Parallel Computing 36 (2010) 48–64

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

# Design and performance of a scheduling framework for resizable parallel applications

Rajesh Sudarsan [*,1], Calvin J. Ribbens

*Department of Computer Science, Virginia Tech, Blacksburg, VA 24061-0106, United States*

ARTICLE INFO

*Article history:*
Received 30 August 2008
Received in revised form 13 August 2009
Accepted 22 December 2009
Available online 4 January 2010

*Keywords:*
Scheduling malleable applications
Run-time systems
Malleability in distributed-memory systems
Dynamic scheduling
High performance computing
Dynamic resizing
Data redistribution
Application resizing framework

ABSTRACT

This paper describes the design and initial implementation of a software framework for exploiting resizability in distributed-memory parallel applications. By "resizable" we mean the ability at run-time to expand or contract the number of processes participating in a parallel application. The ReSHAPE framework described here includes a cluster scheduler, a library supporting data redistribution and process remapping, and an application programming interface (API) which allows applications to interact with the scheduler and resizing library with only minor code modifications. Parallel applications executed using the ReSHAPE framework can expand to take advantage of additional free processors or contract to accommodate a high priority application without being suspended. Experimental results show that the ReSHAPE framework can significantly improve individual job turnaround time and overall system throughput, even with very simple application scheduling policies. In addition, the framework serves as a convenient platform for research into much more sophisticated cluster scheduling policies and methods.

## 1. Introduction

Processor counts on parallel supercomputers continue to rise steadily. Thousands of processor cores are becoming almost common place in high-end machines. 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. High capability computing platforms are by definition expensive, so the cost of underutilization is high. Failing to use 5% of the processor cores on a 100,000 core cluster is a much more serious problem than on a 100 core cluster. Furthermore, high capability computing is characterized by long-running, high processor-count jobs. A job-mix dominated by a relatively small number of such jobs is more difficult to schedule effectively on a large cluster. In almost 4 years of experience operating System X, a terascale system at Virginia Tech, we have observed that conventional schedulers struggle to achieve over 90% utilization with typical job-mixes, consisting of a high percentage of jobs requiring a large number of processors. A fundamental problem is that conventional parallel schedulers are static; once a job is allocated a set of resources, it continues to use those same resources until it finishes execution. 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 run-time. This is the focus of our research—dynamically reconfiguring (*resizing*) parallel applications.

* Corresponding author.
*E-mail addresses:* sudarsar@vt.edu (R. Sudarsan), ribbens@vt.edu (C.J. Ribbens).
[1] Supported in part by the National Science Foundation (CNS-0325534).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2009.12.010

There are many ways in which dynamic resizing can improve the utilization of clusters as well as reduce the time-to-completion (queue waiting time plus execution time) of individual cluster jobs. From the perspective of the scheduler, dynamic resizing can yield higher machine utilization and job throughput. For example, with static scheduling it is common to see jobs stuck in the queue because they require just a few more processors than are currently available. With resizing, the scheduler may be able to launch a job earlier (i.e., *back-fill*), by squeezing the job onto the processors that are available, and then possibly adding more processors later. Alternatively, the scheduler can add unused processors to a job so that the job finishes earlier, thereby freeing up resources 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.

Dynamic resizing also has potential benefits from the perspective of an individual job. A scheduling mechanism that allows a job to start earlier or gain processors later can reduce the time-to-completion for that job. Applications that consist of multiple phases, some of which are more computationally intensive or scalable than others, can benefit from resizing to the most appropriate processor count for each phase. Another possible way to exploit resizability is in identifying a processor count *sweet spot* for a particular job. For any (fixed problem size) parallel application, there is a point beyond which adding more processors does not help. Dynamic resizing gives applications and schedulers the opportunity to probe for processor-count sweet spots for a particular application and problem size.

In order to explore the potential benefits and challenges of dynamic resizing, we have developed *ReSHAPE*, a framework for dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment. The ReSHAPE framework includes a programming model and API, a run-time library containing methods for data redistribution, and a parallel scheduling and resource management system. In order for ReSHAPE to be a usable and effective framework, there are several important criteria which these components should meet. The programming model needs to be simple enough so that existing code can be ported to the new system without an unreasonable re-coding burden. Run-time mechanisms must include support for releasing and acquiring processors and for efficiently redistributing application state to a new set of processors. The scheduler must exploit resizability to increase system throughput and reduce job turn-around time.

In [15], we described the initial design of ReSHAPE and illustrated its potential with some simple experiments. In this paper, we describe the evolved design more fully, along with the application programming interface (API) and example ReSHAPE-enabled code, and give new and more complete experimental results which demonstrate the performance improvements possible with our framework. We focus here on improving the performance of iterative applications. However, the applicability of ReSHAPE can be extended to non-iterative applications as well. The main contributions of our framework include:

(1) a scheduler which dynamically manages resources for parallel applications executing in a homogeneous cluster;
(2) an efficient run-time library for processor remapping and data redistribution;
(3) a simple programming model and easy-to-use API for modifying existing scientific applications to exploit resizability.

The rest of the paper is organized as follows. Section 2 presents related work in the area of dynamic processor allocation. Section 3 describes the design and implementation details of the ReSHAPE framework, including the API. Section 4 presents the results of several experiments, which illustrate and evaluate the performance gains possible with our framework. Section 5 concludes with a summary and describes plans for extending and exploiting ReSHAPE.

## 2. Related work

Dynamic scheduling of parallel applications has been an active area of research for several years. Much of the early work targets shared memory architectures although several recent efforts focus on grid environments. Feldmann et al. [4] propose an algorithm for dynamic scheduling on parallel machines under a PRAM programming model. McCann et al. [9] propose a dynamic processor allocation policy for shared memory multiprocessors and study space-sharing vs. time-sharing in this context. Corbalan et al. [3] present a scheduling policy for shared memory systems that allocates processors based on the performance of the application. Moreira and Naik [10] propose a technique for dynamic resource management on distributed systems using a checkpointing framework called Distributed Resource Management Systems (DRMS). The framework supports jobs that can change their active number of tasks during program execution, map the new set of tasks to execution units, and redistribute data among the new set of tasks. DRMS does not make reconfiguration decisions based on application performance however, and it uses file-based checkpointing for data redistribution. A more recent work by Kalè et al. [6] achieves reconfiguration of MPI-based message passing programs. However, the reconfiguration is achieved using Adaptive MPI (AMPI), which in turn relies on Charm++ [8] for the processor virtualization layer, and requires that the application be run with many more threads than processors. Weissman et al. [21] 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 [20]. All processors send data to the root node for data redistribution. The authors present simulated results based on supercomputer workload traces.

Cirne and Berman [2] use the term *moldable* to describe jobs that can adapt to different processor sizes. 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. Vadhiyar and Dongarra [18,19] describe a user-level checkpointing framework called Stop Restart Software (SRS) for developing malleable and migratable applications for distributed and Grid computing systems. The framework implements a rescheduler which monitors application progress and can migrate the application to a better resource. Data redistribution is done via user-level file-based checkpointing. El Maghraoui et al. [7] describe a framework to enhance the performance of MPI applications through process checkpointing, migration and load-balancing. The infrastructure uses a distributed middleware framework that supports resource-level and application-level profiling for load-balancing. The framework continuously evaluates application performance, discovers new resources, and migrates all or part of an application to better resources. Huedo et al. [5] also describe a framework, called Gridway, for adaptive execution of applications in Grids. Both of these frameworks target Grid environments and aim at improving the resources assigned to an application by replacement rather than increasing or decreasing the number of resources.

The ReSHAPE framework described in this paper has several aspects that differentiate it from the above work. ReSHAPE is designed for applications running on distributed-memory clusters. Like [2,21], applications must be malleable in order to take advantage of ReSHAPE but in our case the user is not required to specify the legal partition sizes ahead of time. Instead, ReSHAPE can dynamically calculate partition sizes based on the run-time performance of the application. ReSHAPE has an additional advantage of being able to change dynamically an application's partition size, whereas the framework described by Cirne and Berman [2] needs to decide on the partition size before starting an application's execution. Our framework uses neither file-based checkpointing nor a single node for redistribution. Instead, we use an efficient data redistribution algorithm which remaps data on-the-fly using message-passing over the high-performance cluster interconnect. Finally, we evaluate our system using experimental data from a real cluster, allowing us to investigate potential benefits both for individual job turn-around time and overall system utilization and throughput.

## 3. System organization

The architecture of the ReSHAPE framework, shown in Fig. 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 basic design of the application scheduler was originally part of the DQ/GEMS project [17]. The initial extension of DQ to support resizing was done by Chinnusamy and Swaminathan [1,16]. 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. The library is implemented using standard MPI-2 [11] functions and can be used with any MPI-2 implementation. The resizing library currently includes redistribution algorithms for generic one-dimensional block data distribution and one- and two-dimensional block-cyclic data distribution (see Section 3.2.2), but it can be extended to support other data structures and other redistribution algorithms.

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. Hence, in the experiments described in this paper, we use applications with a single outer iteration, and with one or more large computations dominating each iteration. Our API gives programmers a simple way to indicate *resize points* in the application, typically at the end of each iteration of the outer loop. At resize points, the application contacts the scheduler and provides performance data to the scheduler. The metric used to measure performance is the time taken to compute each iteration.

### 3.1. Application scheduling and monitoring module

The application 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, along with a particular set of scheduling policies that are implemented in our current system. It is important to note that the ReSHAPE framework can easily be extended to support more sophisticated policies.

#### 3.1.1. 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 (i.e., the monitor receives an error signal) or finishes its execution successfully, the application monitor sends a job error signal or a job end sig-

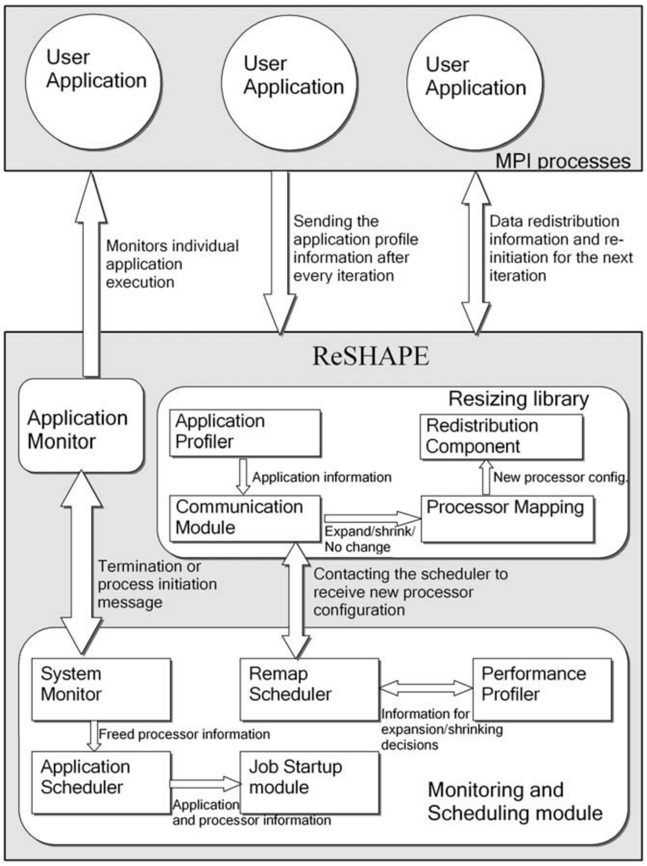

User Application
User Application
User Application
MPI processes
Monitors individual application execution
Sending the application profile information after every iteration
Data redistribution information and re-initiation for the next iteration
ReSHAPE
Application Monitor
Resizing library
Application Profiler
Redistribution Component
Application information
New processor config.
Communication Module
Expand/shrink/ No change
Processor Mapping
Termination or process initiation message
Contacting the scheduler to receive new processor configuration
System Monitor
Remap Scheduler
Performance Profiler
Information for expansion/shrinking decisions
Freed processor information
Application Scheduler
Job Startup module
Application and processor information
Monitoring and Scheduling module

**Fig. 1.** Architecture of ReSHAPE.

nal 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.

### 3.1.2. 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. Our current implementation supports two basic resource allocation policies, First Come First Served (FCFS) and simple back-fill. A job is backfilled if sufficient processors are available and the wallclock time of the job (supplied by the user) is smaller than the expected start time of the first job in the queue.

### 3.1.3. 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.

### 3.1.4. 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. The Profiler also maintains a list of possible shrink points of various applications and the anticipated impact on the application's performance. Each entry in the shrink points list consists of job id of an application, number of processors each application can relinquish and the expected performance degradation which would result from this change. Note that applications can only contract to processor configurations on which they have previously run.

### 3.1.5. 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. As an example of a simple but effective scheduling policy, our current implementation of ReSHAPE increases the number of processors given to an application if

(1) there are idle processors in the system, and
(2) there are no jobs waiting to be scheduled on the idle processors, and
(3) either the job has not yet been expanded, or the system predicts—based on performance results held by the Performance Profiler—that additional processors will help performance.

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 case an application prefers "nearly-square" topologies, additional processors are added to the smallest row or column of the existing topology.

The current Remap Scheduler decides to contract the number of processors for an application if it has previously run on a smaller processor set and

(1) at the last resize point, the application expanded its processor set to a size that did not provide any performance benefit, or
(2) there are applications in the queue waiting to be scheduled.

In our current implementation, the scheduler implements a policy that favors queued applications, i.e., it attempts to minimize the queue wait time. When an application contacts the scheduler at its resize point, the Remap Scheduler checks whether it can schedule the first queued job by contracting one or more running applications. If it can, then the scheduler contracts applications in ascending order of performance impact: applications expected to suffer the least impact from contraction, based on previous performance data, lose processors first. The Remap Scheduler checks whether the current job is one of the candidates that needs to be contracted. If it is, then the application is contracted either to its starting processor size or to one of its previous processor allocations depending upon the number of processors required to schedule the queued job. If more processors are required, then the scheduler will harvest additional processors when other applications check in at their resize point. If the scheduler chooses not to contract the current application, then it predicts whether the application will benefit from additional processors. If so, and if there are processors available with no queued applications, then the scheduler expands the application to its next possible processor set size. If the application's performance was hurt by the last expansion, then it is contracted to its previous smaller processor allocation. If none of the above condition are met, then the scheduler leaves the application's processor allocation unchanged.

A simple performance model is used to predict an application's performance using past results. Based on the prediction, the scheduler decides whether an application should expand, contract or maintain its current processor size. The model uses performance results from an application's last three resize intervals to construct a quadratic interpolant of the performance of the application over that range of processor allocations. A resize interval is the change in the number of processors between two resize points. Currently we use the application's iteration time as a measure of performance. We use the slope of the interpolant at the current processor set size as a predictor of potential performance improvement. If the slope exceeds a pre-determined threshold (currently 0.2), then the model predicts that adding more processors will benefit the application's performance. Otherwise, it predicts that the application will have little or no benefit from adding additional processors. In the case of applications that have only two resize intervals, the model predicts performance using a straight line. The prediction model is not used when the application has only one or no resize intervals. We emphasize again that a more sophisticated prediction model and policies are possible. Indeed, a significant motivation for ReSHAPE in general, and the Performance Profiler in particular, is to serve as a platform for research into more sophisticated resizing strategies.

## 3.2. Resizing library and API

The ReSHAPE resizing library includes routines for changing the size of the processor set assigned to an application and for mapping processors and data from one processor set to another. An application needs to be re-compiled with the resize library to enable the scheduler to add or remove processors dynamically to or from the application. During resizing, rather than suspending the job, the application execution control is transferred to the resize library which maps the new set of processors to the application and redistributes the data (if any). Once mapping is completed, the resize library returns control back to the application and the application continues with its next iteration. The application programmer needs to indicate the globally distributed data structures and variables so that they can be redistributed to the new processor set after resizing. Fig. 2 shows the different stages of execution required for changing the size of the processor set for an application.

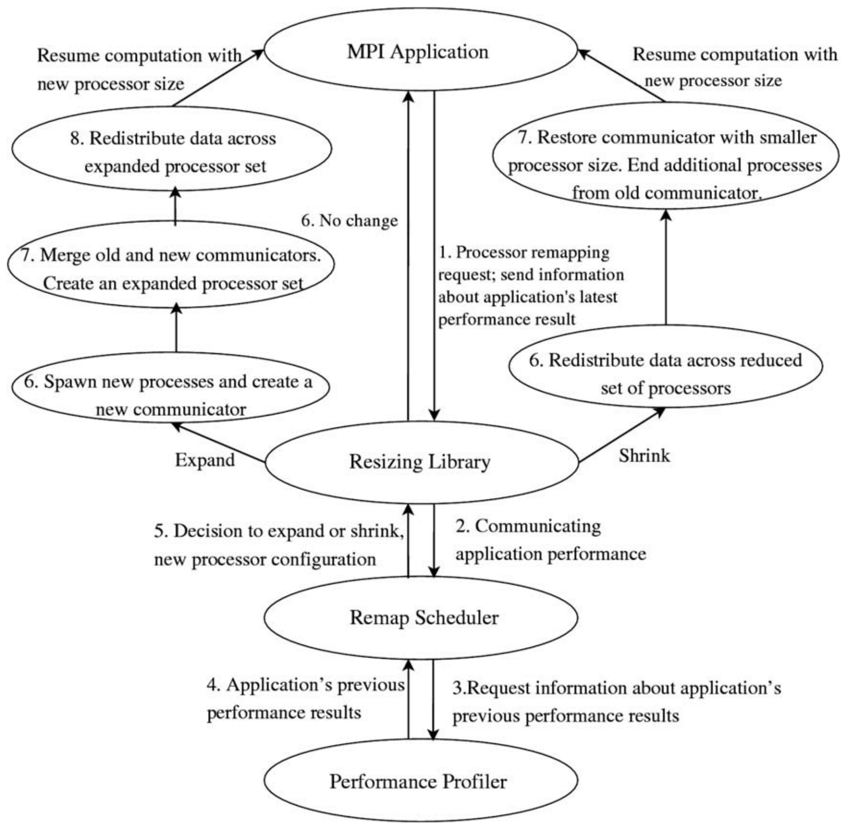

MPI Application
Resume computation with new processor size
Resume computation with new processor size
8. Redistribute data across expanded processor set
7. Restore communicator with smaller processor size. End additional processes from old communicator.
6. No change
7. Merge old and new communicators. Create an expanded processor set
1. Processor remapping request; send information about application's latest performance result
6. Redistribute data across reduced set of processors
6. Spawn new processes and create a new communicator
Expand
Resizing Library
Shrink
5. Decision to expand or shrink, new processor configuration
2. Communicating application performance
Remap Scheduler
4. Application's previous performance results
3.Request information about application's previous performance results
Performance Profiler

**Fig. 2.** State diagram for application expansion and contraction.

### *3.2.1. Processor remapping*

At resize points, if an application is allowed to expand to more processors, the response from the Remap Scheduler includes the size and the list of processors to which an application should expand. The resizing library spawns new processes on these processors using the MPI-2 function MPI_Comm_spawn_multiple called from within the library. The spawned child processes and the parent processes are merged together and a new communicator is created for the expanded processor set. The old MPI communicator is replaced with the newly created expanded intra-communicator. A call to the redistribution routine remaps the globally distributed data to the new processor set. If the scheduler tells an application to contract, then the application first redistributes its global data to the smaller processor subset, replaces the current communicator with the previously stored MPI communicator for the application and terminates any additional processes present in the old communicator. The resizing library notifies the Remap Scheduler about the number of nodes relinquished by the application. The current implementation of the remap scheduler supports processor configurations for three types of processor topologies: one-dimensional, two-dimensional (nearly-square topology), and powers-of-2 processor topology.

### *3.2.2. Data redistribution*

The data redistribution library in ReSHAPE uses an efficient algorithm for redistributing block and block-cyclic arrays between processor sets organized in a 1D (row or column format) or 2D processor topology. The algorithm used for redistributing 1D and 2D block-cyclic data uses a table-based technique to develop an efficient and contention-free communication schedule for redistributing data from $P$ to $Q$ processors, where $P \neq Q$. The initial (existing) data layout and the final data layout are represented in a tabular format. The number of rows in the initial and final data layout tables is equal to the least common multiple of the number of processor rows in the old and resized processor set. A least common multiple of the number of processor columns in the old and resized processor set determines the number of columns in the initial and final data layout tables. A third table—called the communication send table—represents the mapping between the initial and the final data layout, where columns correspond to the processors in the source processor set. Each row in this table represents a single communication step in which data blocks from the initial layout are transferred to their destination processors indicated by the individual entries in the communication send table. Another table—called the communication receive table—is derived from the communication send table, and stores the mapping information similar to the communication send table. In this table, the columns correspond to the destination processor set and each entry in the table indicates the source processor of the data stored in that location. The communication send and receive tables are rearranged such that the maximum possible number of processors are either sending or receiving at each communication step (depends whether $P > Q$ or $P < Q$)

without any node contention. Our current implementation of the checkerboard redistribution algorithm for 2D processor topology requires that the number of processors evenly divides the problem size. Sudarsan and Ribbens [14] describe the redistribution algorithm for a 2D processor topology in detail.

The one-dimensional block redistribution algorithm uses a linked list to represent the communication send and receive schedule. Each processor maintains an individual list for the communication send and receive schedule that stores the processor rank and the amount of data that needs to be sent or received from that processor. Similar to the block-cyclic redistribution algorithm, the block redistribution algorithm ensures that the maximum possible number of processors are either sending or receiving at each communication step. We plan to add additional routines to the library to support redistribution of three-dimensional dense matrices and one- and two-dimensional sparse matrices.

### *3.2.3. Application programming interface (API)*

A simple API allows user codes to access the ReSHAPE framework and library. These functions provide access to the main functionality of the resizing library by contacting the scheduler, remapping the processors after an expansion or a contraction, and redistributing the data. The API provides interfaces to support both C and Fortran language bindings. The API defines a global communicator called RESHAPE_COMM_WORLD required to communicate among the MPI processes executing within the ReSHAPE framework. This communicator is automatically updated after each resizing decision and reflects the most recent group of active MPI processes for an application. The core functionality of the framework is accessed through the following interfaces.

- *reshape_Init (struct ProcTopology):* Initializes the ReSHAPE system for an application. During initialization, information about processor topology (rectangular, power-of-2, or no topology), number of dimensions in the processor topology, number of processor in each dimension and the total number of processors is passed to the framework using a structure called *ProcessorTopology*. The structure holds a valid processor size for each dimension only if the number of dimensions is greater than 1.
- *reshape_Exit ():* Closes all socket connections and exits the ReSHAPE system.
- *reshape_ContactScheduler (time, niter, schedDecision):* The resizing library contacts the Remap scheduler with an average execution time for the last iteration and receives a response indicating whether to expand, contract or continue execution with the current processor set. The parameter *niter* holds the most recent value of the iteration counter used in the application.
- *reshape_Resize (schedDecision):* Actuates the processor remapping based on the scheduler's decision. After the job is mapped to the new set of processors, RESHAPE_COMM_WORLD is updated to reflect the new processor set.
- *reshape_Redistribute1D (data, arraylength, blocksize):* Redistributes data block-cyclically across processors arranged in a one-dimensional topology.
- *reshape_RedistributeUniformBlock (data, arraylength, blocksize):* Redistributes data using block distribution across processors arranged in one-dimensional topology.
- *reshape_Redistribute2D (data, Nrows, Ncols, blocksize):* Redistributes data according to a checkerboard data distribution across processors arranged in a 2D topology. *Nrows* and *Ncols* indicate the rows and columns of the 2D input matrix.
- *reshape_setCommunicator (MPI_Comm newcomm):* Updates RESHAPE_COMM-_WORLD with the communicator *newcomm*. This API must be called if an application creates its own processor topology and wants the global communicator to reflect it.
- *reshape_getSchedulerDecision (SchedDecision):* Returns the Remap scheduler's decision at the last resize point.
- *reshape_getNewProcessorDimensions (ndims, dimensions[ndims]):* Returns dimensions of the resized processor set (applicable only if ndims > 1).
- *reshape_getOldProcessorDimensions (ndims, dimensions[ndims]):* Returns the dimensions of the processor set before resizing (applicable only if ndims > 1).
- *reshape_getIterCount (iter):* Returns the current value of the main loop index; index is assumed to be zero for application's initial iteration.

Besides these API calls, other modifications needed to use ReSHAPE with an existing code involve code-specific local initialization of state that may be needed when a new process is spawned and joins an already running application, replacing all occurrences of MPI_COMM_WORLD with RESHAPE_COMM_WORLD and inserting a code-specific routine that checks the validity of processor size for the application, if not already present.

Fig. 3 illustrates a skeleton code for a typical parallel iterative application using MPI and a distributed 2D data structure. We modified the original MPI code to insert ReSHAPE API calls. Fig. 4 shows the modified code. The modification required adding at most 22 lines of additional code in the original application. The modifications are indicated in bold text.

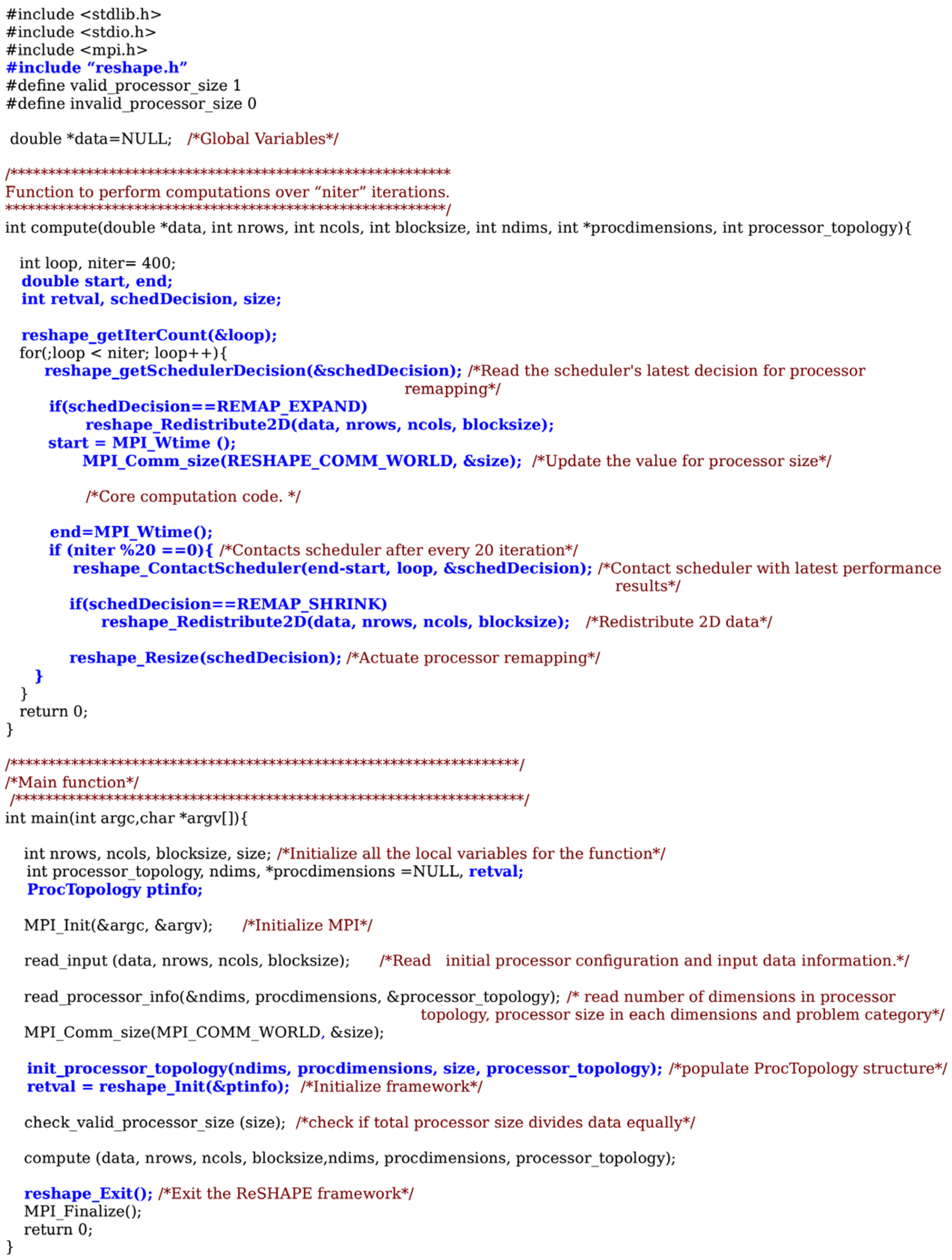
## 4. Experimental results

This section presents experimental results that demonstrate the potential of dynamic resizing for parallel applications. The experiments were conducted on 51 nodes of a large homogeneous cluster (Virginia Tech's System X). Each node has

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define valid_processor_size 1
#define invalid_processor_size 0

 double *data=NULL;   /*Global Variables*/

/************************************************************
Function to perform computations over "niter" iterations.
************************************************************/
int compute(double *data, int nrows, int ncols, int blocksize, int ndims, int *procdimensions,
                int processor_topology){

  int loop, niter= 400;

  for(loop=0;loop < niter;loop++){

      /*Core computation code. */
  }
  return 0;
}

/********************************************************************
Checks whether the new processor size is valid for the application.
********************************************************************/

int check_valid_processor_size(int procsize){

     int datasize;

     /*Implementation of this function is application-specific.
       1. Read input data size
       2. Check to see if the new processor size divides the data equally among all processors*/

      if (datasize % procsize == 0)
           return valid_processor_size;
      else
           return invalid_processor_size;
}

/*********************************************************************
Main function
*********************************************************************/
int main(int argc,char *argv[]){

    int nrows, ncols, blocksize, size;  /*Initialize all the local variables for the function*/
    int processor_topology, ndims, *procdimensions=NULL;
    MPI_Init (&argc, &argv);       /*Initialize MPI*/

     read_input (data, nrows, ncols, blocksize);    /*Read initial processor configuration and input data information.*/

     read_processor_info(&ndims, procdimensions, &processor_topology); /* read number of dimensions in processor
                                                          topology, processor size in each dimensions and problem category*/
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    check_valid_processor_size(size);  /*check if total processor size divides data equally*/

    compute (data, nrows, ncols, blocksize, ndims, procdimensions, processor_topology);

    MPI_Finalize();

    return 0;
}
```

**Fig. 3.** Original application code.

two 2.3 GHz PowerPC 970 processors and 4 GB of main memory. Message passing was done using OpenMPI [13] over an Infiniband interconnection network. We present results from two sets of experiments. The first set focuses on the benefits of resizing for individual parallel applications, while the second set looks at improvements in cluster utilization and throughput when several resizable applications are running concurrently. While these test problems and workloads do not corre-



```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>
#include "reshape.h"
#define valid_processor_size 1
#define invalid_processor_size 0

 double *data=NULL;   /*Global Variables*/

/************************************************************
Function to perform computations over "niter" iterations.
************************************************************/
int compute(double *data, int nrows, int ncols, int blocksize, int ndims, int *procdimensions, int processor_topology){

  int loop, niter= 400;
  double start, end;
  int retval, schedDecision, size;

  reshape_getIterCount(&loop);
  for(;loop < niter; loop++){
     reshape_getSchedulerDecision(&schedDecision); /*Read the scheduler's latest decision for processor
                                                    remapping*/
     if(schedDecision==REMAP_EXPAND)
         reshape_Redistribute2D(data, nrows, ncols, blocksize);
     start = MPI_Wtime ();
         MPI_Comm_size(RESHAPE_COMM_WORLD, &size);  /*Update the value for processor size*/

         /*Core computation code. */

     end=MPI_Wtime();
     if (niter %20 ==0){ /*Contacts scheduler after every 20 iteration*/
        reshape_ContactScheduler(end-start, loop, &schedDecision); /*Contact scheduler with latest performance
                                                                    results*/
       if(schedDecision==REMAP_SHRINK)
           reshape_Redistribute2D(data, nrows, ncols, blocksize);   /*Redistribute 2D data*/

       reshape_Resize(schedDecision); /*Actuate processor remapping*/
    }
  }
  return 0;
}

/*******************************************************************/
/*Main function*/
/*******************************************************************/
int main(int argc,char *argv[]){

   int nrows, ncols, blocksize, size; /*Initialize all the local variables for the function*/
   int processor_topology, ndims, *procdimensions =NULL, retval;
   ProcTopology ptinfo;

   MPI_Init(&argc, &argv);      /*Initialize MPI*/

   read_input (data, nrows, ncols, blocksize);      /*Read   initial processor configuration and input data information.*/

   read_processor_info(&ndims, procdimensions, &processor_topology); /* read number of dimensions in processor
                                                   topology, processor size in each dimensions and problem category*/
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   init_processor_topology(ndims, procdimensions, size, processor_topology); /*populate ProcTopology structure*/
   retval = reshape_Init(&ptinfo);  /*Initialize framework*/

   check_valid_processor_size (size);  /*check if total processor size divides data equally*/

   compute (data, nrows, ncols, blocksize,ndims, procdimensions, processor_topology);

   reshape_Exit(); /*Exit the ReSHAPE framework*/
   MPI_Finalize();
   return 0;
}
```

**Fig. 4.** ReSHAPE instrumented application code.

spond to real-world workloads in any rigorous sense, they are representative of the typical applications and data-distributions encountered on System X, and they serve to highlight the potential of the ReSHAPE framework.

## *4.1. Performance benefits for individual applications*

Although in practice applications rarely have an entire cluster to themselves, it is not uncommon for a relatively large number of processors to be available for at least part of the running-time of large applications. We can use those available processors to increase performance, or perhaps to discover that additional resources are not beneficial for a given application. The ReSHAPE Performance Profiler and Remap Scheduler uses this technique to probe for "sweet spots" in processor allocations and to monitor the trade-off between improvements in performance and the overhead of data redistribution. Four simple applications are used for the experiments in this section (see Table 1).

### *4.1.1. Adaptive sweet spot detection*

An appropriate definition of sweet spot for a parallel application depends on the context. A simple application-centric view is that the sweet spot is the point at which adding processors no longer helps reduce execution time. A more system-centric view would look at relative speedups when processors are added, and at the requirements and potential speedups of other applications currently running on the system. With the performance data gathering and resizing capabilities of ReSHAPE we can explore different approaches to sweet spot definition and detection. The current implementation of ReSHAPE uses the simple application-centric view of sweet spot: an application is given no more processors if the performance model (described in Section 3.1) predicts that additional processors will not improve performance.

To illustrate the potential of sweet spot detection, consider the data in Fig. 5, which shows the performance of LU for various problem sizes and processor configurations. All the jobs were run at the following processor set sizes (topologies): 2 ($1 \times 2$), 4 ($2 \times 2$), 8 ($2 \times 4$), 16 ($4 \times 4$), 32 ($4 \times 8$) and 64 ($8 \times 8$). We observe that the performance benefit of resizing for smaller matrix sizes is not high. In this case, ReSHAPE can easily detect that improvements are vanishingly small after the first few processor size increases. As expected, the performance benefits of adding more processors are greater for larger problem sizes. For example, in the case of a $8192 \times 8192$ matrix, the iteration time improves by 39.5% when the processor set increases from 16 to 32 processors. Our initial implementation of sweet spot detection in ReSHAPE simply adds processors as long as they are available and as long as there is improvement in iteration time. If an application grows to a configuration that yields no improvement, it is contracted back to its most recent configuration. This strategy yields reasonable sweet spots in all cases. For example, problem size 6000 has a sweet spot at 32 processors. However, the figure suggests a more sophisticated sweet spot detection algorithm is possible where performance over several configurations is used to detect relative improvements below some required threshold, or when resources are available probes configurations beyond the largest considered so far, to see if performance depends strongly on particular processor counts.

**Table 1**
Applications used to illustrate the potential of ReSHAPE for individual applications.

| Application | Description |
|---|---|
| LU | LU factorization (PDGETRF from ScaLAPACK) |
| MM | Matrix–matrix multiplication (PDGEMM from PBLAS) |
| MstWkr | Synthetic master-worker application. Each job requires 20,000 fixed-time work units |
| FFT | 2D fast fourier transform application used for image transformation |
| Jacobi | An iterative Jacobi solver (dense-matrix) application |

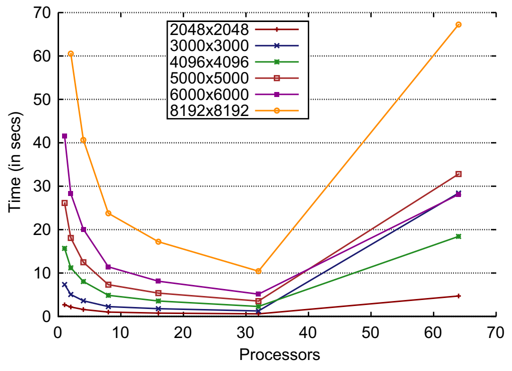


**Fig. 5.** Running time for LU with various matrix sizes.

### 4.1.2. Redistribution overhead

Every time an application adds or releases processors, the globally distributed data has to be redistributed to the new processor topology. Thus, the application incurs a redistribution overhead each time it expands or contracts. As an example, Table 2 shows the overhead for redistributing large dense matrices for different matrix sizes using the ReSHAPE resizing library. The figure lists six possible redistribution scenarios for expansion and contraction and their corresponding redistribution overhead for various matrix sizes. The overheads were computed using the 2D block-cyclic redistribution algorithm [14] with a block size of $256 \times 256$ elements. Each element in the matrix is stored as a double value. From the data in Table 2, we see that when expanding, the redistribution cost increases with matrix size, but for a particular matrix size the cost decreases as we increase the number of processors. This makes sense because for small processor sets, the amount of data per processor that must be transferred is large. Also the cost depends upon the size of the destination processor set, i.e., the larger the size of the destination processor set, the smaller the cost to redistribute. For example, the cost of redistributing data from 4 to 16 processors is lower than the cost for redistributing from 4 to 8 processors. The reasoning behind this is that for large destination processor size, the amount of data per processor that must be transferred is small. A similar reasoning can be applied for estimating the redistribution overhead when the processor size is contracted. The cost of redistributing data from 16 to 4 processors is much higher than the cost to redistribute from 16 to 8 processors. Also, for a particular matrix size, the redistribution cost when contracting is higher than that for expanding between the same sets of processors. This is because when contracting, the amount of data received by the destination processors is higher than the amount received when expanding. The large overhead incurred when redistributing a $32{,}768 \times 32{,}768$ matrix from 4 to 16 processors can be attributed to an increased number of page faults in the virtual memory system. The total data requirement for an application includes space required for storing problem data and for temporary buffers. Since the distributed $32{,}768 \times 32{,}768$ matrix does not fit in an individual node's actual memory, data must be swapped from disk, thereby increasing the redistribution cost.

By tracking data redistribution costs as applications are resized, ReSHAPE can better weigh the potential benefits of resizing a particular application. When considering an expansion, the important comparison is between data redistribution and potential improvements in performance. In some cases it may take more than one iteration at the new processor configuration to realize performance gains sufficient to outweigh the extra overhead incurred by data redistribution. But ReSHAPE can take this into account. As an example, Table 3 compares the improvement in iteration time with overhead cost involved in redistribution for the LU application with a matrix of size $16{,}000 \times 16{,}000$. This is a very good case for resizing, as the redistribution overhead is more than offset by performance improvements in only one iteration for each expansion carried out. For this application, the redistribution algorithm used a block size of $200 \times 200$ elements. The application is executed for 20 iterations and the timing results are recorded after every three iterations. The application reaches its resize point after every

**Table 2**
Redistribution overhead for expansion and contraction for different matrix sizes.

| Processor remapping | Redistribution for various matrix sizes (time in s) | | | |
|---|---|---|---|---|
| $P \to Q$ | 4096 | 8192 | 16,384 | 32,768 |
| *Expansion* | | | | |
| $4 \to 8$ | 0.252 | 0.896 | 3.451 | 250.380 |
| $4 \to 16$ | 0.264 | 0.790 | 3.011 | 158.761 |
| $8 \to 16$ | 0.100 | 0.290 | 1.088 | 4.362 |
| $16 \to 32$ | 0.130 | 0.297 | 1.047 | 4.092 |
| $16 \to 64$ | 0.162 | 0.290 | 0.901 | 3.530 |
| $32 \to 64$ | 0.057 | 0.121 | 0.351 | 1.540 |
| *Contraction* | | | | |
| $8 \to 4$ | 0.356 | 0.967 | 3.868 | 343.020 |
| $16 \to 4$ | 0.245 | 0.921 | 3.606 | 96.038 |
| $16 \to 8$ | 0.095 | 0.300 | 1.151 | 4.143 |
| $32 \to 16$ | 0.108 | 0.332 | 1.171 | 4.602 |
| $64 \to 16$ | 0.129 | 0.313 | 1.021 | 4.238 |
| $64 \to 32$ | 0.054 | 0.124 | 0.366 | 1.301 |

**Table 3**
Iteration and redistribution for LU on problem size 16,000.

| Processors | Iteration time ($T$) (in s) | $\Delta T$ (in s) | Redistribution cost (in s) |
|---|---|---|---|
| 4 ($2 \times 2$) | 161.54 | 0.00 | 0.00 |
| 16 ($4 \times 4$) | 63.49 | 98.04 | 2.83 |
| 20 ($4 \times 5$) | 54.99 | 8.50 | 1.38 |
| 25 ($5 \times 5$) | 46.59 | 8.40 | 1.08 |
| 64 ($8 \times 8$) | 33.73 | 12.86 | 1.52 |
| 100 ($10 \times 10$) | 28.32 | 5.41 | 0.89 |

three iterations and resizes based on the decision received from the remap scheduler. The application starts its execution with four processors and reaches its first resize point after three iterations. At each resize point, the application expands to the next processor size that equally divides the data among all processors and has a nearly-square topology. In the figure, $T$ indicates the time taken to execute one iteration of the application at the corresponding processor size. The application expands to 100 processors in the fifteenth iteration and continues to execute with the same size from iteration 18 to 20 due to non-availability of additional processors. In the figure, $\Delta T$ indicates the improvement in performance compared to the previous iteration. For this problem size the application does not reach its sweet spot, as it continues to gain performance with more processors (unlike the smaller problems shown in Fig. 5). The cost of redistribution at all the resize points is less than 20% of the improvement in iteration time.

A complex prediction strategy can estimate the redistribution cost [22] for an application for a particular processor size. The remap scheduler could use this estimate to make resizing decisions. However, with ReSHAPE we save a record of actual redistribution costs between various processor configurations, which allows for more informed decisions.

### 4.1.3. Comparison with file-based checkpointing

To get an idea of the relative overhead of redistribution using the ReSHAPE library compared to file-based checkpointing, we implemented a simple checkpointing library in which all data is saved and restored through a single node. Fig. 6 shows the computation (iteration) time and redistribution time with data redistribution by file-based checkpoint/restart and by the ReSHAPE redistribution algorithm. In both cases dynamic resizing is provided by ReSHAPE. The time for static scheduling is also shown in the figure, i.e., the time taken if the processor allocation is held constant at the initial value for the entire computation. Clearly, using more processors gives ReSHAPE an advantage over the statically scheduled case, which uses only four processors for LU, MM, MstWrk and Jacobi, and two processors for FFT. However, we include the statically scheduled case to make the point that only with ReSHAPE can we automatically probe for a sweet spot, and to give an indication of the relative cost of using file-based checkpointing. The results are from 10 iterations of each application, with ReSHAPE adding processors until a sweet spot is detected. The processor sets used for each problem are shown in Table 4. Notice that in order to detect the sweet spot, ReSHAPE goes beyond the optimal point, until speed-down is detected, before contracting back to the sweet spot, e.g., LU runs at $4 \times 5$ on iteration 6 before contracting back to a $4 \times 4$ processor topology.

Redistribution via checkpointing is obviously much more expensive than using our message-passing based redistribution algorithm. For example, with LU, checkpointing is 8.3 times more expensive than ReSHAPE redistribution. For matrix multiplication and FFT, the relative cost of checkpointing vs. ReSHAPE redistribution is 4.5 and 7.9, respectively. The master-worker application has no data to redistribute and hence shows no difference between checkpointing and ReSHAPE. For three of the test problems (LU, Jacobi and FFT) having to use file-based checkpointing means that much of the gains due to resizing are lost.

## 4.2. Performance benefits with multiple applications

The second set of experiments involves concurrent scheduling of a variety of jobs on the cluster using the ReSHAPE framework. Many interesting scenarios are possible; we only illustrate a few here. We assume that when multiple jobs are concurrently scheduled in a system and are competing for resources, not all jobs can adaptively discover and run at their sweet spot for the entirety of their execution. For example, a job might be forced to contract before reaching its sweet spot to accommodate new jobs. Since the applications that are closer to their sweet spot do not show high-performance gains from

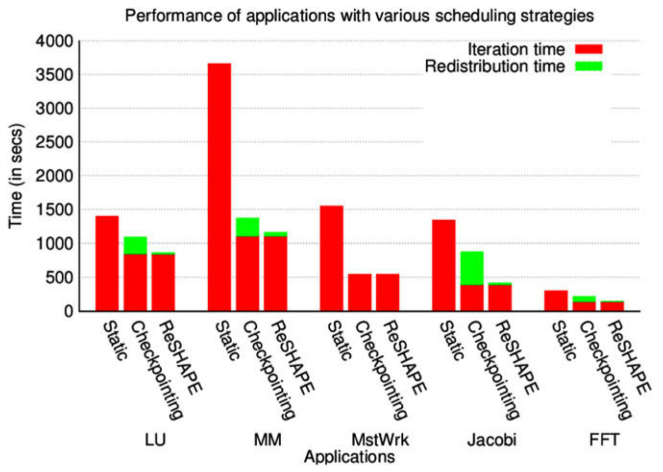


**Fig. 6.** Performance with static scheduling, dynamic scheduling with file-based checkpointing, and dynamic scheduling with ReSHAPE redistribution, for five test problems.

**Table 4**
Processor configurations for each of 10 iterations with ReSHAPE.

| Problem (size) | Processor configurations |
|---|---|
| LU (12,000) | 2 × 2, 2 × 3, 3 × 3, 3 × 4, 4 × 4, 4 × 5, 4 × 4, 4 × 4, 4 × 4, 4 × 4 |
| MM (14,000) | 2 × 2, 2 × 4, 4 × 4, 4 × 5, 5 × 5, 5 × 7, 7 × 7, 7 × 7, 7 × 7, 7 × 7 |
| MstWrk (20,000) | 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 |
| FFT (8192) | 2, 4, 8, 16, 32, 32, 32, 32, 32, 32 |
| Jacobi (8000) | 4, 8, 10, 16, 20, 32, 40, 50 |

**Table 5**
Workloads for job-mix experiment using NAS parallel benchmark applications.

| Workload | Application | Number of job(s) | Starting processors | Execution time for one iteration (s) |
|---|---|---|---|---|
| W1 | LU Class C | 1 | 32 | 426.65 |
| | CG Class C | 1 | 8 | 129.35 |
| | FT Class C | 1 | 4 | 267.00 |
| | IS Class C | 1 | 4 | 28.97 |
| | IS Class C | 1 | 32 | 5.90 |
| W2 | CG Class A | 2 | 4 | 2.00 |
| | FT Class A | 2 | 4 | 4.98 |
| | IS Class A | 2 | 4 | 1.58 |
| | IS Class B | 2 | 4 | 6.77 |
| W3 | LU Class C | 2 | 32 | 426.65 |
| | FT Class C | 2 | 64 | 20.65 |

adding processors, they become the most probable candidates for shrinking decisions. In addition, long running jobs can be contracted to a smaller size without a relatively high-performance penalty, thereby benefiting shorter jobs waiting in the queue. The ReSHAPE framework also benefits the long running jobs as they can utilize resources beyond their initial allocation as they become available. These scenarios are not possible in a conventional static scheduler where jobs in the queue wait until a fixed minimum number of processors become available.

We illustrate and evaluate ReSHAPE's performance using three different workloads. The first workload (W1) consists of a mix of five long running and short running jobs, whereas the second workload (W2) consists of 8 short running jobs. Each workload consists of a combination of four NAS parallel benchmarks [12]: LU, FT, CG and IS. For each case, a single job consisted of 20 iterations, where each iteration generated data and called the benchmark kernel once. No data was redistributed at resize points for these tests since the current ReSHAPE library does not include redistribution routines for sparse or 3D matrices. All problems were constrained to run at processor set sizes that were powers-of-2, i.e., 4, 8, 16, 32 and 64. Table 5 lists the problem sizes used for each workload. Workload W3 is used to illustrate the average queue wait time experienced by applications requesting high node-counts. A total of 102 processors were available for all the workloads in these experiments.

### *4.2.1. Workload 1*

Fig. 7 shows the number of processors allocated to each job for the lifetime of the five jobs scheduled by the ReSHAPE framework. Using this workload we illustrate how an application contracts to accommodate queued jobs. The data sizes considered for this experiment are limited by the availability of number of processors. On a larger cluster comprising hundreds or even thousands of processors, much larger applications can execute—larger in terms of both data size and processor allocations. In this workload, a 32 processor LU application was scheduled at $t = 0$ s. A CG application arrived at $t = 100$ s and was scheduled immediately with 8 processors. At $t = 220$ s, a FT application arrived followed by an IS job at $t = 500$ s. Both jobs were scheduled immediately with 4 processors. CG and IS gained additional processors at their resize points and expanded to 32 processors, thereby increasing the system utilization to 98% at $t = 752.5$ s. Since IS was a relatively short running job, it finished its execution at $t = 795$ s. The system utilization again reached 98% when CG expanded to 64 processor at $t = 869$ s. At $t = 1900$ a new IS job arrived with an initial request for 32 processors. Since LU had the next resize point at $t = 2130$ s, it contracted by 32 processors to accommodate the queued job. As a result, IS had to wait in the queue for 230.2 s before being scheduled. As there were no other running or queued jobs in the system after $t = 2957$ s, the LU application expanded to its maximum number of processors. Fig. 8 shows the total number of busy processors at any time for both static and ReSHAPE scheduling.

Table 6 shows the improved execution time for the applications in workload W1. The average processor utilization[2] with static scheduling with workload W1 is 43.1%, with the final job completed at time 10,033.4. Whereas the average processor uti-

[2] Utilization is defined as the percentage of total available cpu-seconds that are assigned to a running job.

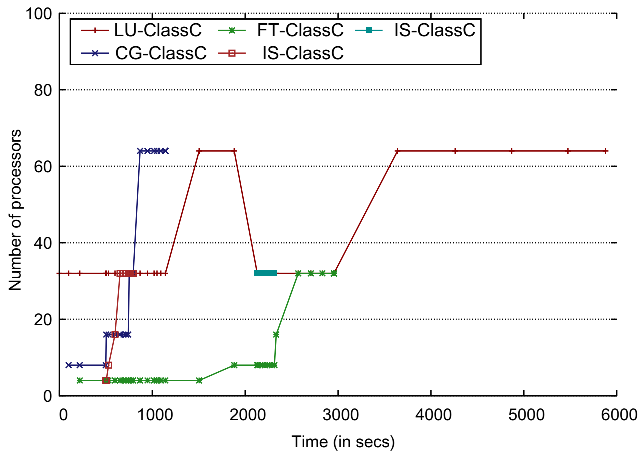


**Fig. 7.** Processor allocation history for workload W1.

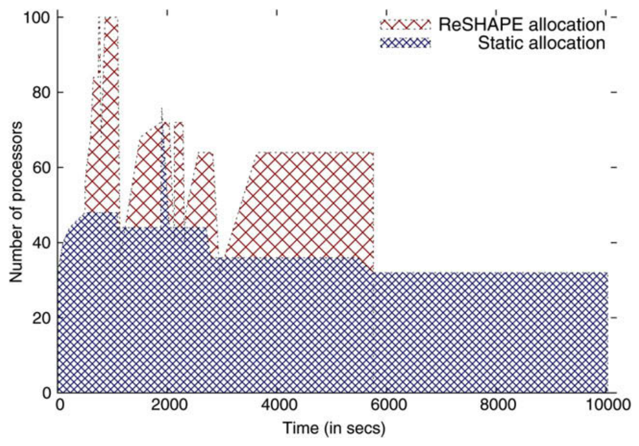


**Fig. 8.** Total processors used for workload W1.

**Table 6**
Job turn-around time.

| Job | Initial proc. alloc. | Static scheduling time (in s) | Dynamic scheduling time (in s) | Difference (in s) |
|---|---|---|---|---|
| LU Class C | 32 | 10,033.40 | 5879.07 | 4154.33 |
| CG Class C | 8 | 2698.00 | 1041.03 | 1656.97 |
| FT Class C | 4 | 5541.19 | 1736.12 | 2805.07 |
| IS Class C | 4 | 597.20 | 294.69 | 302.51 |
| IS Class C | 32 | 138.57 | 414.99 | −276.42 |

lization using ReSHAPE dynamic scheduling is 68.5%, with the final job completed at time 5879.07. Note that the IS application with initial request of 32 processors did not benefit from resizing, actually experiencing a later completion time than it would have under static scheduling. This is because in addition to waiting in the queue for 230.2 s waiting for another job to be contracted, it finished executing before additional processors became available.

### 4.2.2. Workload 2

Workload W2 (see Fig. 9) illustrates the potential benefits of resizing when a stream of short running jobs arriving at regular intervals are scheduled using the ReSHAPE framework. One would expect this case to be less promising for dynamic resizing since short duration jobs do not give ReSHAPE as many opportunity to exploit available processors. In this scenario, applications arrive at regular intervals with a delay of 10 s. A 4 processor CG application arrives at $t = 0$ s and expands to 64 processors at $t = 33.18$ s. FT, IS (Class A) and IS (Class B) jobs arrive at $t = 10$, 20 and 30 s, respectively, and expand to 32 processors during their execution lifetime. A second set of CG, FT and IS applications arrive at $t = 40$, 50, 60 and 70 s and are

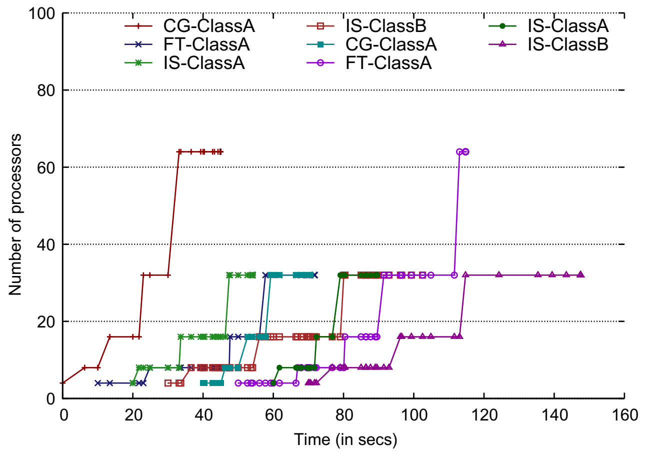


**Fig. 9.** Processor allocation history for workload W2.

scheduled immediately. All the jobs maintain their existing processor size for multiple resize points due to unavailability of additional processors. A caveat when scheduling short running jobs is that if the jobs arrive too quickly, they will be queued and scheduled statically. This is because these jobs generally have smaller processor requirements and thus will be scheduled immediately for execution. As a result, these jobs will use all the processors in the system leaving very few or no processors for resizing. As a result, all the applications will be forced to run at their starting processor size during the life of their execution. This will improve the overall system utilization but will also increase individual application turn-around time. If the jobs arrive with relatively longer interval delays, then they will be able to grow and use the entire system for their execution. Since the applications in workload W2 have short execution time, the interval between their resize points is small and hence they are resized frequently. As a result CG and IS (Class A) applications received little or no benefit due to resizing due to the overhead incurred during spawning new processes. FT and IS (Class B) benefit from resizing because they have relatively longer execution time compared to CG and IS (Class A) applications and hence they are able to offset the resizing overhead. Table 7 compares the execution time between static and dynamic scheduling for workload W2.

The performance of CG and IS (Class A) using ReSHAPE is worse than static scheduling due to frequent resizing. In such cases, a simple solution to detect and prevent frequent resizing is to compare the achieved speedup to the overall resizing cost (cost of spawning a new process plus redistribution overhead). Resizing must be allowed only when the speedup is greater than the resizing overhead. FT and IS (Class B) applications had moderately longer execution times and hence were resized less frequently resulting in an improvement in performance.

On the other hand, the steady stream of short running applications arriving at the scheduler helped maintain a high overall system utilization because these jobs were able to grow and use nearly all the processors in the system during their short execution lifetime. If the scheduling policy of the ReSHAPE framework is set to maximize the overall system utilization, scheduling short running jobs can be highly beneficial. Fig. 10 shows that for workload W2, dynamic scheduling using ReSHAPE ensures a high overall system utilization compared to static scheduling. The average processor utilization with static scheduling with workload W2 is 16%, whereas the average processor utilization using ReSHAPE dynamic scheduling is 69%.

### 4.2.3. *Average queue wait time*

Fig. 11 uses workload W3 to illustrate the average queue wait time experienced by high node-count applications when scheduled using ReSHAPE. Fig. 12 shows the processor allocations and queue wait time incurred by applications using static scheduling. With ReSHAPE, two LU applications, with an initial request for 32 processors, arrived at $t = 0$ and 100 s and were

**Table 7**
Job turn-around time.

| Job | Initial proc. alloc. | Static scheduling time (in s) | Dynamic scheduling time (in s) | Difference (in s) |
|---|---|---|---|---|
| CG (Class A) | 4 | 40.00 | 44.95 | –4.95 |
| FT (Class A) | 4 | 99.60 | 61.62 | 37.98 |
| IS (Class A) | 4 | 31.60 | 31.11 | –2.51 |
| IS (Class B) | 4 | 135.40 | 72.51 | 62.89 |
| CG (Class A) | 4 | 40.00 | 30.61 | 9.39 |
| FT (Class A) | 4 | 99.60 | 64.75 | 34.85 |
| IS (Class A) | 4 | 31.60 | 29.51 | 2.09 |
| IS (Class B) | 4 | 135.40 | 77.51 | 57.89 |

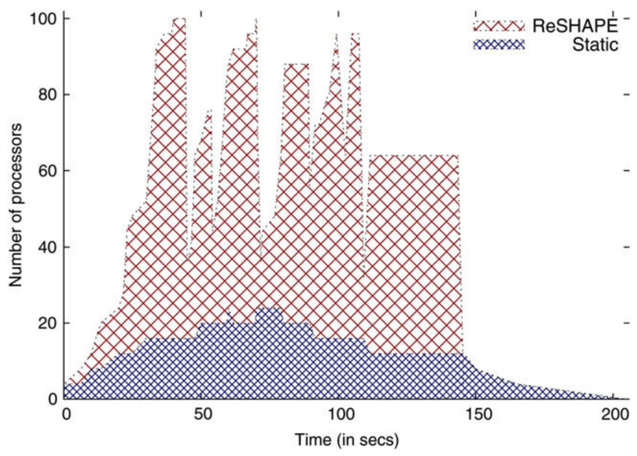


**Fig. 10.** Total processors used for workload W2.

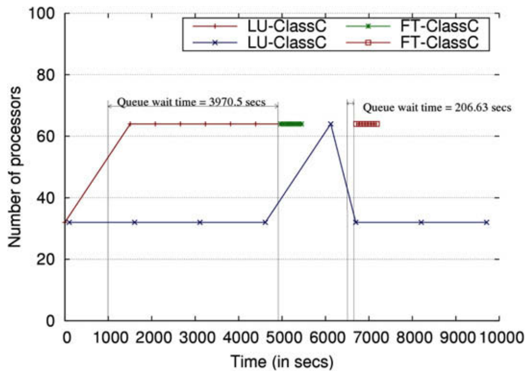


**Fig. 11.** Processor allocation history for workload W3 with ReSHAPE.

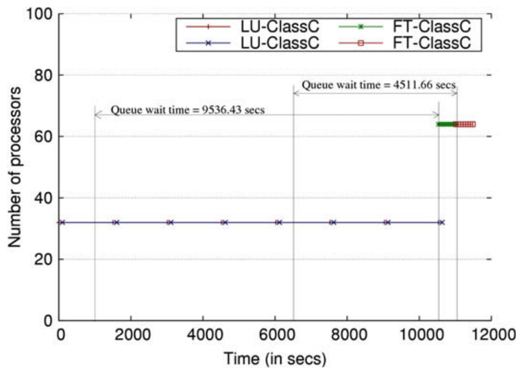


**Fig. 12.** Processor allocation history for workload W3 with static scheduling.

scheduled immediately. A FT application arrived at $t = 1000$ with an initial request of 64 processors and was queued due to insufficient processors. At $t = 2091$ s, the first LU application reached its resize point. Since the scheduler could not contract the application below its starting processor size, it allowed the application to expand to 64 processors. The LU application finished its execution at $t = 4978.3$ s. The queued FT application was scheduled at $t = 4978.3$ s after waiting in the queue for 3970.5 s. At $t = 5446.81$ s, FT finished its execution and released its 64 processors. The second LU application used these processors and expanded to 64 processors at its next resize point at $t = 6120.7$ s. A second FT application arrived at $t = 6500$ s with an initial request for 64 processors and was queued accordingly. The second LU application was contracted to 32 processors at it next resize point at $t = 6706.33$ s to accommodate the queued FT application. Since the resize point of LU was close to the arrival time of FT, the resulting wait time was just 206.33 s. In contrast, when jobs in W3 were statically scheduled, the first FT application was scheduled only after the LU application finished its execution at $t = 10{,}536.43$ s, thereby incurring a wait time of 9536.43 s. The second FT application was scheduled only after the first FT application finished its execution and released 64 processors at $t = 11{,}011.66$ s. The average wait time for the four applications in workload W3 scheduled using ReSHAPE was 1419.28, whereas the average wait increased to 3512.02 s when they were statically scheduled.

## 5. Conclusions and future work

In this paper, we have introduced a framework that enables parallel message-passing applications to be resized during execution. The ReSHAPE framework enables iterative applications to expand to more processors, thereby automatically probing for potential performance improvements. When multiple applications are scheduled using ReSHAPE, the system uses performance results from the executing applications to select jobs to contract in order to accommodate new jobs waiting in the queue while minimizing the negative impact on any one job. These applications can later be expanded to larger processor sets if the system has idle processors. The results show that applications executed using ReSHAPE demonstrate a significant improvement in job turn-around time and overall system throughput. The ReSHAPE run-time library includes efficient algorithms for remapping distributed arrays from one process configuration to another using message-passing. The system also records data redistribution times, so that the overhead of a given resizing can be compared with the potential benefits for long-running applications. The ReSHAPE API enables conventional SPMD (Single Program Multiple Data) programs to be ported easily to take advantage of dynamic resizing.

We are currently working to take advantage of the ReSHAPE framework to evaluate scheduling policies and strategies for processor reallocation, load-balancing, quality-of-service and advanced reservation services. Other current emphases include adding resizing capabilities to production scientific codes and adding support for a wider array of distributed data structures and other data redistribution algorithms. Finally, we plan to explore the combination of resizability and fault-tolerance (e.g., changing processor sets to avoid failed nodes as well as to improve performance), and to make ReSHAPE a more extensible framework so that support for heterogeneous clusters, grid infrastructure, shared memory architectures, and distributed-memory architectures can be implemented as individual plug-ins to the framework.

## References

[1] M. Chinnusamy, Data and processor mapping strategies for dynamically resizable parallel applications, Master's Thesis, Virginia Polytechnic Institute and State University, June 2004.
[2] W. Cirne, F. Berman, Using moldability to improve the performance of supercomputer jobs, Journal of Parallel and Distributed Computing 62 (10) (2002) 1571–1602.
[3] J. Corbalan, X. Martorell, J. Labarta, Performance-driven processor allocation, IEEE Transactions on Parallel and Distributed 16 (7) (2005) 599–611.
[4] A. Feldmann, J. Sgall, S.-H. Teng, Dynamic scheduling on parallel machines, Theoretical Computer Science 130 (1) (1994) 49–72.
[5] E. Huedo, R. Montero, I. Llorente, A framework for adaptive execution in grids, Software Practice and Experience 34 (7) (2004) 631–651.
[6] L.V. Kalè, S. Kumar, J. DeSouza, A malleable-job system for timeshared parallel machines, in: CCGRID '02: Proceedings of the Second IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE Computer Society, Washington, DC, USA, 2002, p. 230.
[7] Kaoutar El Maghraoui, Boleslaw Szymanski, Carlos Varela, An architecture for reconfigurable iterative MPI applications in dynamic environments, in: R. Wyrzykowski, J. Dongarra, N. Meyer, J. Wasniewski (Eds.), Proceedings of the Sixth International Conference on Parallel Processing and Applied Mathematics (PPAM'2005), vol. 3911 in LNCS, Poznan, Poland, 2005, pp. 258–271.
[8] Laxmikant V. Kalè, Sanjeev Krishnan, CHARM++: a portable concurrent object oriented system based on C++, in: OOPSLA '93: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press, 1993, pp. 91–108.
[9] C. McCann, R. Vaswani, J. Zarhojan, A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors, ACM Transactions on Computer Systems (TOCS) 11 (2) (1993) 146–178.
[10] J.E. Moriera, V.K. Naik, Dynamic resource management on distributed systems using reconfigurable applications, IBM Journal of Research and Development 41 (3) (1997) 303–330.
[11] Message Passing Interface Forum (MPI-2.1), 2008. <http://www.mpi-forum.org/>.
[12] NAS Parallel Benchmarks (NPB-MPI 2.4). <http://www.nas.nasa.gov/Software/NPB>.
[13] Open MPI v1.25, 2008. <http://www.open-mpi.org/>.
[14] R. Sudarsan, C.J. Ribbens, Efficient multidimensional data redistribution for resizable parallel computations, in: Proceedings of the International Symposium of Parallel and Distributed Processing and Applications (ISPA '07), Niagara falls, ON, Canada, 2007, pp. 182–194.
[15] R. Sudarsan, C.J. Ribbens, ReSHAPE: a framework for dynamic resizing and scheduling of homogeneous applications in a parallel environment, in: ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing (ICPP 2007), IEEE Computer Society, Xi'an, China, 2007, p. 44.
[16] G. Swaminathan, A scheduling framework for dynamically resizable parallel applications, Master's Thesis, Virginia Polytechnic Institute and State University, June 2004.
[17] S. Tadepalli, C.J. Ribbens, S. Varadarajan, GEMS: a job management system for fault tolerant grid computing, in: J. Meyer (Ed.), Proceedings of High Performance Computing Symposium 2004, Soc. for Modeling and Simulation Internat, San Diego, CA, 2004, pp. 59–66.
[18] S. Vadhiyar, J. Dongarra, SRS – a framework for developing malleable and migratable parallel applications for distributed systems, Parallel Processing Letters 13 (2) (2003) 291–312.
[19] S.S. Vadhiyar, J. Dongarra, A performance oriented migration framework for the grid, in: Proceedings of the Third IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), 2003, pp. 130–137.
[20] J.B. Weissman, Prophet: automated scheduling of SPMD programs in workstation networks, Concurrency: Practice and Experience 11 (6) (1999) 301–321.
[21] J.B. Weissman, L. Rao, D. England, Integrated scheduling: the best of both worlds, Journal of Parallel and Distributed Computing 63 (6) (2003) 649–668.
[22] R. Wolski, G. Shao, F. Berman, Predicting the cost of redistribution in scheduling, in: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing.