SRS - A Framework for Developing Malleable and Migratable Parallel Applications for Distributed Systems *

Sathish S. Vadhiyar and Jack J. Dongarra

Computer Science Department
University of Tennessee, Knoxville
{vss, dongarra}@cs.utk.edu

Abstract. The ability to produce malleable parallel applications that can be stopped and reconfigured during the execution can offer attractive benefits for both the system and the applications. The reconfiguration can be in terms of varying the parallelism for the applications, changing the data distributions during the executions or dynamically changing the software components involved in the application execution. In distributed and Grid computing systems, migration and reconfiguration of such malleable applications across distributed heterogeneous sites which do not share common file systems provides flexibility for scheduling and resource management in such distributed environments. The present reconfiguration systems do not support migration of parallel applications to distributed locations. In this paper, we discuss a framework for developing malleable and migratable MPI message-passing parallel applications for distributed systems. The framework includes a user-level checkpointing library called SRS and a runtime support system that manages the checkpointed data for distribution to distributed locations. Our experiment results indicate that the parallel applications, with instrumentation to SRS library, were able to achieve reconfigurability incurring about 15-35% overhead.

1 Introduction

Distributed systems and computational Grids [18] involve large system dynamics that it is highly desirable to reconfigure executing applications in response to the change in environments. Since parallel applications execute on large number of shared systems, the performance of the applications will be degraded if there is increase in external load on the resources caused by other applications. Also, it is difficult for users of parallel applications to determine the amount of parallelism for their applications and hence may want to determine the amount of parallelism by means of trial-and-error experiments. Due to the large number of machines involved in the distributed computing systems, the mean single

* This work is supported in part by the National Science Foundation contract GRANT #EIA-9975020, SC #R36505-29200099 and GRANT #EIA-9975015

processor failure rate and hence the failure rate of the set of machines where parallel applications are executing are fairly high [7]. Hence, for long running applications involving large number of machines, the probability of successful completions of the applications is low. Also, machines may be removed from executing environment for maintenance.

In the above situations, it will be helpful for the users or the scheduling system to stop the executing parallel application and continue it possibly with a new configuration in terms of the number of processors used for the execution. In cases of the failure of the application due to non-deterministic events, restarting the application on a possibly new configuration also provides a way of fault tolerance. Reconfigurable or malleable and migratable application provide added functionality and flexibility to the scheduling and resource management systems for distributed computing.

In order to achieve starting and stopping of the parallel applications, the state of the applications have to be checkpointed. Elonazhy [16] and Plank [29] have surveyed several checkpointing strategies for sequential and parallel applications. Checkpointing systems for sequential [30, 37] and parallel applications [15, 10, 4, 34, 20] have been built. Checkpointing systems are of different types depending on the transparency to the user and the portability of the checkpoints. Transparent and semi-transparent checkpointing systems [30, 12, 34] hide the details of checkpointing and restoration of saved states from the users, but are not portable. Non-transparent checkpointing systems [23, 21, 27, 20] involves the users to make some modifications to their programs but are highly portable across systems. Checkpointing can also be implemented at the kernel level or user-level.

In this paper, we describe a checkpointing infrastructure that helps in the development and execution of malleable and migratable parallel applications for distributed systems. The infrastructure consists of a user-level semi-transparent checkpointing library called SRS (**S**top **R**estart **S**oftware) and a Runtime Support System (RSS). Our SRS library is semi-transparent because the user of the parallel applications has to insert calls in his program to specify the data for checkpointing and to restore the application state in the event of a restart. But the actual storing of checkpoints and the redistribution of data in the event of a reconfiguration are handled internally by the library. Though there are few checkpointing systems that allow changing the parallelism of the parallel applications [21, 27], our system is unique in that it allows for the applications to be migrated to distributed locations with different file systems without requiring the users to manually migrate the checkpoint data to distributed locations. This is achieved by the use of a distributed storage infrastructure called IBP [28] that allows the applications to remotely access checkpoint data. Our experiment results indicate that the use of SRS library incurs low overheads for the the parallel applications. Our checkpointing infrastructure provides both pro-active preemption and restarts of the applications and tolerance in the event of failures. The infrastructure has been implemented and tested and the software is available for download at http://www.cs.utk.edu/~vss/srs.htm

The contributions of our checkpointing infrastructure are:

1. providing an easy-to-use checkpointing library that allows reconfiguration of parallel applications.
2. allowing checkpoint data to be ported across heterogeneous machines and
3. providing migration of the application across locations that do not share common file systems without requiring the user to migrate data.

In Section 2, we describe the SRS checkpointing library for reconfiguring applications. Section 3 explains the other parts of the checkpointing infrastructure, mainly the RSS service and the interactions between the applications and the checkpointing infrastructure. Section 4 lists the various steps needed for using the checkpointing infrastructure. Section 5 describes our experiments and results to demonstrate the overhead of our checkpointing system. Section 6 presents the current limitations of our checkpointing infrastructure. Section 7 looks at the relevant work in the field of checkpointing and migration of applications. In Section 8, we give concluding remarks and in Section 9, we present our future plans.

2 SRS Checkpointing Library

SRS (**S**top **R**estart **S**oftware is a user-level checkpointing library that helps to make iterative parallel MPI message passing applications reconfigurable. Iterative parallel applications cover a broad range of important applications including linear solvers, heat-wave equation solvers, partial differential equation (PDE) applications etc. The SRS library has been implemented in both C and Fortran and hence SRS functions can be called from both C and Fortran MPI programs. The SRS library consists of 6 main functions:

1. SRS_Init,
2. SRS_Restart_Value,
3. SRS_Read,
4. SRS_Register,
5. SRS_Check_Stop and
6. SRS_Finish.

The user calls SRS_Init after calling MPI_Init. SRS_Init is a collective operation and initializes the various data structures used internally by the library. SRS_Init also reads various parameters from a user-supplied configuration file. These parameters include the location of the Runtime Support System (RSS) and a flag indicating if the application needs periodic checkpointing. SRS_Init, after reading these parameters, contacts the RSS and sends the current number of processes that the application is using. It also receives the previous configuration of the application from the RSS if the application has been restarted from a previous checkpoint.

In order to stop and continue an executing application, apart from checkpointing the data used by the application, the execution context of the application also needs to be stored. For example, when the application is initially

started on the system, various data needs to be initialized, whereas when the application is restarted and continued, data needs to be read from a checkpoint and the initialization phase can be skipped. Most checkpointing systems [30] restore execution context by storing and retrieving execution stack. This solution compromises on the portability of the checkpointing system. Since the main goal of the SRS library is to provide heterogeneous support, the task of restoring the execution context is implemented by the user by calling SRS_Restart_Value. SRS_Restart_Value returns 0 if the application is starting its execution and 1 if the application is continued from its previous checkpoint. By using these values returned by SRS_Restart_Value, the user can implement conditional statements in his application to execute certain parts of the code when the application begins its execution and certain other parts of the code when the application is continued from its previous checkpoint.

SRS library uses Internet Backplane Protocol(IBP)[28] for storage of the checkpoint data. IBP depots are started on all the machines the user wants to use for the execution of his application. SRS_Register is used to mark the data that will be checkpointed by the SRS library during periodic checkpointing or when SRS_Check_Stop is called. Only the data that are passed in the SRS_Register call are checkpointed. The user specifies the parameters of the data including the size, data type and data distribution when calling SRS_Register. The data distributions supported by the SRS library include common data distributions like block, cyclic and block-cyclic distributions. For checkpointing data local to a process of the application or for data without distribution, a distribution value of 0 can be specified. SRS_Register stores the various parameters of the data in a local data structure. SRS_Register does not perform actual checkpointing of the data.

SRS_Read is the main function that achieves reconfiguration of the application. When the application is stopped and continued, the checkpointed data can be retrieved by invoking SRS_Read. The user specifies the name of the checkpointed data, the memory into which the checkpointed data is read and the new data distribution when calling SRS_Read. The data distribution specified can be conventional distributions or 0 for no distribution or SAME if the same data has to be propagated over all processes. The value SAME is useful for retrieving iterator values when all the processes need to start execution from the same iteration. The SRS_Read contacts the RSS and retrieves the previous data distribution and the location of the actual data. If no distribution is specified for SRS_Read, each process retrieves the entire portion of the data from the corresponding IBP depot used in the previous execution. If SAME is used for the data distribution, the first process reads the data from the IBP depot corresponding to the first process in the previous execution and broadcasts the data to the other processes. If data distribution is specified in SRS_Read, SRS_Read determines the data maps for the old and new distributions of the data corresponding to the previous and the current distributions. Based on the information contained in the data maps, each process retrieves its portion of data from the IBP depots containing the data portions. Thus reconfiguration of the application is achieved

by using different level of parallelism for the current execution and specifying a data distribution in SRS_Read that may be different from the distribution used in the previous execution.

SRS_Check_Stop is a collective operation and called at various phases of the program to check if the application has to be stopped. If SRS_Check_Stop returns 1, then an external component has requested for the application to stop, and the application can execute application-specific code to stop the executing application. SRS_Check_Stop contacts the RSS to retrieve a value that specifies if the application has to be stopped. If an external component has requested for the application to be stopped, SRS_Check_Stop stores the various data distributions and the actual data registered by SRS_Register to the IBP [28] depots. Each process of the parallel application stores its piece of data to the local IBP depot. By storing only the data specified by SRS_Register and requiring each process of the parallel application to the IBP depot on the corresponding machine, the overhead incurred for checkpointing is significantly low. SRS_Check_Stop sends the pointers for the checkpointed data to RSS and deletes all the local data structures maintained by the library.

SRS_Finish is called collectively by all the processes of the parallel application before MPI_Finish in the application. SRS_Finish deletes all the local data structures maintained by the library and contacts the RSS requesting the RSS to terminate execution.

Apart from the 6 main functions, SRS also provides SRS_DistributeFunc_Create and SRS_DistributeMap_Create to allow the user specify his own data distributions instead of using the data distributions provided by the SRS library.

Figure 1 shows a simple MPI based parallel program. The global data indicated by *global_A* is initialized in the first process and distributed across all the processes in a block distribution. The program then enters a loop where each element of the global data is incremented by a value of 10 by the process holding the element. Figure 2 shows the same code instrumented with calls to the SRS library. The application shown in Figure 2 is reconfigurable in that it can be stopped and continued on a different number of processors.

3 Runtime Support System (RSS)

RSS is a sequential application that can be executed on any machine with which the machines used for the execution of actual parallel application will be able to communicate. RSS exists for the entire duration of the application and spans across multiple migrations of the application. Before the actual parallel application is started, the RSS is launched by the user. The RSS prints out a port number on which it listens for requests. The user fills a configuration file called *srs.config* with the name of the machine where RSS is executing and the port number printed by RSS and makes the configuration file available to the first process of the parallel application. When the parallel application is started, the first process retrieves the location of RSS from the configuration file and registers

```
int main(int argc, char** argv){
int *global_A, int* local_A;
int global_size, local_size;
MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  local_size = global_size/size;

  if(rank == 0){

    for(i=0; i<global_size; i++){
      global_A[i] = i;
    }
  }

  MPI_Scatter (global_A, local_size, MPI_INT, local_A, local_size,
                MPI_INT, 0, comm );

  for(i=0; i<global_size; i++){

    proc_number = i/local_size;
    local_index = i\%local_size;

    if(rank == proc_number){
      local_A[local_index] += 10;
    }
  }

  MPI_Finalize();

  exit(0);

}
```

Figure 1. Original code

```
int main(int argc, char** argv){
int *global_A, int* local_A;
int global_size, local_size;
MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);
  SRS_Init();

  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  local_size = global_size/size;
  restart_value = SRS_Restart_Value();

  if(restart_value == 0){
    if(rank == 0){

      for(i=0; i<global_size; i++){
        global_A[i] = i;
      }
    }

    MPI_Scatter (global_A, local_size, MPI_INT, local_A, local_size,
                 MPI_INT, 0, comm );

    iter_start = 0;
  }
  else{
    SRS_Read("A", local_A, BLOCK, NULL);
    SRS_Read("iterator", &iter_start, SAME, NULL);
  }

  SRS_Register("A", local_A, GRADS_INT, local_size, BLOCK, NULL);
  SRS_Register("iterator", &i, GRADS_INT, 1, 0, NULL);

  for(i=iter_start; i<global_size; i++){
    stop_value = SRS_Check_Stop();
    if(stop_value == 1){
      MPI_Finalize();
      exit(0);
    }

    proc_number = i/local_size;
    local_index = i\%local_size;

    if(rank == proc_number){
      local_A[local_index] += 10;
    }
  }

  SRS_Finish();
  MPI_Finalize();

  exit(0);

}
```

Figure 2. Modified code with SRS calls

with the RSS during SRS_Init. The RSS maintains the application configuration of the present as well as the previous executions of the application.

The RSS also maintains an internal flag, called *stop_flag* that indicates if the application has to be stopped. Initially, the flag is cleared by the RSS. A utility called *stop_application* is provided and allows the user to stop the application. When the utility is executed with the location of RSS specified as input parameter, the utility contacts the RSS and makes the RSS set the *stop_flag*. When the application calls SRS_Check_Stop, the SRS library contacts the RSS and retrieves the *stop_flag*. The application either continues executing or stops its execution depending on the value of the flag.

When the SRS_Check_Stop checkpoints the data used in the application to IBP depots, it sends the location of the checkpoints and the data distributions to the RSS. When the application is later restarted, it contacts the RSS and retrieves the location of the checkpoints from the RSS. When the application finally calls SRS_Finish, the RSS is requested by the application to terminate itself. The RSS cleans the data stored in the IBP depots, deletes its internal data structures and terminates.

The interactions between the different components in the SRS checkpointing architecture is illustrated in Figure 3.

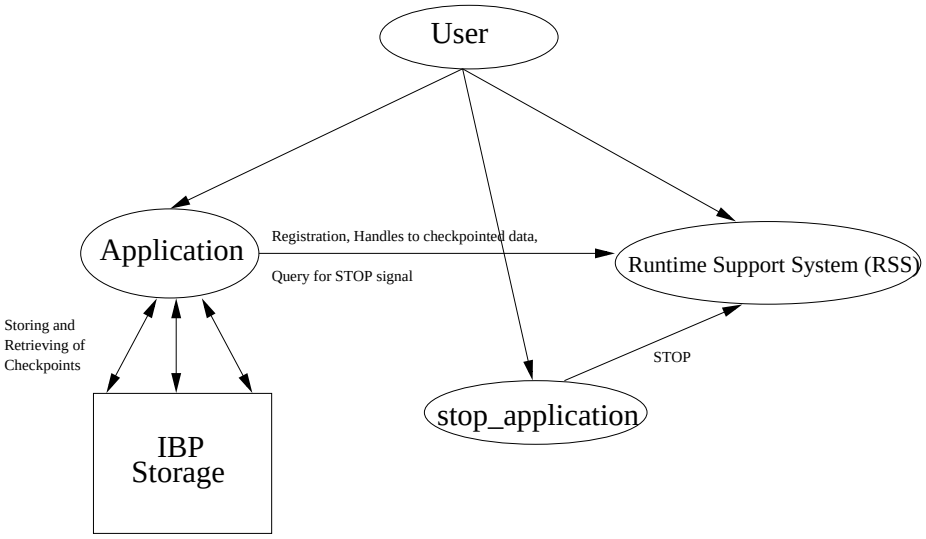

Figure 3. Interactions in SRS

4 Experiments and Results

Our experimental testbed consists of two clusters, one in University of Tennessee and another in University of Illinois, Urbana-Champaign. The Tennessee cluster consists of 8 933 MHz dual- processor Pentium III machines running Linux and connected to each other by 100 Mb switched Ethernet. The Illinois cluster consists of 16 450 MHz single-processor Pentium II machines running Linux and connected to each other by 1.28 Gbit/second full duplex myrinet. The two clusters are connected by means of the Internet.

ScaLAPACK QR factorization application was instrumented with calls to SRS library such that the modified code was malleable and migratable across heterogeneous clusters. The data that were checkpointed by the SRS library for the application included the matrix, A and the right-hand side vector, B. The experiments were conducted on non-dedicated machines.

4.1 SRS Overhead

In the first experiment, the overhead of SRS library was analyzed when checkpointing of data is not performed. Thus the application instrumented with SRS library simply connects to a RSS daemon and runs to completion. Figure 4 compares the execution of the factorization application on 8 UT machines when operated in three modes. The "Normal" mode is when the plain application without SRS calls is executed. In the second mode, the application instrumented with SRS library was executed connecting to a RSS daemon started at UT. In the third mode, the application instrumented with SRS library was executed connecting to a RSS daemon started at UIUC. The x-axis represents the matrix sizes used for the problem and the y-axis represents the total elapsed execution time of the application.

The maximum overhead of using SRS when RSS was started at UT was 15% of the overall execution time of the application. This is close to the 10% overhead that is desired for checkpointing systems [26]. The worst-case overhead of using SRS when RSS was started at UIUC was 29% of the overall execution time of the application. The increased overhead is due to the communication between SRS and RSS during initialization and at different phases of the application. Since RSS was located at UIUC, the communications involved slow Internet bandwidth between UT and UIUC. The large overhead can be justified by the benefits the SRS library provide in reconfiguring applications across heterogeneous sites.

Figure 5 shows the results of an experiment similar to the first experiment, but with the periodic checkpointing option turned on. In the periodic checkpointing mode, the SRS library checkpoints the application data to IBP depots every 10 minutes.

The worst-case SRS overheads in this experiment was high - 23% of the application time when RSS was located at UT and 36% of the application time when RSS was located at UIUC. The details of the periodic checkpointing used for Figure 5 is given in Table 1.

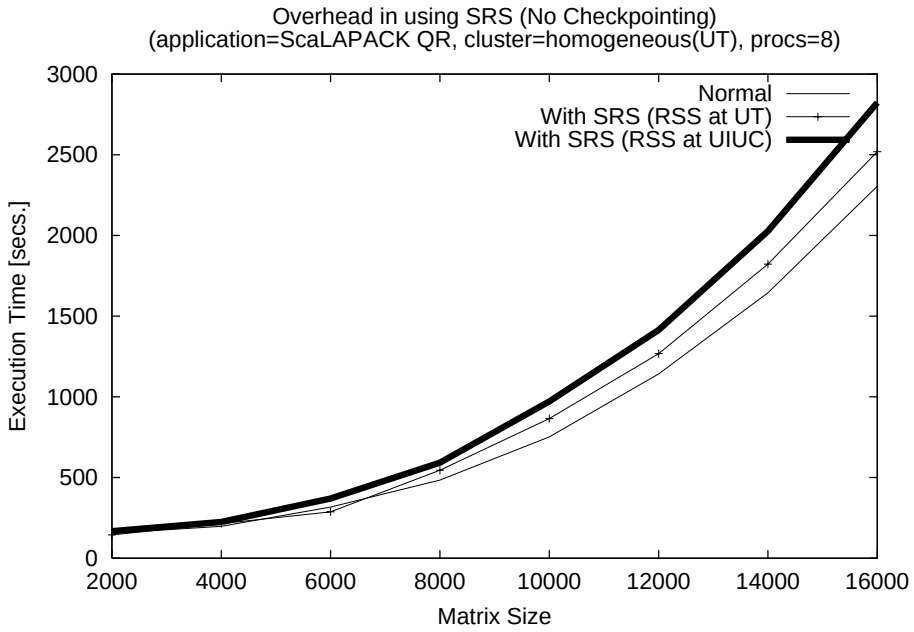

Figure 4. Overhead in SRS on a homogeneous cluster (No Checkpointing)

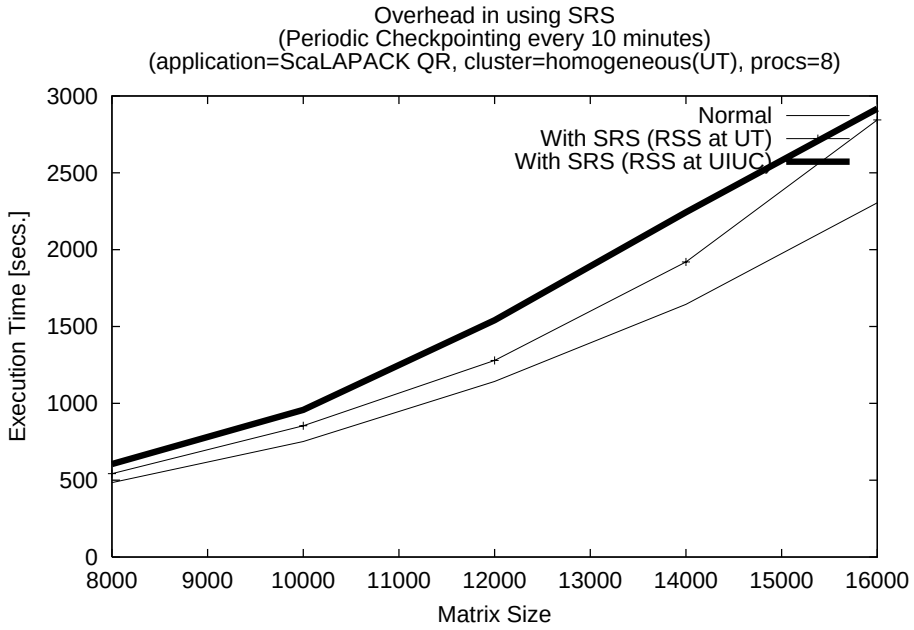

Figure 5. Overhead in SRS on a homogeneous cluster (Periodic Checkpointing)

Table 1. Details of Periodic Checkpointing used for Figure 5

Matrix Size	*Number of Checkpoints*	*Size per Checkpoint (MBytes)*	*Time for storing a checkpoint (Seconds)*
8000	1	64	6.51
10000	2	100	10.06
12000	3	144	13.68
14000	4	196	32.34
16000	5	256	93.25

From Table 1, it is clear that the high worst-case SRS overheads seen in Figure 5 are not due to the time taken for storing checkpoints. We suspect that the overheads are due to the transient loads on the non-dedicated machines.

In the third experiment in this subsection, the application was executed in a heterogeneous environment comprising 8 UIUC and 4 UT machines. The application was operated in 3 modes. "Normal" was when the plain application was executed. In the second mode, the application instrumented with SRS calls was executed without checkpointing. In the third mode, the application instrumented wit SRS calls was executed with periodic checkpointing of every 10 minutes. In the SRS mode, the RSS was started at UT. Figure 6 shows the results of the third experiment. The worst-case SRS overhead for the application was 15% and hardly noticeable in the figure. The details of the periodic checkpointing used in the third mode for the figure is given in Table 2.

Table 2. Details of Periodic Checkpointing used for Figure 6

Matrix Size	*Number of Checkpoints*	*Size per Checkpoint (MBytes)*	*Time for storing a checkpoint (Seconds)*
2000	1	2.5	4.59
4000	1	10.6	9.34
6000	2	24	11.22
8000	3	42.7	13.50
10000	5	66.7	18.51

4.2 SRS for Moldable Applications

In this subsection, results for experiments where the application is stopped and restarted on the same number of processors are shown. The application instrumented with SRS calls was initially executed at 8 UT machines. 3 minutes after the start of the application, the application, the application was stopped using the *stop_application* utility. The application was restarted on the same number of

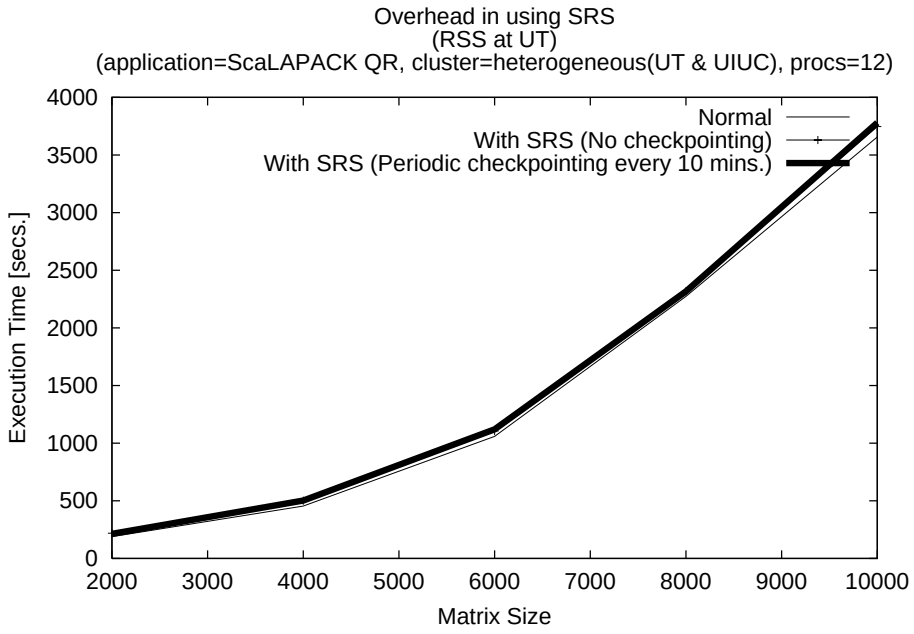

Figure 6. Overhead in SRS on a heterogeneous cluster

machines. In this scenario, the processes of the parallel application read the corresponding checkpoints from the IBP storage with performing any redistribution of data. The RSS daemon was started at UT.

Figure 7 shows the times for writing and reading checkpoints when the application was restarted on the same 8 UT machines on which it was originally executing. From the figure, we find that the times for writing and reading checkpoints are very low and in the range of 7-10 seconds. Thus the application can be removed from a system and restarted later on the same system for various reasons without much overhead. The time between when the stop signal was issued to the application and when the application actually stops depends on the moment when the application calls SRS_Check_Stop after the stop signal. Table 3 gives the checkpoint sizes used in Figure 7.

Figure 8 shows the results when the application was started at 8 UT machines, stopped and restarted at 8 UIUC machines. The increased times in reading checkpoints is due to the communication of checkpoints across the Internet from UT to UIUC machines.

4.3 SRS for Malleable Applications

In the experiments in this section, the application was started on 8 UT machines and restarted on a different number of machines spanning UT and UIUC. In this case, the restarted application, through the SRS library, determines the new data

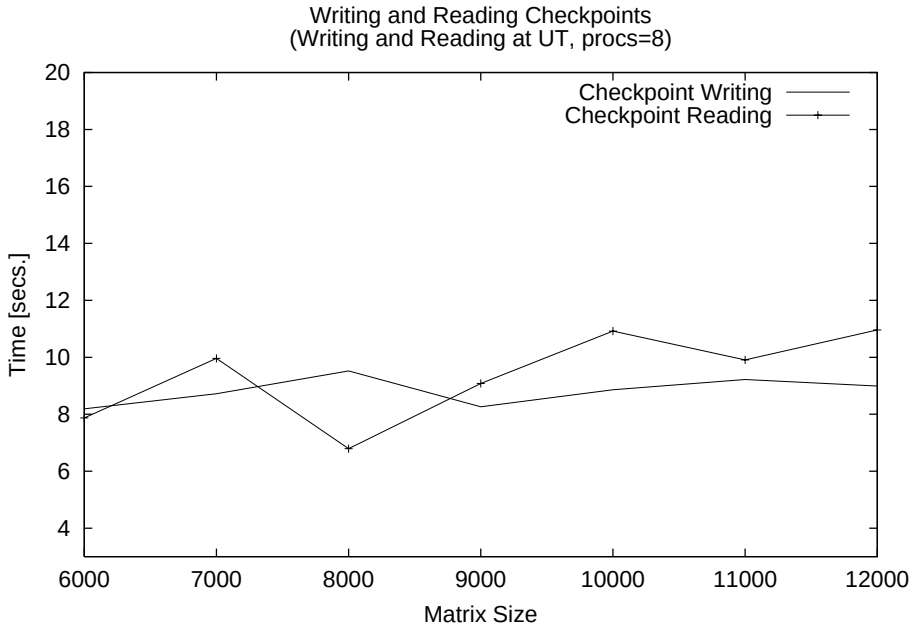

Figure 7. Times for Checkpoint Writing and Reading when the application was restarted on UT machines

Table 3. Details of Checkpointing used in Figure 7

Matrix Size	*Size per Checkpoint (MBytes)*
6000	36
7000	49
8000	64
9000	81
10000	100
11000	121
12000	144

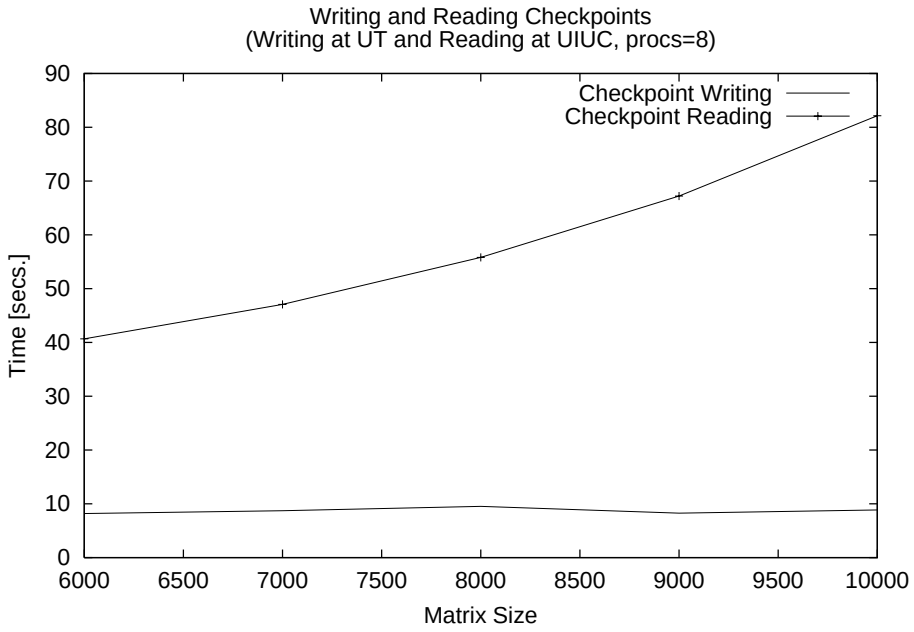

Figure 8. Times for Checkpoint Writing and Reading when the application was restarted on UIUC machines

maps for the processors and redistributed the stored checkpoint data among the processors. The RSS daemon was started on UT.

In Figure 9, results are shown when the ScaLAPACK QR application corresponding to matrix size 8000 was restarted on different number of processors (3 UIUC machines - 8 UIUC + 2 UT machines). The size of a single stored checkpoint was 64 MBytes. The time for data redistribution depends on the number and size of the data blocks that are communicated during redistribution and the network characteristics of the machines between which the data are transferred. When the application is restarted on a smaller number of processors, the size of the data blocks are large and hence the redistribution time is large. For larger number of processors, the redistribution time decreases due to the reduced size of data blocks communicated between the processors.

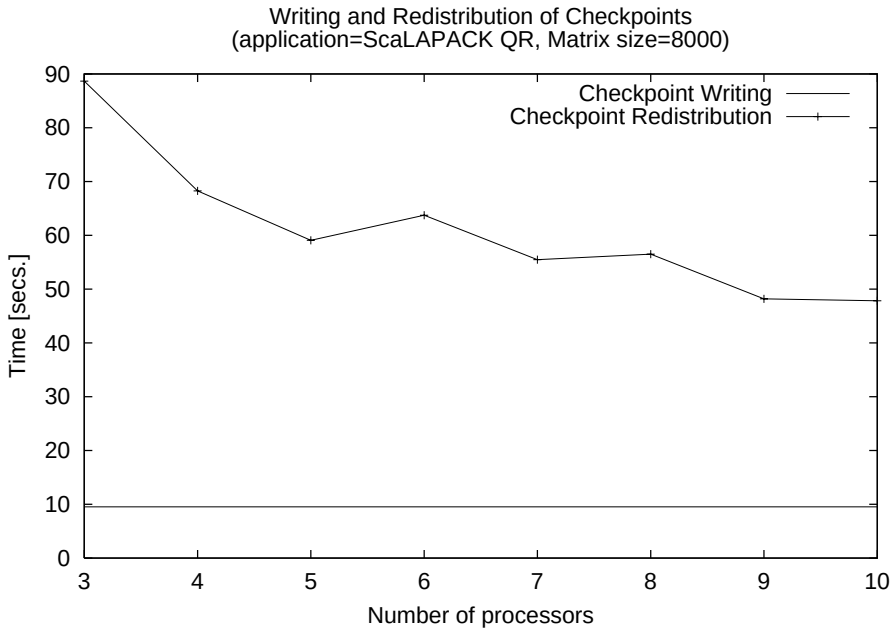

Figure 9. Times for Checkpoint Writing and Redistribution when the application was restarted on different number of processors

Figure 10 shows the dependence of the redistribution times on the problem size. For this experiment, the application was initially started on 8 UT machines and restarted on 8 UIUC and 2 UT machines.

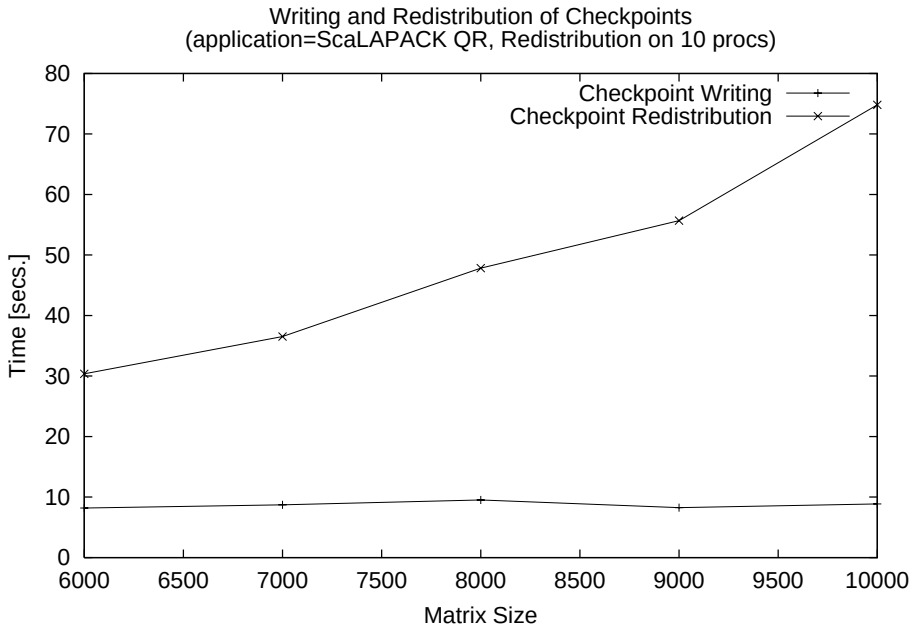

Figure 10. Times for Checkpoint Writing and Redistribution for different problem sizes

5 Steps for Developing and Executing Malleable Applications

Following is the summary of the actions needed for developing and executing malleable and migratable MPI message passing applications with the SRS library.

1. The user starts IBP depots on all machines where he many execute his application.
2. The user converts his parallel MPI application into a malleable application by inserting calls to SRS library. He then compiles and links with the SRS library.
3. The user then executes RSS on a machine with which the machines for application execution will be able to execute. The RSS will output a port number on which it listens for requests.
4. The user creates a configuration file specifying the machine and the port number of RSS.
5. The user stores the configuration file at the working directory of the first process of the parallel application.
6. The user starts his parallel application on a set of machines. The application, through the SRS library communicates with the RSS.
7. In the middle of the application execution, the user can stop the application by using the *stop_application* utility. The user specifies the location and the port number of the RSS to the *stop_application* utility.
8. The user can restart his application on possibly a different number of processors in the same way he initially started his application. After the application completes, the RSS terminates.

6 Limitations

Although the SRS framework is robust in supporting migration of malleable parallel applications across heterogeneous environments, it has certain limitations in terms of the checkpointing library and the kind of applications it can support.

1. Although the SRS library can be used in a large number of parallel applications, it is most suitable to iterative applications where SRS_Check_Stop can be inserted at the beginning or at the end of the loop. The SRS library is not suitable for applications like multi-component applications where different data can be initialized and used at different points in the program.
2. Currently, the execution context is restored by the user by the use of appropriate conditional statements in the program. This approach is cumbersome and difficult for the users when programs where multiple nested procedures are involved.
3. The SRS library supports only native data type like single and double precision floating point numbers, integers, characters etc. It does not support checkpointing of complex pointers, files and structures.

4. Although the main motivation of the SRS library is to help the user proactively stop an executing application and restart and continue it with a different configuration, SRS also allows fault tolerance by means of periodic checkpointing. However, the fault tolerance supported by SRS is limited in that it can tolerate only application failures due to non-deterministic events and not total processor failures. This is because the IBP depots on which the checkpoints are stored also fail when the machines on which the IBP depots are located fail.
5. The machine on which the RSS daemon is executing must be failure-free for the duration of the application.

7 Related Work

Checkpointing parallel applications have been widely studied in [16, 29, 25] and checkpointing systems for parallel applications have been developed [12, 10, 33, 38, 31, 15, 20, 34, 3, 23, 20, 4, 22, 21, 27]. Some of the systems were developed for homogeneous systems [12, 11, 33, 34] while some checkpointing systems allows applications to be checkpointed and restarted on heterogeneous systems [15, 20, 3–5, 23, 21, 27]. Calypso [5] and Plinda [23] require application writers to write their programs in terms of special constructs and cannot be used with proprietary software. Systems including Dynamic PVM [15] and CUMULVS [20] use PVM mechanisms for fault detection and process spawning and can only be used with PVM environments. Cocheck [34] and Starfish [3] provide fault tolerance with their own MPI implementations and hence are not suitable for distributed computing and Grid systems where the more secure MPICH-G [19] is used. CUMULVS [20], Dome [4, 7], the work by Hofmeister [22] and Deconick [13, 14, 9], DRMS [27] and DyRecT [2, 21] are closely related to our research in terms of the checkpointing API, the migrating infrastructure and reconfiguration capabilities.

The CUMULVS [20] API is very similar to our API in that it require the application writers to specify the data distributions of the data used in the applications and it provides support for some of the commonly used data distributions like block, cyclic etc. CUMULVS also supports stopping and restarting of applications. But the applications can be stopped and continued only on the same number of processors. Though CUMULVS supports MPI applications, it uses PVM as the base infrastructure and hence poses the restriction of executing applications on PVM.

Dome [4, 7] supports reconfiguration of executing application in terms of changing the parallelism for the application. But the data that can be redistributed for reconfiguration have to be declared as Dome objects. Hence it is difficult to use Dome with proprietary software like ScaLAPACK where native data is used for computations. Also Dome uses PVM as the underlying architecture and cannot be used for message passing applications.

The work by Hofmeister [22] supports reconfiguration in terms of dynamically replacing a software module in the application, moving a module to a different processor and adding or removing a module to and from the applications. But

the package by Hofmeister only works on homogeneous systems. The work by Deconinck [13, 14, 9] is similar to SRS in terms of the checkpointing API and the checkpointing infrastructure. Their checkpoint control layer is similar to our RSS in terms of managing the distributed data and the protocols for communication between the applications and the checkpoint control layer is similar to ours. By using architecture-independent checkpoints, the checkpoints used in their work are heterogeneous and portable. But the work by Deconick does not support reconfiguration of application in terms of varying the parallelism for the applications.

The DyRecT [2, 21] framework for reconfiguration allows dynamic reconfiguration of applications in terms of varying the parallelism by adding or removing the processors during the execution of parallel application. The user-level checkpointing library in DyRecT also supports the specification of data distribution. The checkpoints are system-independent and MPI applications can use the checkpointing library for dynamic reconfiguration across heterogeneous systems. But DyRecT uses LAM MPI [1] for implementing the checkpointing infrastructure to use the dynamic process spawning and fault detection mechanisms provided by LAM. Hence DyRecT is mainly suitable for workstation clusters and not distributed and Grid systems where the more secure MPICH-G is used [19]. Also, DyRecT requires the machines to share a common file system and hence applications cannot be migrated and reconfigured to distributed locations that do not share common file systems.

The DRMS [27] checkpointing infrastructure uses DRMS programming model to support checkpointing and restarting parallel applications on different number of processors. It uses powerful checkpointing mechanisms for storing and retrieving checkpoint data to and from permanent storage. It is the closest related work to SRS in that it supports a flexible checkpointing API for reconfiguring MPI message passing applications implemented on any MPI implementations to be reconfigured on heterogeneous systems. But DRMS also does not support migrating and restarting applications on environments that do not share common file systems with the environments where the applications initially executed.

A more recent work by Kalé et. al [24] achieves reconfiguration of MPI-based message passing programs. But reconfiguration is achieved by using a MPI implementation called AMPI [8] that is less suitable to Grid systems than MPICH-G.

8 Conclusions and Future Work

In this paper, a checkpointing infrastructure for developing and executing malleable and migratable parallel applications across heterogeneous sites was explained. The SRS API has limited number of functions for seamlessly enabling parallel applications malleable. The uniqueness of the SRS system is achieved by the use of IBP distributed storage infrastructure. Results were shown to evaluate the overhead incurred to the applications and the times for storing, reading and

redistributing checkpoints. The results show that SRS can enable reconfigurability of the parallel applications with limited overhead.

One of the main goals will be to use precompiler technologies to restore the execution context and to relieve the user from having to make major modifications in his program to provide malleability of his applications. The precompilation strategies will be similar to the approaches taken by Ferrari [17], Dome [7], Zandy [39] and Sun et. al. [36]. Other future investigations include support for checkpointing files, complex pointers and structures and to provide support for different kinds of applications.

Although the design of the checkpointing framework supports migration of heterogeneous environments, the current implementation stores the checkpoint data as raw bytes. This approach will lead to misinterpretation of the data by the application if, for example, the data is stored on a Solaris system and read by a Linux machine. This is due to the different byte orderings and floating point representations followed on different systems. We plan to use the External Data Representation (XDR) or Porch Universal Checkpointing Format (UCF) [35, 32] for representing the checkpoints.

We also plan to separate the storage nodes for checkpoints from the computational nodes for application execution by employing the eXNode [6] architecture. This will provide robust fault tolerant mechanism for withstanding the processor failures in SRS.

We also intend to collaborate with the CUMULVS project [20] to provide a generic visualization architecture that will be used to monitor the execution of malleable applications.

There are also plans to extend the RSS daemon to make it fault-tolerant by periodically checkpointing its state so that the RSS service can be migrated across sites. Presently, all the processes of the parallel application communicate with a single RSS deamon. This may pose a problem for the scalability of the checkpointing system, especially when large number of machines are involved. Our future plan is to implement a distributed RSS system to provide scalability.

References

1. LAM-MPI. *http://www.lam-mpi.org.*
2. A.Chowdhury. Dynamic Reconfiguration: Checkpointing Code Generation. In *In Proceedings of IEEE 5th International Symposium on Assessment of Software Tools and Technologies (SAST97)*, 1997.
3. A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In *In the 8th IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.
4. J.N.C. Arabe, A.B.B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan. Dome: Parallel Programming in a Heterogeneous Multi-User Environment. *Supercomputing*, 1995.
5. A. Baratloo, P. Dasgupta, and Z. M. Kedem. CALYPSO: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 122–129, August 1995.

6. M. Beck, T. Moore, and J. Plank. An End-to-End Approach to Globally Scalable Network Storage. In *ACM SIGCOMM 2002 Conference*, Pittsburgh, PA, USA, August 2002.
7. A. Beguelin, E. Seligman, and P. Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. Technical Report CMU-CS-96-157, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, August 1996.
8. M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.
9. B. Bieker, G. Deconinck, E. Maehle, and J. Vounckx. Reconfiguration and Checkpointing in Massively Parallel Systems. In *Proceedings of 1st European Dependable Computing Conference (EDCC-1)*, volume Lecture Notes in Computer Science Vol. 852, pages 353–370. Springer-Verlag, October 1994.
10. J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing, 1995.
11. J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole. MPVM: A Migration Transparent Version of PVM. Technical Report CSE-95-002, 1, 1995.
12. Y. Chen, K. Li, and J. S. Plank. CLIP: A Checkpointing Tool for Message-passing Parallel Programs. In *SC97: High Performance Networking and Computing*, San Jose, November 1997.
13. G. Deconinck and R. Lauwereins. User-Triggered Checkpointing: System-Independent and Scalable Application Recovery. In *Proceedings of 2nd IEEE Symposium on Computers and Communications (ISCC97)*, pages 418–423, Alexandria, Egypt, July 1997.
14. G. Deconinck, J. Vounckx, R. Lauwereins, and J.A. Peperstraete. User-triggered Checkpointing Library for Computation-intensive Applications. In *Proceedings of 7th IASTED-ISMM International Conference On Parallel and Distributed Computing and Systems (IASTED, Anaheim-Calgary-Zurich) (ISCC97)*, pages 321–324, Washington, DC, October 1995.
15. L. Dikken, F. van der Linden, J. J. J. Vesseur, and P. M. A. Sloot. DynamicPVM: Dynamic Load Balancing on Parallel Systems. In Wolfgang Gentzsch and Uwe Harms, editors, *Lecture notes in computer science 797, High Performance Computing and Networking*, volume Proceedings Volume II, Networking and Tools, pages 273–277, Munich, Germany, April 1994. Springer Verlag.
16. M. Elnozahy, L. Alvisi, Y.M. Wang, and D.B. Johnson. A Survey of Rollback-Recovery Protocols in Message Passing Systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.
17. A.J. Ferrari, S.J. Chapin, and A.S. Grimshaw. Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification. Technical Report Technical Report CS-97-05, Department of Computer Science, University of Virginia, March 1997.
18. I. Foster and C. Kesselman eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, ISBN 1-55860-475-8, 1999.
19. I. Foster and N. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. *In Proceedings of SuperComputing 98 (SC98)*, 1998.

20. G. A. Geist, J. A. Kohl, and P. M. Papadopoulos. CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications. *International Journal of High Performance Computing Applications*, 11(3):224–236, August 1997.
21. E. Godard, S. Setia, and E. White. DyRecT: Software Support for Adaptive Parallelism on NOWs. In *in IPDPS Workshop on Runtime Systems for Parallel Programming*, Cancun, Mexico, May 2000.
22. C. Hofmeister and J. M. Purtilo. Dynamic Reconfiguration in Distributed Systems : Adapting Software Modules for Replacement. In *Proceedings of the 13 th International Conference on Distributed Computing Systems*, Pittsburgh, USA, May 1993.
23. A. Jeong and D. Shasha. PLinda 2.0: A Transactional/Checkpointing Approach to Fault Tolerant Linda. In *Proceedings of the 13th Symposium on Reliable Distributed Systems*, pages 96–105. IEEE, 1994.
24. L.V. Kalé, S. Kumar, and J. DeSouza. A Malleable-Job System for Timeshared Parallel Machines. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2002)*, May 2002.
25. R. Koo and S. Toueg. Checkpointing and Rollback Recovery for Distributed Systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, 1987.
26. K. Li, J.F. Naughton, and J.S. Plank. Real-time Concurrent Checkpoint for Parallel Programs. In *In Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.
27. V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. In *SuperComputing (SC) '97*, San Jose, November 1997.
28. J. S. Plank, M. Beck, W. R. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the Network. *NetStore99: The Network Storage Symposium*, 1999.
29. James S. Plank. An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance. Technical Report UT-CS-97-372, 1997.
30. James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. Technical Report UT-CS-94-242, 1994.
31. P. Pruitt. An Asynchronous Checkpoint and Rollback Facility for Distributed Computations, 1998.
32. B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *27th International Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.
33. S. H. Russ, B. K. Flachs, J. Robinson, and B. Heckel. Hector: Automated Task Allocation for MPI. In *Proceedings of IPPS '96, The 10th International Parallel Processing Symposium*, pages 344–348, Honolulu, Hawaii, April 1996.
34. G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 526–531, Honolulu, Hawaii, 1996.
35. V. Strumpen and B. Ramkumar. Portable Checkpointing and Recovery in Heterogeneous Environments. Technical Report Technical Report 96-6-1, Department of Electrical and Computer Engineering, University of Iowa, June 1996.
36. X. H. Sun, V. K. Naik, and K. Chanchio. Portable hijacking. In *SIAM Parallel Processing Conference*, March 1999.
37. T. Tannenbaum and M. Litzkow. The condor distributed processing system. *Dr. Dobb's Journal*, pages 40–48, February 1995.

38. Z. You-Hui and P. Dan. A Task Migration Mechanism for Mpi Applications. In *In Proceedings of 3rd Workshop on Advanced Parallel Processing Technologies (APPT'99)*, pages 74–78, Changsha, China, October 1999.
39. V.C. Zandy, B.P. Miller, and M. Livny. Portable hijacking. In *The Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC'99)*, pages 177–184, August 1999.