler gathers data about the performance of running applications in order to inform decisions about who should get extra processors or from whom processors should be harvested.

We have developed a software framework, *ReSHAPE*, to explore the potential benefits and challenges of dynamic resizing. In [8] we described the design and implementations of ReSHAPE and illustrated its potential for individual jobs and work loads. In this paper, we explore the potential for interesting and effective parallel scheduling techniques, given resizable applications and a framework such as ReSHAPE. We describe two

[†]The work of the first author was supported in part by NSF ITR grant CNS-0325534

978-1-4244-3750-4/09/$25.00 ©2009 IEEE

typical scheduling policies and explore a set of related scheduling scenarios and strategies. Depending upon the policy, the ReSHAPE scheduler decides which jobs to expand and which to contract. We evaluate the scenarios using a job trace and show that these policies significantly improve overall system utilization and application turn-around time.

Though most of the recent research on dynamic scheduling has focused on grid environments, a few researchers have focused on cluster scheduling. Weissman et al. [10] describe an application-aware job scheduler that dynamically controls resource allocation among concurrently executing jobs. The scheduler implements policies for adding or removing resources from jobs based on performance predictions from the Prophet system [9]. The authors present simulated results based on supercomputer workload traces. Cirne and Berman [1] describe the adaptive selection of partition size for an application using their AppLes application level scheduler. In their work, the application scheduler AppLeS selects the job with the least estimated turn-around time out of a set of moldable jobs, based on the current state of the parallel computer. Possible processor configurations are specified by the user, and the number of processors assigned to a job does not change after job-initiation time.

The rest of this paper is organized as follows. Section 2 briefly discusses the ReSHAPE framework, highlighting its components and capabilities. Section 3 describes scheduling policies for improving application execution turn-around time and overall system utilization, possible scenarios associated with these policies, and the strategies used to build these scenarios. Section 4 describes the experimental setup used to evaluate these scheduling policies and scenarios and their performance. Finally, Section 5 summarizes the contribution and identifies directions for future work.

2. ReSHAPE Framework

The architecture of the ReSHAPE framework, shown in Figure 1, consists of two main components. The first component is the application scheduling and monitoring module which schedules and monitors jobs and gathers performance data in order to make resizing decisions based on application performance, available system resources, resources allocated to other jobs in the system, and jobs waiting in the queue.The second component of the framework consists of a programming model for resizing applications. This includes a resizing library and an API for applications to communicate with the scheduler to send performance data and actuate resizing decisions. The resizing library includes algorithms for mapping processor topologies and redistributing data from one processor topology to another. ReSHAPE targets applications that are *homogeneous* in two important ways. First, our approach is best suited to applications where data and computations are relatively uniformly distributed across processors. Second, we assume that the application is iterative, with the amount of computation done in each iteration being roughly the same. While these assumptions do not hold for all large-scale applications, they do hold for a significant number of large-scale scientific simulations. The application

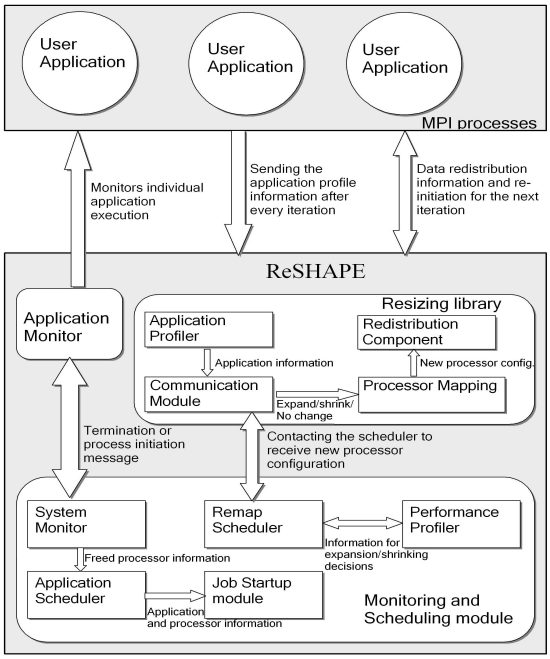


Figure 1. Architecture of ReSHAPE

scheduling and monitoring module includes five components, each executed by a separate thread. The different components are System Monitor, Application Scheduler, Job Startup, Remap Scheduler, and Performance Profiler. We describe each component in turn, detailing their capabilities.

System Monitor. An application monitor is instantiated on every compute node to monitor the status of an application executing on the node and report the status back to the System Monitor. If an application fails due to an internal error or finishes its execution successfully, the application monitor sends a job error or a job end signal to the System Monitor. The System Monitor then deletes the job and recovers the application's resources. For each application, only the monitor running on the first node of its processor set communicates with the System Monitor.

Application Scheduler. An application is submitted to the scheduler for execution using a command line submission process. The scheduler enqueues the job and

waits for the requested number of processors to become available. As soon as the resources become available, the scheduler selects the compute nodes, marks them as unavailable in the resource pool, and sends a signal to the job startup thread to begin execution. Different scheduling policies implemented in ReSHAPE are described in detail in Section 3.

Job Startup. Once the Application Scheduler allocates the requested number of processors to a job, the job startup thread initiates an application startup process on the set of processors assigned to the job. The startup thread sends job start information to the application monitor executing on the first node of the allocated set. The application monitor sends job error or job completion messages back to the System Monitor.

Performance Profiler. At every resize point, the Remap Scheduler receives performance data from a resizable application. The performance data includes the number of processors used, time taken to complete the previous iteration, and the redistribution time for mapping the distributed data from one processor set to another, if any. The Performance Profiler maintains lists of the various processor sizes each application has run on and the performance of the application at each of those sizes. Note that applications can only contract to processor configurations on which they have previously run.

Remap Scheduler. The point between two subsequent iterations in an application is called a resize point. After each iteration, a resizable application contacts the Remap Scheduler with its latest iteration time. The Remap Scheduler contacts the Performance Profiler to retrieve information about the application's past performance and decides to expand or shrink the number of processors assigned to an application according to the scheduling policies enforced by the particular scheduler. The size and topology of the expanded processor set can be problem and application dependent. In our current implementation we require that the global data be equally distributable across the new processor set. Furthermore, at job submission time applications can indicate (in a simple configuration file) if they prefer a particular processor topology, e.g., a rectangular processor grid. In the case where applications prefer "nearly-square" topologies, additional processors are added to the smallest row or column of the existing topology.

The resizing library, API and the redistribution algorithms implemented in ReSHAPE are described in detail in Sudarsan and Ribbens [8, 7].

3. Scheduling with ReSHAPE

We use the term *scheduling policy* to refer to an abstract high-level objective that a scheduler strives to achieve when scheduling arriving jobs. For example, one scheduling policy might be to minimize individual application turn-around time while keeping overall system utilization as high as possible. Clearly, such a policy may not be achievable in a mathematically optimal sense. Rather, a policy simply gives an indication of the approach to scheduling used on a particular parallel resource. Given such high-level policy, a *scheduling scenario* defines a specific, concrete attempt at achieving the scheduling policy. A scenario defines a procedure that the scheduler is configured to follow in order to achieve the objectives dictated by the policy. In the context of the ReSHAPE framework for dynamically resizable applications, a scheduling scenario must answer three fundamental questions: when to resize a job, which jobs to resize, and which direction to resize (expand or contract). We use the term *scheduling strategies* to refer to specific underlying methods or algorithms, implemented to realize resizing decisions. These methods or strategies define whether a job should be expanded or contracted and by how much. For example, the ReSHAPE scheduler could use a strategy which selects those jobs for expansion that are predicted to have maximum benefit from an expansion. Similarly, a strategy for harvesting processors might be to choose those jobs that are expected to suffer the least impact from contraction. In summary, a scheduling policy can be implemented in multiple scenarios, each realized using a particular collection of strategies. In the following subsections we briefly describe some simple strategies and scenarios that are implemented in ReSHAPE and which we use to illustrate the power of ReSHAPE for parallel scheduling research and development.

3.1. Scheduling strategies

Scheduling strategies can be categorized into processor allocation and processor harvesting strategies. A processor allocation strategy decides which applications to expand and by how much whereas a processor harvesting strategy decides which application to contract and by how much. In our current implementation, all allocation strategies use a simple model to predict the performance of a given job on a candidate processor size where that job has not yet run. Data from previous iterations on smaller processor sizes is used to inform contraction decisions. This combination of predictive model and historical data is also used to predict the time an application will take to reach its next resize point. An application must be expanded a minimum number of times before it is considered as a candidate for a resizing strategy. The minimum number is indicated by *remap_window_size* and its value is set by the system administrator. The *expand potential* for an application at a resize point is calculated only after the application

has been resized *remap_window_size* times. The *expand potential* of an application at a resize point is calculated by fitting a polynomial curve to the performance values of that application at its last *remap_window_size* number of resize points and computing the slope of the curve at its last resize point. The larger the value of the expand potential, the greater the chances that the job will benefit from its next expansion. The minimum threshold value for an *expand potential* beyond which an application can be predicted to provide significant performance benefit is set as part of the scheduling policy. A job that has reached its *sweet spot* is not considered as a candidate for processor allocation strategy. (The *sweet spot* is an estimate of the processor count beyond which no performance improvement is realized for a given job.) However, an application can be contracted below its sweet spot.

ReSHAPE currently provides three processor allocation strategies—Max-benefit, FCFS-expand and Uniform-expand—and two processor harvesting policies—Least-impact and Fair-contract—defined below.

Max-benefit: In this strategy, idle processors are allocated to jobs arranged in descending order of their expand potential at their last resize point. In other words, allow a job to grow that is predicted to benefit most when expanded to its next possible processor size. Only those jobs whose expand potential is greater than the minimum threshold value are considered as candidates for expansion. If a job that arrives at its resize point does not find itself at the top the sorted list, then follow the steps listed below.

* For every job that has a higher expand potential than the current job and is expected to reach its resize point before the current job's next resize point, count the number of processors required to expand that job to its next possible larger size.
* If there are still sufficient idle processors remaining, after the above "pre-allocation" step, then assign processors to the current job, and expand.

FCFS-expand: In this strategy, jobs are expanded to their next possible processor size if sufficient idle processors are available. The jobs are serviced in the order they arrive at their resize point.

Uniform-expand: This strategy ensures that a job is expanded only if there are enough idle processors to expand all running jobs. At each resize point, a list is prepared with the number of processors required to expand all the running jobs to their next resize point. If there are enough idle processors to satisfy all the jobs, then the current job is allowed to expand to its next larger size.

Least-impact: In this processor harvesting strategy, jobs are contracted in the ascending order of their expected performance impact suffered due to contraction. At every resize point, a list is created to indicate the possible performance impact at the next resize point for all the jobs that are currently running. The list is traversed till the required number of processors can be freed. The current job is contracted to one of its previous resize point if it is one of the possible candidates in the traversed list, i.e., if it is encountered on the list before the total number of desired processors has been identified. The procedure is continued till the required number of processors are available or till all jobs have reached their starting processor configuration.

Fair-contract: The Fair-contract processor harvesting strategy contracts all jobs to their immediate previous processor configuration. The jobs are selected in the order they arrive at their resize point and are harvested an equal number of times. A job is not harvested further if it has reached its starting processor configuration. The strategy ensures that no job is contracted twice before all the jobs have been contracted at least once. A boolean variable, *contract*, is used to indicate whether or not a job is due for contraction. When a job contracts, it sets *contract* to true for all the jobs. The value is set to false only after a job is contracted to its previous configuration or if it reaches its starting processor configuration. If a job is scheduled for contraction and has its *contract* set to false, it is contracted only if all the jobs have their *contract* variable set to false. This prevents penalization of short running jobs. The procedure is continued till the required number of processors are available or till the jobs are contracted to their starting processor size.

3.2. Scheduling policies and scenarios

In order to illustrate the potential of ReSHAPE and dynamic resizing for effective cluster scheduling, and to begin studying the wide variety of policies and scenarios enabled by ReSHAPE, we define two typical scheduling policies, as follows. The first policy aims at improving an individual application's turn-around time and overall system utilization by favoring queued applications over running applications. In this policy, running applications are not allowed to expand until all the jobs in the queue have been scheduled. The second policy considered here aims to improve an individual application's turn-around time and overall cluster utilization by favoring running applications over queued applications. In this policy, running applications are favored over queued jobs and are allowed to expand to their next valid processor size, irrespective of the number of jobs waiting in the queue. It is important to note that ReSHAPE can support more sophisticated scheduling policies. To realize these two scheduling policies, we describe five different scheduling scenarios. The scenarios are implemented by com-

bining different scheduling strategies. Three of the scenarios aim to achieve the policy that favors queued jobs. They are Performance-based allocation (PBA-Q), Fair-share allocation (Fair-Q), FCFS with least impact resizing (FCFS-LI-Q). Two more scenarios aim to achieve the policy that favors running jobs. They are Greedy resizing (Greedy-R) and Fair-share resizing (Fair-R). All the scenarios (described in detail below) support backfilling as part of their queuing strategy. All jobs arriving at the queue are assumed to have equal priority. They are queued in a FCFS order and are scheduled if the requested number of processors becomes available.

Policy 1: Improve application turn-around time and system utilization - Favor queued applications:

Scenario 1: Performance-based allocation (PB-Q): In this scenario, jobs are expanded using the *Max-benefit* processor allocation strategy and contracted using the *Least-impact* harvesting strategy. The procedure followed to determine whether to expand or contract a job or to maintain its current processor size is detailed below.

* When a job at its resize point contacts the scheduler for remapping, check whether there are any queued jobs. If there are jobs waiting in the queue, then contract the current job if it is selected based on the *Least-impact* processor harvesting strategy.
* If there are queued jobs but the current job is not selected for contraction, then maintain the job's current processor size.
* If the first queued job cannot yet be scheduled using the processors harvested from the current job, then backfill as many waiting jobs as possible.
* Once the maximum number of queued jobs have been scheduled, check whether the application has already reached its sweet spot processor configuration. If it has, then maintain the current processor size for the job.
* If the application has not yet reached its sweet spot configuration, check whether the current job benefited from its previous expansion. If it did not then contract the job to its immediate previous configuration and record the application has reached its sweet spot.
* If there are no queued jobs and if the application benefited due to expansion at its last resize point, then expand the job using the *Max-benefit* processor allocation strategy.
* If the job cannot be expanded due to insufficient processors, then maintain the job's current processor size.

Scenario 2: Fair-share allocation (Fair-Q): In a Fair-share allocation scenario, jobs are expanded using the *Max-benefit* processor allocation strategy and contracted using the *Fair-contract* harvesting strategy. The procedure followed to determine whether to expand or contract a job or to maintain its current processor size is detailed below.

* When a job at its resize point contacts the scheduler for remapping, check whether there are any queued jobs. If there are jobs waiting in the queue, then see if the current job should be contracted based on the *Fair-contract* strategy.
* If the current job is already at its starting processor configuration, then maintain its current processor size.
* Backfill smaller jobs if the first queued job cannot be scheduled using the available idle processors.
* Once the maximum number of queued jobs have been scheduled, check whether the application has already reached its sweet spot processor configuration. If it has, then maintain the current processor size for the job.
* If the application has not yet reached its sweet spot configuration, then check whether the current job benefited from its previous expansion. If it did not, then contract the job to its immediate previous configuration and record that the application has reached its sweet spot configuration.
* If there are no queued jobs and if the application benefited due to expansion at its last resize point, then expand the jobs using the *Max-benefit* processor allocation strategy.
* If the job cannot be expanded due to insufficient processors, then maintain the job's current processor size.

Scenario 3: FCFS with least impact resizing (FCFS-LI-Q): The FCFS-LI-Q scenario uses the *FCFS-expand* processor allocation strategy to expand jobs and the *Least-impact* processor harvesting strategy to contract jobs. The scheduler follows the steps listed below to decide whether to expand or contract an application that arrives at its resize point.

* When a job at its resize point contacts the scheduler for remapping, check whether there are any queued jobs. If there are, then see if the current job should be contracted using the *Least-impact* strategy.

* If there are queued jobs but the current job is not selected for contraction, then maintain the job's current processor size.
* If the first queued job cannot be scheduled using the available idle processors, then backfill smaller jobs.
* Once the maximum number of queued jobs have been scheduled, check whether the application has already reached its sweet spot processor configuration. If it has, then maintain the current processor size for the job.
* If the application has not yet reached its sweet spot configuration, check whether the current job benefited from its previous expansion. If it did not then contract the job to its immediate previous configuration and indicate that the application has reached its sweet spot configuration.
* If there are no queued jobs and if the application benefited due to expansion at its last resize point, then expand the jobs using the *FCFS-expand* processor allocation strategy.
* If the job cannot be expanded due to insufficient processors, then maintain the job's current processor size.

Policy 2: Improve application turn-around time and system utilization - Favor running applications: The scenarios implemented in this policy do not consider the number of queued jobs in their resizing decision and expand jobs if enough processors are available. Schedulers implementing this policy will not contract running jobs to schedule queued jobs.

Scenario 1: Greedy Resizing (Greedy-R):

* When a job contacts the scheduler at its resize point, check whether the job benefited from expansion at its last resize point. If it did not, then contract the job to its previous processor size.
* If it benefited from previous expansion, then expand the job to its next possible processor size using the *FCFS-expand* processor allocation strategy.
* If the job cannot be expanded due to insufficient processors, then maintain the job's current processor size.
* Schedule queued jobs using backfill to use the idle processors available in the cluster.

Scenario 2: Fair-share resizing (Fair-R):

* When a job contacts the scheduler at its resize point, check whether the job benefited from expansion at its last resize point. If it did not, then contract the job to its previous processor size.
* If it benefited from expansion at its last resize point, then expand the job to its next possible processor size using the *Uniform-expand* processor allocation strategy.
* If the job cannot be expanded due to insufficient processors, then maintain the job's current processor size.
* Schedule queued jobs using backfill to use the idle processors available in the cluster.

4. Experiments and Results

This section presents experimental results to demonstrate the potential of dynamic resizing for parallel job scheduling The experiments were conducted on 50 nodes of a large cluster (Virginia Tech's System X). Each node has two 2.3 GHz PowerPC 970 processors and 4GB of main memory. Message passing was done using OpenMPI [6] over an Infiniband interconnection network. As mentioned above, we consider two broad scheduling policies, one that favors queued applications and one that favors running applications. Both policies seek to reduce job turn-around time while maintaining high system utilization. The policy favoring queued application is represented by three scenarios and the policy favoring running applications is represented by two scenarios, as described in the previous section.

We first compare the performance of one of these scenarios against a conventional scheduling scenario, namely static scheduling with backfill. We then compare all five ReSHAPE-enabled scheduling scenarios head-to-head, looking at their performance with respect to application turn around time and overall cluster utilization. The job mix used in the experiments consists of four applications from the NAS parallel benchmark suite [5]: LU, FT, CG and IS. We use class A and class B problem sizes for each benchmark, for a total of eight different jobs. We generate a workload of 50 jobs from these eight job instances, with each newly arriving job randomly selected with equal probability. The number of each job in the trace used in these experiments is listed in Table 1. The arrival time for each job in the workload was randomly determined using a uniform distribution between 0 and 30 seconds. Each job consists of twenty iterations of the particular NAS benchmark. ReSHAPE was allowed to resize any job to a processor size that is a power-of-2, i.e., 2, 4, 8, 16, 32 and 64. For these experiments no data was redistributed at resize points since the current ReSHAPE library does not include data redistribution routines for the particular data structures used by

Table 1. Job trace using NAS parallel benchmark applications.

Job type	No. of jobs	Starting procs	Exec time for 1 iteration (secs)
IS Class A	3	2	2.748
CG Class A	7	2	3.549
IS Class B	4	4	6.607
FT Class A	5	2	9.231
FT Class B	5	4	63.348
CG Class B	9	4	90.293
LU Class A	8	4	101.436
LU Class B	9	8	316.302

the NAS benchmarks, and because the focus of this paper is on job scheduling. Data distribution is considered in more detail in Sudarsan and Ribbens [8, 7].

4.1. Comparing with baseline policies

We consider static scheduling with backfill as the baseline scheduling policy for this experiment. We compare the performance of a simple ReSHAPE-enabled scheduling scenario, namely FCFS with least impact resizing (FCFS-LI-Q), with the baseline scheduling policy. The FCFS-LI-Q scenario favors queued applications, i.e., it does not allow running jobs to expand if there are jobs waiting in the queue. We evaluate the performance using three metrics — job completion time, job execution time and queue wait time. Queue wait time is the time spent by a job waiting in the queue before it is scheduled for execution. The time take to successfully execute a job is indicated by the job execution time. Job completion time is the sum of queue wait time and job execution time. Figure 3 shows that FCFS-LI-Q reduces the total job completion time, job execution time and queue wait time by 10.4%, 11.3% and 8.5%, respectively. Figure 2 compares the total job completion time for all the jobs of each problem size. The job completion times are normalized to 1 to demonstrate clearly the performance benefits of scheduling using FCFS-LI-Q compared to a conventional scheduling policy. IS class B and FT class A jobs show the maximum benefit, with an improvement of 38.1% and 30.5%, respectively. This is due primarily to significant reductions in queue wait time for several of these jobs. Since the FCFS-LI-Q scenario favors queued applications, most jobs were not allowed to expand more than once in this experiment.

4.2. Performance of ReSHAPE-enabled scenarios

We now compare the performance of all five new scenarios in a head-to-head fashion. We use four metrics

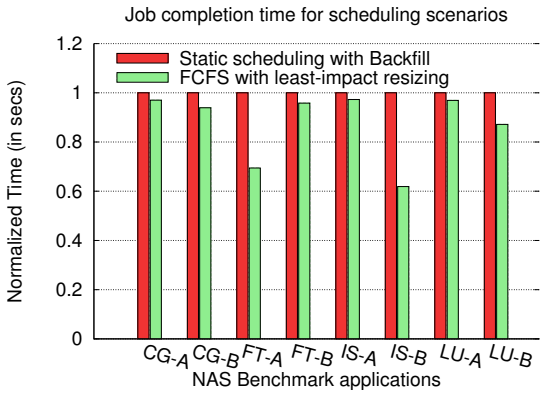


Figure 2. Comparing job completion time for individual applications executed with FCFS-LI-Q scheduling scenario and static scheduling policy

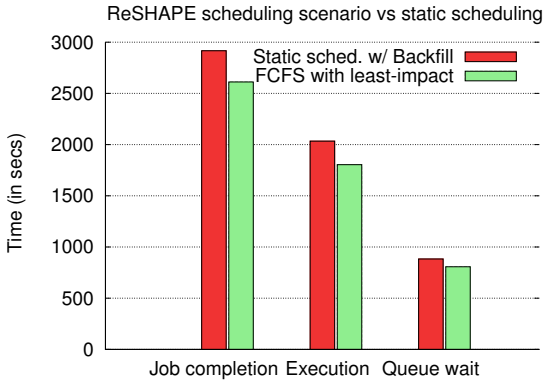
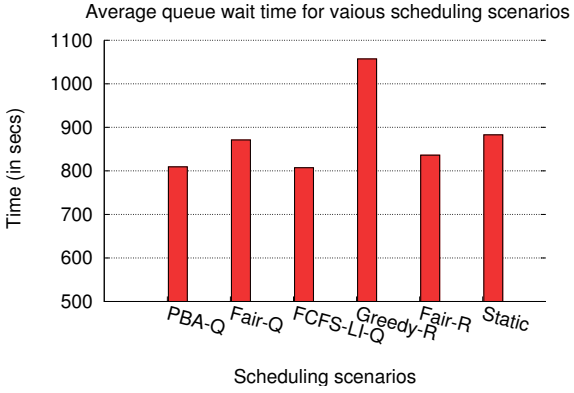


Figure 3. Comparing performance of FCFS-LI-Q scheduling scenario and static scheduling with backfill policy

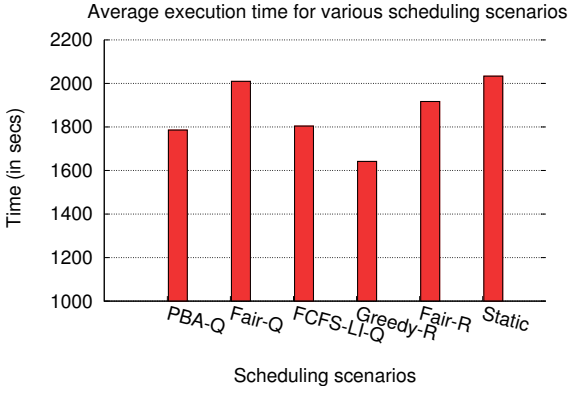
to evaluate performance: average job completion time, average execution time, average queue wait time, and overall system utilization.

Figure 4(a) shows the average queue wait time experienced by all fifty jobs when scheduled using the different scheduling scenarios. Applications scheduled using Greedy-R experienced the longest queue wait. This is because in the Greedy-R scenario, running applications are allowed to gain additional processors even when there are jobs waiting in the queue. Furthermore, available processors are allocated to running jobs as long as the performance model predicts any reduction in iteration time. In practice this aggressive awarding of processors could be throttled back by requiring that the performance model predict a performance improvement

from additional processors that exceeds some threshold. In the simple Greedy-R scenario used here however, queued jobs have to wait till running jobs finish execution and free sufficient processors to schedule the queued job. The average wait time for jobs scheduled using Greedy-R scenario is 1057 seconds. Fair-Q and Fair-R perform slightly better than static scheduling, with an average wait time of 871 seconds and 836 seconds respectively. Even though the Fair-R scenario favors running jobs, it allows jobs to expand only when sufficient processors are available to expand all the running jobs. Since is not possible at every resize point, the idle processors are used to schedule queued jobs at an earlier time. Applications scheduled using PBA-Q and FCFS-LI-Q scenarios experienced low average wait times of 809 seconds and 807 seconds respectively.



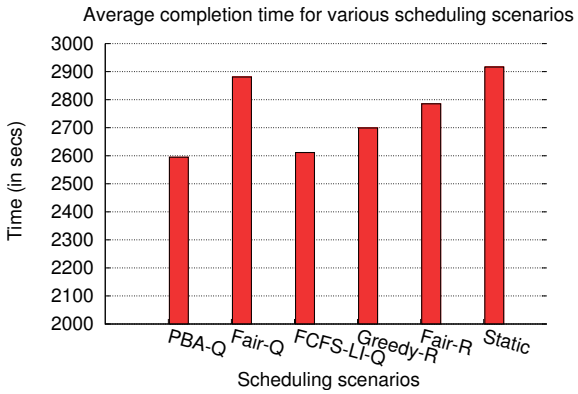
(a) Average wait time



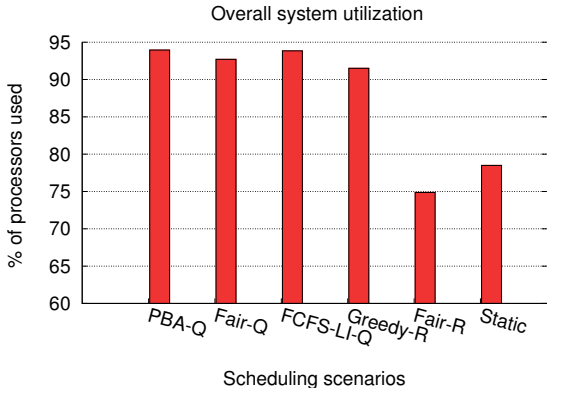
(b) Average execution time

Figure 4. Average wait time and average execution time for all the applications in the job trace

Figure 4(b) shows the average execution time for the applications in the trace. Not surprisingly, applications scheduled using the Greedy-R scenario had the lowest



(a) Average completion time



(b) System utilization

Figure 5. Average completion time and overall system utilization for all applications in job.

average execution time of 1642 seconds. Fair-R performed better than Fair-Q because Fair-R expands running applications before scheduling a queued job. In the case of Fair-Q, applications are contracted uniformly to accommodate a queued job, thereby increasing the average execution time. Applications scheduled with PBA-Q and FCFS-LI-Q have almost identical average execution times. Statically scheduled applications have the longest average execution time of 2034 seconds.

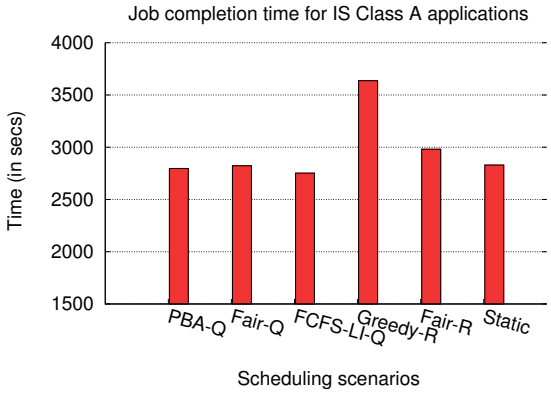
Figure 5(a) shows the average job completion time for the jobs in the trace. Jobs scheduled using Fair-Q had the highest average job completion time compared to other ReSHAPE scheduling scenarios due to its high average execution time. Only jobs scheduled using static scheduling had a higher average job completion time than Fair-Q. The Greedy-R scenario had a lower average job completion time than Fair-Q and Fair-R because of its low average execution time. PBA-Q and FCFS-LI-Q had the lowest average job completion times at 2595

seconds and 2611 seconds, respectively.

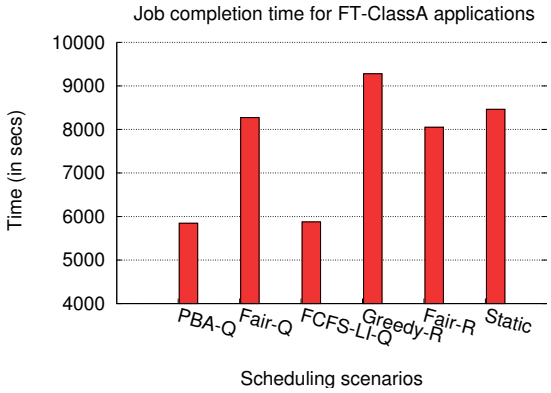
The overall cluster utilization is shown in Figure 5(b). The PBA-Q, Fair-Q, FCFS-LI-Q, and Greedy-R scenarios all had overall system utilization over 90%. In each of these scenarios, idle processors are used either to expand running applications or to schedule queued applications, thereby resulting in high system utilization. Fair-R had the lowest overall system utilization at 75%, even lower than the utilization for static scheduling at 78.5%. This is because Fair-R does not allow running applications to resize even if there are idle processors with no queued jobs unless there are sufficient idle processors to resize *all* running jobs. As a result of this extremely fair scheduling scenario, processors can sometimes remain idle, resulting in a low overall system utilization.

Turning now to performance results averaged across jobs of a particular type, Figure 6 and Figure 7 show job completion times for a representative subset of the eight problem/size combinations: IS class A, FT Class A, CG class B and LU class B applications scheduled with different scheduling scenarios. LU class B jobs have the largest problem size of the four types of jobs shown here. The larger the problem size, the longer the problem takes to complete its execution. IS class A jobs have the shortest running time. For short running jobs, jobs scheduled using Greedy-R and Fair-R scenarios have a longer job completion time compared to static scheduling. This is because the jobs spend most of their time waiting in the queue. PBA-Q and Fair-Q perform marginally better than static scheduling for the IS class A jobs. This application benefits from a smaller queue wait time but is unable to benefit from resizing. This is because the IS class A jobs finish their execution so quickly that they are rarely allowed to expand. IS class A has the shortest job completion time of 2752 seconds when scheduled with FCFS-LI-Q. When compared with PBA-Q and FCFS-LI-Q, the FT class A application has a higher job completion time with Fair-Q, Fair-R and Greedy-R due to high queue wait time. FT class A has the shortest running time of 2796 seconds with the PBA-Q scenario. CG class B and LU class B jobs have a high job completion time with both Fair-Q and Fair-R scenarios. A few LU class B and CG class B jobs arrive early and are scheduled immediately. Other LU and CG jobs must wait in the queue and can not be backfilled because they require long execution time. Since under policies favoring queued jobs they are not allowed to expand till there are no queued jobs, LU class B and CG class B jobs mostly are stuck at their starting processor size resulting in high job turnaround time. LU class B jobs benefited most when scheduled using the Greedy-R scenario. This is because the long running LU class B jobs are allowed to expand at almost every resize point, thus reducing their overall execution time.

The performance of the Fair-Q scenario deteriorate as the problem size increases. This scenario has the longest job completion time with LU class B applications, which are the largest problem considered. This is because longer running jobs are not easily backfilled and are executed at their starting processor configuration (or close to it) for most of their total execution time.



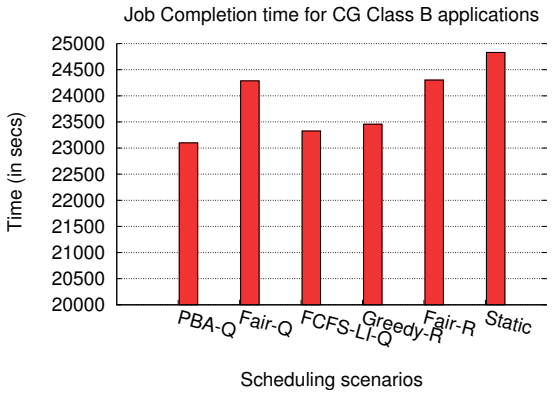
(a) Job completion time for IS Class A



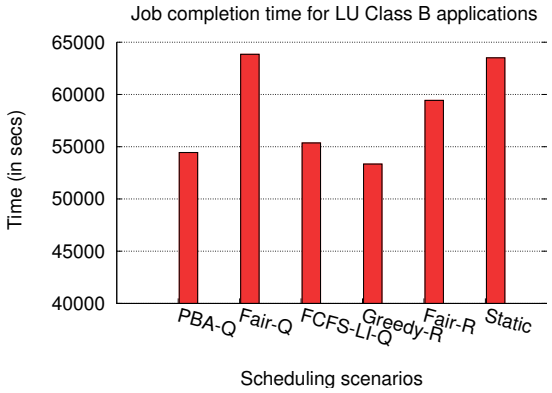
(b) Job completion time for FT Class A

Figure 6. Job completion time for FT and IS applications with different scheduling scenarios

On the other hand, the performance of Greedy-R improves as problem size increases. This scenario has the longest job completion time with IS class A jobs and the shortest completion time with LU class B jobs. The performance of PBA-Q and FCFS-LI-Q does not vary significantly as the problem size varies. These two scenarios rank as the top two best performing scenarios for most applications.



(a) Job completion time for CG Class B



(b) Job completion time for LU Class B

Figure 7. Job completion time for CG and LU applications with different scheduling scenarios

5. Conclusions and Future Work

In this paper we explore the potential of dynamically resizable applications for scheduling on large parallel clusters. We introduce new policies and strategies for scheduling resizable applications using the ReSHAPE framework. A scheduling policy is viewed as an abstract high-level objective that the scheduler strives to achieve. The scheduling scenarios provide a more concrete and implementable representation of the policy. Scheduling scenarios, in turn, are implemented using scheduling strategies which are methods responsible for actuating the resizing decisions. The scheduling policies discussed in this paper improve overall cluster utilization as well an individual application turn around time. The policies were evaluated using the five scenarios.

The current implementation of the ReSHAPE framework uses a common performance model for all resizable applications. The scheduling strategies use this predicted performance value in their resizing decisions for an application. More sophisticated prediction models and policies are certainly possible. Indeed, a primary motivation for ReSHAPE is to serve as a platform for research into more sophisticated resizing and scheduling strategies. Experimental results show that the proposed scheduling policies and scenarios outperform conventional scheduling policies with respect to average execution time, average job completion time and overall system utilization.

We are currently working on including more sophisticated scheduling policies in the ReSHAPE framework. Some of these policies include improving application turn-around time and system utilization using an adaptive partitioning strategy, scheduling priority-based applications and supporting advanced reservation services. We plan to make ReSHAPE more extensible so that job-specific performance models and new scheduling policies or strategies can be added to the framework.

References

[1] W. Cirne and F. Berman. Adaptive Selection of Partition Size for Supercomputer Requests. *Lecture Notes in Computer Science*, pages 187–208, 2000.

[2] D. Feitelson. Packing Schemes for Gang Scheduling. *Lecture Notes in Computer Science*, 1162:89–111, 1996.

[3] D. Feitelson and M. Jette. Improved Utilization and Responsiveness with Gang Scheduling. *Lecture Notes in Computer Science*, 1291:238–261, 1997.

[4] A. Mu'alem and D. Feitelson. Utilization, Predictability, Workloads, and User Runtime Estimates in Scheduling the IBM SP2 with Backfilling. *IEEE Transactions on Parallel and Distributed Systems*, pages 529–543, 2001.

[5] NAS Parallel Benchmarks (NPB-MPI 2.4). Available from: http://www.nas.nasa.gov/Software/NPB.

[6] Open MPI v1.25, 2008. Available from: http://www.open-mpi.org/.

[7] R. Sudarsan and C. Ribbens. Efficient Multidimensional Data Redistribution for Resizable Parallel Computations. In *Proceedings of the International Symposium of Parallel and Distributed Processing and Applications (ISPA '07)*, pages 182–194, Niagara falls, ON, Canada, August 29-31, 2007.

[8] R. Sudarsan and C. Ribbens. ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. In *Proceedings of the 2007 International Conference on Parallel Processing (ICPP)*, page 44, XiAn, China, September 10-14, 2007.

[9] J. B. Weissman. Prophet: Automated scheduling of SPMD programs in workstation networks. *Concurrency: Practice and Experience*, 11(6):301–321, 1999.

[10] J. B. Weissman, L. Rao, and D. England. Integrated Scheduling: The Best of Both Worlds. *Journal of Parallel and Distributed Computing*, 63(6):649–668, 2003.