Mobile MPI Programs in Computational Grids [*]

Rohit Fernandes, Keshav Pingali, Paul Stodghill

Department of Computer Science,
Cornell University,
Ithaca, NY 14853.
{rohitf,pingali,stodghil}@cs.cornell.edu

Abstract

Utility computing is becoming a popular way of exploiting the potential of computational grids. In utility computing, users are provided with computational power in a transparent manner similar to the way in which electrical utilities supply power to their customers. To take full advantage of utility computing, an application needs to be mobile; that is, it needs to be able to migrate between heterogeneous computing platforms while it is executing. Further, it needs to be able to adapt to the computing resources at each site, such as the number of available physical processors. At present, there are few high-performance computing applications of this sort, and re-engineering legacy codes to be mobile can take enormous effort.

In this paper, we describe the PC^3 system, which converts C/MPI codes into mobile programs almost transparently. Because it is based on portable application-level checkpointing, it enables the state of running applications to be saved so that the application can be restarted on different architectures, operating systems and MPI implementations. Moreover, the number of processors on these machines can be different. To our knowledge, this is the first system to provide all these features. Experimental results show that the overhead introduced by the system is usually small.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors— Run-time environments; D.4.5 [*Operating Systems*]: Reliability— Checkpoint/restart; C.4 [*Performance of Systems*]: Fault Tolerance

General Terms Reliability, Performance, Measurement

Keywords Grid Computing, Heterogeneity, Checkpoint/restart, Application-level Checkpointing, Portable Checkpointing, Over-decomposition, MPI

1. Introduction

Utility computing is becoming a popular way of exploiting the potential of computational grids. Instead of submitting jobs for execution on a particular computer as is done currently, users submit jobs to an agent that (i) finds the necessary resources somewhere in a federation of high-performance computers and high-bandwidth networks, (ii) schedules and monitors the execution of the job at that site, and (iii) migrates the job during its execution to different sites in the federation if it is advantageous to exploit changing resource availability.

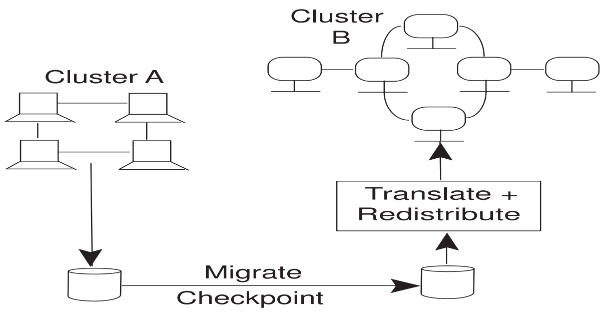


Figure 1. Migration of Mobile MPI Applications

To take full advantage of utility computing, an application needs to be mobile; that is, it needs to be able to migrate between platforms while it is executing, as shown in Figure 1. Although applications execute on a single site at any given time, a mobile application will be able to migrate to a different site if more resources become available there. Such applications are also more resilient to hardware faults since they can be restarted on a new site if the site they were executing on suffers a hardware failure.

At present, there are few high-performance computing applications of this sort. However, there are large numbers of existing codes written for traditional parallel machines, which might be good candidates for using emerging grid systems. How can these static applications be transformed into mobile applications? One approach would be to transform these applications by hand. Although this has been done for some codes such as the Cactus library [5], transforming codes manually is expensive. A more promising approach is to develop software tools to perform this transformation with minimal manual input. Criteria for success include the following.

- *Transparency*: A software tool that can transform programs more or less automatically to make them mobile would dramatically lower the cost of entry into the grid. We call this property transparency.
- *Adaptivity*: A mobile application must be able to adapt to the available resources. This implies that it should be able to re-

[*] This work was supported by NSF grants ACI-0103723, and ACI-0121401, ACI-0406345, and ACI-0541193.

PPoPP'06 March 29–31, 2006, New York, New York, USA.
Copyright © 2006 ACM 1-59593-189-9/06/0003...$5.00.

sume execution on a larger number of processors if some extra processors become available or on fewer processors when the situation demands (eg. a few processors crash).

- *Heterogeneity*: The problem of making applications mobile is obviously easier if all platforms are identical (same kind of processors, same number of processors, same operating system etc.). In reality, one rarely finds identical parallel platforms at different sites, so mobility between heterogenous platforms is desirable.
- *Performance*: Users who expect a high level of performance on one platform are unlikely to sacrifice performance for mobility. Although there may be cases where this tradeoff is appropriate, applications should retain their level of performance when they are made mobile.

In this paper, we describe a system called PC^3 (for Portable Cornell Checkpointing Compiler) that converts parallel C programs using the MPI communication library into mobile programs for computational grids. Our system is based on three key ideas.

- *Over-decomposition*: We decompose the original MPI application into a large number of virtual processes. This allows us to execute the application on different numbers of processors and obtain good load-balance by appropriately assigning virtual processes to physical processors.
- *Application-level checkpointing*: We use a pre-compiler to instrument a C program so that it can save and restore its own state without relying on operating system or hardware support. Unlike system-level checkpointing systems like Condor [11] that are tied to particular architectures and operating systems, application-level checkpointing provides an approach for making programs self-checkpointing and self-restarting. Further, our checkpointing substrate uses type information to save the state in a portable manner which allows restart on a different architecture.
- *Coordination layer*: To save the state of a parallel program while it is executing, it is necessary to co-ordinate the state-saving by the different processes. If the program is written in a bulk-synchronous manner, the state of the computation can be saved at global barriers. However, some programs, such as many mesh generators, do not have global barriers. To checkpoint such programs, our system uses a co-ordination protocol implemented by a thin software layer, which intercepts all calls made by the program to the MPI library [2, 3]. Saving the state of the MPI library in a portable way presents its own challenges.

To our knowledge, the PC^3 system is the first system to provide all of these features. Previous work on portable checkpointing [4, 14, 16] has focused on uniprocessor applications. Systems based on over-decomposition such as AMPI [6] provide either non-transparent support for portable checkpointing or use operating system checkpointing primitives that cannot be used on different architectures.

The rest of the paper is organized as follows. In Section 2, we describe our system architecture. In Section 3, we describe a selective re-implementation of MPI for supporting efficient over-decomposition. In Section 4, we describe our sequential checkpointer and the state translation mechanisms it uses to migrate state between different platforms. In Section 5, we discuss how we checkpoint the state of the MPI library. In Section 6, we present an experimental evaluation of our system. In Section 7, we survey related work. Finally, in Section 8, we discuss ongoing work.

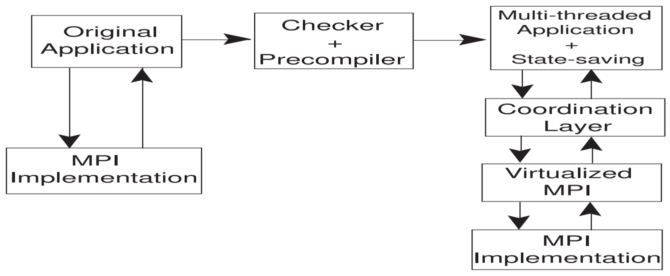


Figure 2. PC^3 System Architecture

2. System Architecture

Figure 2 shows the system architecture. At a conceptual level, the application is executed by some number of MPI processes that we call *logical* processes. The usual model of MPI execution is that each logical process is mapped to one physical processor.

Our system is based on the idea of *over-decomposition*, so the number of logical processes can be larger than the number of physical processors. Although there is nothing in principle that prevents multiple MPI processes from running as different OS processes on a single physical processor, the MPI implementations we have worked with were unable to support this model of computation efficiently. Therefore, in our system, the logical MPI processes mapped to a single physical processor are ”condensed” into a single multi-threaded MPI process in which each thread corresponds to one logical MPI process.

This design implies that the MPI communication layer must be able to accept messages from a number of end-points at a given processor, and deliver messages to a number of end-points at the receiving processor. Since this is different from the usual MPI model, we implement a thin layer that provides a virtual MPI interface to the application, as shown in Figure 2. The application code deals only with logical MPI processes. The layer knows the mapping of logical processes to physical processors, so it replaces logical process numbers in messages with the appropriate physical processor number, and piggy-backs the logical process number on the message. At the receiving end, the logical process number is used by the layer to determine which thread should receive an incoming message.

The coordination layer includes a checkpointing substrate that implements portable state-saving and restoring routines. The state of each logical MPI process is saved separately in a global checkpoint. The coordination layer also includes protocols for coordinating checkpointing of MPI processes so that the collection of checkpoints constitutes a global checkpoint from which the application can be restored [2].

To migrate a checkpoint to a different platform, the type of each object in the checkpoint is used to translate the representation of that object to the representation on the new platform. The state of each logical MPI process is translated in this way to its representation on the new platform, and the logical processes are redistributed among the physical processors. The processes mapped to a given processor are condensed into a single multi-threaded MPI process, and the program is restarted. As we discuss in Section 6, we have been able to take checkpoints on an Alpha cluster at the Pittsburgh Supercomputing Center and restart them on a Windows cluster at the Cornell Theory Center, demonstrating that these ideas work in practice.

Our two-pass source-to-source pre-compiler accepts C/MPI programs annotated with potential checkpoint locations and automatically generates the over-decomposed, fault-tolerant application. The first pass converts the program into a multi-threaded program so that the semantics remain the same as the original application. The second pass instruments the code for checkpointing by adding state-saving and recovery code. Since we use type-based checkpointing, we can only handle well-typed programs in which the type of every object in the checkpoint can be determined. Our pre-compiler has a type checker, which determines if the program is well-typed and which reports warnings if it detects unsafe features. It is purely syntactic at present, so it is rather conservative about warnings. The number of warnings can be reduced at the cost of performing more elaborate type checking.

3. Execution of Over-decomposed MPI Programs

We assume that the application has been decomposed into some number *n* of logical MPI processes. Because our system supports over-decomposition, the application can be run on fewer than *n* physical processors, but not on more than *n* physical processors. It is up to the programmer to determine the appropriate number of logical MPI processes. A parallel application of a given size does not scale beyond some number of processors, so that number is an appropriate value for *n*.

The goal of the virtualized MPI is to provide an efficient MPI interface so that the application can run in over-decomposed mode with little overhead. It also maintains statistics of the application's communication pattern so that on migration, it can compute a remapping of logical MPI processes to physical processors that reduces the amount of network communication.

3.1 Threaded Execution

The application runs as a set of multi-threaded processes, one per physical processor, in which each thread corresponds to a logical MPI process of the application. To fully utilize multi-processor nodes, we run as many processes on the node as there are processors. Each process also contains an additional master thread that performs some initialization at the beginning before spawning the threads corresponding to virtual processes.

We permit only a single thread per process to make progress at any time; all other threads are blocked. When a thread cannot make progress because of pending communication, it yields to another another. When all threads have completed, the master thread is scheduled, which terminates the application.

3.2 Program Transformation

To preserve the semantics of the original application, our pre-compiler performs a simple code transformation to ensure that globals are thread-local. For each global, we allocate an array on the heap, the size of which is the number of threads in the process. For each access to the corresponding global, we use an index to the array, which is computed from the rank of the thread. This index is passed as an additional parameter to all functions. For example, the code

```
int x;
double a[4];
int g(int i)
{ ...
}

int main(int argc, char **argv)
{ x=3;
    a[2]=7.0
    ...
    return g(x);
}
```

is transformed to

```
int *x;
double (*a)[4];

void pccc_allocate_globals()
{
    x=malloc(sizeof(int)* pccc_num_thr_local);
    a=malloc(sizeof(double[4])* pccc_num_thr_local);
}

int g(int pccc_tid, int i)
{
   ...
}

int pccc_main(int pccc_tid, int argc, char **argv)
{
    x[pccc_tid]=3;
    a[pccc_tid][2]=7.0
    ...
    return g(pccc_tid, x);
}
```

The PC^3 initialization phase calls **pccc_allocate_globals()**. **pccc_num_thr_local** refers to the number of virtual processes running in the current process. For each thread, the rank of the corresponding logical process in the original MPI application is used to compute **pccc_tid**, the index used by that thread for accessing its globals.

3.3 Design of Virtualized MPI Core

There are two communication paths - message transfer between threads within the same process (local send and receive) and transfers between threads in different processes (network send and receive). For the former, we rely on memory copies. For the latter, we use the native MPI to do the transfer. Each thread maintains two queues. The $pending\_recv\_queue$ contains receives that have been posted but not satisfied. The $inbound\_msgs$ queue contains incoming messages that are destined for the thread but for which receives have not been posted.

On a local send, we look at the $pending\_recv\_queue$ of the target thread to see if a matching receive is posted. If so, the data is directly copied to the destination and the current thread continues executing. Otherwise, if the send is a blocking send, the thread attempts to yield to another thread (an alternative would be to copy the message to a temporary buffer and continue, but we have found this strategy to inhibit the parallelism of the overall application for some common communication patterns). If the send is non-blocking, the thread continues but will attempt to yield when the corresponding *wait* call is issued. On a network send, the data is marshalled into a buffer and sent out on an MPI message to the target process. On a receive, we look at the $inbound\_msgs$ queue for the thread. If a matching message is found, we copy the data into the destination and continue executing. Otherwise, if it is a blocking call, we attempt to yield to another thread. For non-blocking calls, we continue executing the thread; this thread will attempt to yield to another thread at the corresponding *wait* call.

When a thread attempts to yield, it first probes the network interface to see if there are incoming messages; if so, it reads them and populates them in the appropriate queues. Some of these messages may free up blocked destination threads for execution. The current thread then attempts to find another thread to run using a round-robin policy. If it cannot find any thread that is ready to run, it tries to find a thread that is blocked on a send. If it finds such a thread, it copies the contents of the message to a temporary buffer and resumes execution of that thread. If no thread is ready to execute, it probes the network interface until a sequence of messages shows up that unblocks some thread. The first thread that is unblocked runs next.

Collective operations are implemented using point-to-point calls. They use a different communicator so as not be confused with point-to-point messages. We have not yet invested heavily in optimizing these routines. For the common case, where the collective operates on the entire group (MPI_GROUP_WORLD), we have a specialized implementation that maps on to the corresponding native collective. For example, for a barrier, each thread (other than thread 0) within a process sends an $Enter\_Barrier$ synchronization message (M_{eb}) to thread 0. When thread 0 receives all M_{eb} messages, it call **MPI_Barrier()**. After returning from the barrier, thread 0 sends a $Leave\_Barrier$ synchronization message (M_{lb}) to all the other local threads and resumes execution. On receipt of the M_{lb} message, the other threads cross the barrier. We have found that these special implementations perform as well as the native collectives.

The application performs its matching on the tag and rank of the logical MPI processes of the original application. When a network message is sent, these are piggybacked on the message. To preserve MPI ordering semantics, all network MPI messages are sent with a single tag, resulting in FIFO communication of the messages between processes. This tag is not to be confused with the piggybacked tag from the application. This scheme enforces a stronger ordering than that imposed by MPI which only requires that FIFO order be preserved for messages having the same communication channel (sender-receiver pair, tag and communicator). However, we believe this is not likely to affect performance. Having a single tag allows us to implement an efficient advance buffering scheme. We post a number of receives in advance on buffers of a pre-defined size. As messages come in, they get matched into the buffer in order of posting of the receives. Messages of size larger than the buffer are chunked into multiple successive messages. The size of the data is also piggybacked on the message so the receiver can know when a message has been completely received. In our current implementation, we use 512 buffers of size 32K. While we have not completely explored the space for these parameters, we have found that this combination handles the diverse communication requirements of our applications and performs well in practice. We have also found that this advance buffering approach is more stable than the **MPI_Probe()** operation.

Our implementation is efficient in many respects. It context-switches only when necessary. Because it allows only a single thread to make progress at a time, it can safely use a lock-free implementation. Further, it does not need to context switch for calling the native MPI, which would be necessary for systems that allow multiple threads to make progress or that use preemptive scheduling to context switch to a master thread to do all MPI communication.

We have implemented a useful subset of MPI that includes blocking and non-blocking point-to-point communications, collective communications and communicator/group construction operations. We do not see any conceptual hurdles for the remaining operations. An interesting set of operations that we have not completely evaluated is the set of polling operations which return immediately such as **MPI_Test()**. These tend to be used heavily in asynchronous-style communication. The applications we have considered do not perform any of these operations. If such an operation fails it will never be satisfied unless the network is probed or a context-switch is made. This could lead to livelock if such an operation is repeatedly executed in a loop. To ensure progress for such scenarios, we attempt to context switch any time such an operation fails.

4. Portable Checkpointing

This section describes how the state of a single process can be saved in a portable way. Section 5 describes how the uniprocessor checkpoints are coordinated to produce a global checkpoint.

To save the state of a process in a portable way, we must solve two problems. (i) How is the state of the process saved in the checkpoint? (ii) How do we translate the representation of the state of the process on one platform into its representation on a different platform?

Note that we cannot use system-level checkpointing as is done by systems like Condor [11]. The state that is saved by system-level checkpointers is very architecture-dependent (for example, the contents of the program counter, registers, etc. must be saved), so it is not possible in general to translate this representation of the state of the program into a representation that can be restarted on a different platform.

4.1 Application-level Checkpointing

The solution is to use *application-level checkpointing*, and our implementation of it is described in [2]. It is common to manually instrument long-running scientific programs so that they can save their state periodically and restore their state during recovery. Our system performs this instrumentation automatically, provided the programmer specifies where checkpoints should be taken.

For example, instead of saving the program counter, the application maintains a data structure that tracks function calls and returns. This data structure is stored as part of the checkpoint. The application is transformed so that on restart, the application reinvokes the sequence of function calls active at the checkpoint, thereby rebuilding stack frames and restarting execution at the point in the program just after the checkpoint. More details can be found in [2].

4.1.1 Type Representations

To translate checkpoint data from its representation on one machine to its representation on a different machine, it is necessary to know the type of every data object in the checkpoint. For each type in the program, our system maintains information about its representation in a structure called its type metric [14]. Our pre-compiler automatically produces the code to generate the type metrics for all the types of a program. This code is portable and it is executed to obtain the type metrics on that architecture. The type metrics for the C types are described below.

- Arithmetic Types: These include the integer and floating point types. Most modern architectures have standardized representations such as 2's complement for integers and IEEE single and double precision floating point for float and double respectively. Since we also record the endianness of the architecture, the size turns out to be sufficient as a metric. If an architecture uses its own proprietary format, we can resort to more specialized metrics.
- Pointer Types: Pointers in C are machine addresses and C implementations usually provide an integer type that has enough bits to hold a machine address. Hence, the type metric for pointers is similar to that of integers.
- Structure Types: C lays out the fields of a structure in the order they appear in the declaration of the type with an optional padding in between fields. Given two adjacent fields of a struc-

ture and a platform, the padding between the fields is fixed and is determined solely by the type of the structure. However, the padding might be different for different platforms. Type metric for structures includes the number of fields and for each field, its byte offset from the beginning of the structure and its type-metric(defined recursively). Bitfields can be handled as a special case by specifying the offset in bits.

- Union Types: Currently, we replace all unions with structures. In general, the two are not semantically equivalent. We assume that the overlapping of union fields in memory is not used to transfer data implicitly across different fields of the union.
- Array Types: In C, the elements of an array are laid out contiguously. Hence, the type metric for arrays consists of the type metric of an element of the array and the number of elements.

4.1.2 Checkpoint and Translation Mechanisms

Since we only handle well-typed programs, the data of the application can be regarded as a set of typed objects. Objects may be further classified as global objects, stack objects or heap objects depending on where they are located in the address space.

Our system maintains a set of descriptors, one per object. The descriptor of an object o contains the address of the object($ADDR_o$), its type metric(TYP_o) and its size in bytes(SIZ_o). A descriptor is added to the set when the object comes into existence and removed when it is destroyed. In our system, the set of object descriptors is distributed among the following three structures.

- SVD : Set of Stack Variable Descriptors organized as a stack into which descriptors are pushed and popped as the corresponding variables enter and leave the scope during execution.
- GVD : Set of Global Variable Descriptors organized as a list into which the descriptors for all the global variables are inserted when the program starts up. All the descriptors are removed at program termination.
- HOD : Set of Heap Object Descriptors organized as a red-black tree(with the object address serving as the key). Descriptors are inserted on allocation operations such as $malloc()$ and removed when deallocated by $free()$.

When a checkpoint is taken, all the objects are stored in binary mode along with the SVD, GVD and HOD. On restart, we construct the SVD and the GVD for the target architecture as part of the execution up to the point where the checkpoint was taken. We load the SVD, GVD and the HOD for the old architecture from the checkpoint. From the HOD loaded from the checkpoint, we can obtain the number of objects on the heap and the type metric and size of each object on the old architecture. We use this information to recompute the sizes and metrics on the new architecture. Then, we allocate space for the objects on the new architecture and construct the new HOD. At this point, we load each object from the checkpoint as is appropriate for its descriptor on the old architecture and translate it to the new architecture as described in Section 4.1.3.

4.1.3 State Translation

On restart on the target architecture, our system knows the descriptor for each object on the source and target architectures. The two descriptors contain all the information the system needs to do the translation. For integer and floating-point types, the translation achieves exact value equivalence if the value can be represented on the target machine. In case the value cannot be represented on the target architecture, we could adopt migration policies which disable heterogeneous migration for certain applications. An example of this is migrating an application that uses 64-bit integer values to a 32-bit platform that does not support 64-bit integer arithmetic. For the benchmarks used in this paper, we have not run into this problem.

For pointer types, the system needs to locate the corresponding object on the new architecture and set its value to the address of this object. For an object o, let $ADDR_o^s$, TYP_o^s and SIZ_o^s be the values of the descriptor fields corresponding to the source architecture and let $ADDR_o^t$, TYP_o^t and SIZ_o^t be the respective values for the target architecture. When a pointer p needs to be translated, our system searches the set of object descriptors on the source architecture to find the object it points to. If for some object o, $ADDR_o^s$ matches the value of p, then the value of p on the new architecture is $ADDR_o^t$. If the value p lies within the range of some object o which could for instance be a structure or an array ($ADDR_o^s < p < ADDR_o^s + SIZ_o^s$), a recursive search of its type information TYP_o^s is used to infer which sub-object p points to. The translated value of p is then equal to $ADDR_o^t$ incremented by the offset of the corresponding sub-object as obtained from TYP_o^t. If no matching object is found, the pointer is set to null.

Structure and array types are translated in a recursive manner by traversing the type information until we reach an integer, floating point or pointer type which is handled as discussed in the preceding paragraphs.

4.1.4 Efficiency

Since the translation of a pointer needs a search of the object descriptor set, it is crucial that we devise efficient data structures for performing this search. Before the translation phase, we sort the entries of the SVD and the GVD by address and determine the address range of the globals, the stack and the heap regions. The HOD is already sorted as it was serialized from a red-black tree. The sort has a cost of $O(NlogN)$ where N is the total number of objects. When a pointer needs to be translated, we determine from its address which region it points to. If it points to the stack or the globals, we do a binary search on the SVD or the GVD respectively. If the pointer points to the heap region, we search the red-black tree that corresponds to the HOD. For each object, searching takes $O(logN)$ and so the total cost of pointer translation is $O(NlogN)$. This scales well and for large pointer-dense address spaces, we see small overheads. Moreover, we incur this cost only on restart after a migration.

5. Co-ordination of MPI Checkpoints

To checkpoint MPI programs, we must co-ordinate state saving by different MPI processes. In addition, the checkpoint must capture the state of the MPI library, even though our system does not have access to the code of this library.

Both these functions are accomplished by the coordination layer shown in Figure 2. By design, this layer sits between the application and the MPI library, and intercepts all calls from the instrumented application program to the MPI library. This design permits us to implement the coordination protocol without modifying the underlying MPI library, which promotes modularity and eliminates the need for access to MPI library code, which is proprietary on some systems. Further, it allows us to migrate easily from one MPI implementation to another.

5.1 Protocols

The coordination layer supports the following two protocols (i) Barrier Coordination Protocol and (ii) Coordinated Non-blocking Protocol. The application programmer needs to select the appropriate protocol for his application. The coordinated non-blocking protocol is the more general of the two and does not require a global barrier in the application to take a checkpoint. Since it has been described before, we do not explain it here. The interested reader is referred to [2, 3]. The applications considered in this paper can be checkpointed using the lower overhead barrier-coordination protocol which we describe next.

The barrier-coordination protocol is suitable for programs that are written using the Bulk Synchronous Programming (BSP) model. In the BSP model, execution is divided into super-steps. Each super-step consists of a computation phase followed by a global communication phase followed by a global barrier. All communication initiated within a super-step completes before the completion of the synchronous barrier. For such programs, the barrier provides an opportunity to place a potential checkpoint location.

This protocol is initiated by a coordinator which is typically the process with rank 0. When the coordinator arrives at its potential checkpoint location, it makes a decision about whether a checkpoint should be taken. It broadcasts its decision in a message to all the other processes. Based on the decision of the coordinator, either all processes take a checkpoint at that barrier or they do not.

Note that this broadcast is itself a form of coordination, so it is not necessary to have an actual barrier in the MPI program. The barrier protocol is applicable to more programs than those which stictly adhere to the BSP model. It is sufficient if the program is SPMD with a repeating super-step structure and if the potential checkpoint location marks a spot in the super-step at which all communication calls issued previously have completed.

5.2 Portable MPI State Capture

The MPI library includes a number of opaque objects such as requests, statuses, data types, communicators, etc. The types of these objects are tied to the MPI implementation and are not portable. On restart, we need to ensure that these objects are correctly restored i.e. they represent the state of the communication as it was before the checkpoint.

To restore opaque objects, our system maintains a log of all operations performed on them. On restart, the log is used to recreate the objects and replay the operations so they get restored correctly. The logging mechanism was described in [15] and can be easily extended for portable checkpointing. Two features require special attention (i) derived data types and (ii) pack/unpack. Derived data types allow the user to specify data at non-contiguous locations by specifying displacements for the constituent locations. When specified in bytes, these displacements are non-portable. We log the values of the pointers used in constructing the displacements for the derived data types. The pointer translation algorithm from Section 4.1.3 can then be used to obtain the new values of these pointers and recompute the displacement on the target architecture. To support heterogenous pack and unpack operations, we implement our own pack and unpack routines that use a portable format for the data.

6. Experiments

To evaluate the performance of our system, we used the NAS parallel benchmarks [12]. We used $f2c$ to convert the benchmarks that were written in FORTRAN to C before instrumenting them with our pre-compiler.

The platforms we used were the following.

- Velocity : 128 node Velocity II cluster at the Cornell Theory Center running Windows XP. Each node is a Dual Processor 2.4GHz P4 Xeon with 2 GB RAM/Node, 72GB Disk/Node and 512KB Cache/Processor(SMP). The nodes are connected via Force10 Gigabit Ethernet. We used the GNU C compiler and the standard Windows threads for running virtual processes. The native MPI used was MPIPro Version 1.7.0.
- Lemieux : 3000 node Lemiuex cluster at the Pittsburgh Supercomputing Center running TRU64 Unix. Lemieux has quad-processor 750 Compaq Alphaserver ES45 nodes running at 1-GHz with 4GB RAM/node, and connected via a Quadrics interconnection network. We used the native HP C compiler and linked with the $pthreads$ library. The native MPI implementation on this platform is based upon MPICH and uses ELAN as the low-level communication library across nodes.

6.1 Selecting Appropriate Number of Physical Processors for Performance Evaluation

We measured the execution times of the NAS benchmarks on 4 through 128 processors. Our goal is to identify the region around which the benchmarks perform well so we can evaluate our over-decomposition system in that region. In this measurement, the codes are processed by $f2c$ but not by any of our preprocessing passes. Figure 3 shows the speed-ups, computed relative to running times on four processors(the size of the problems precludes running them on fewer processors).

The ep benchmark is embarrassingly parallel and scales perfectly. The other benchmarks perform a lot of communication and so scale to lesser extents depending on the amount of communication performed. They are also more memory-intensive. Hence, super-linear speedups are seen initially for some of the applications. This is because the data set of each process reduces as the number of processors increases leading to better memory hierarchy performance. ft and is communicate almost entirely via collective all-to-all operations. The remaining benchmarks communicate mainly through point-to-point calls. bt, lu and sp do nearest-neighbor communication. cg performs reduction operations which are implemented through point-to-point calls in the code.

From Figure 3, we observe that all these applications perform well in the 16 processor region. Therefore, in the rest of the experiments, we choose to configure the over-decomposed application to run on 16 processors and evaluate various aspects of our system.

6.2 Performance of Over-decomposition Subsystem

The next set of experiments measure the overheads associated with over-decomposition, keeping the number of physical processors (and therefore physical processes) fixed at 16. Table 1 shows how execution times on 16 processors of Velocity and Lemieux change as the number of logical MPI processes is varied from 16 to 256. The amount of over-decomposition therefore varies between 1 and 16.

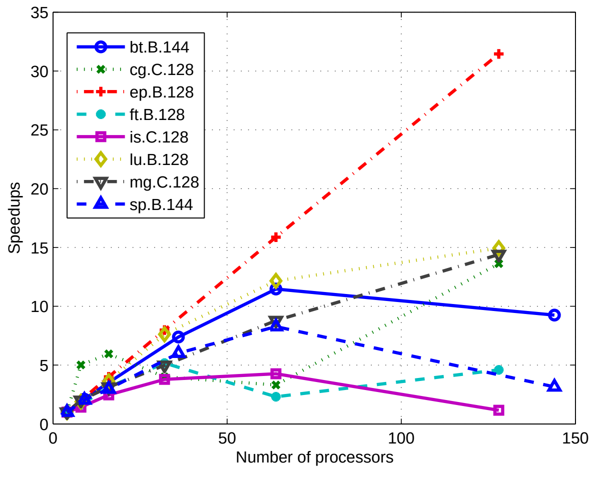
As the number of logical MPI processes increases, we would expect the amount of computational work per processor to stay roughly the same, provided this work is more or less equally distributed among the processors. However, depending on the benchmark, the amount of communication may increase. For benchmarks like ep that do not perform much communication, we would expect the execution time to be independent of the degree of over-decomposition; for benchmarks that perform a lot of communication, the execution time could increase.

The experimental results in Table 1 are consistent with these expectations. The execution time of ep is indeed independent of the degree of over-decomposition. For bt, is and ft, the overheads are small(less than 5% for 8 or 16-way over-decomposition). The codes is and ft do only collective communication and our system was able to map these into collective operations of the native MPI as explained in Section 3. For the other codes, the overheads are more significant. Like bt, these applications communicate primarily using point-to-point calls; however, they send more data per second.

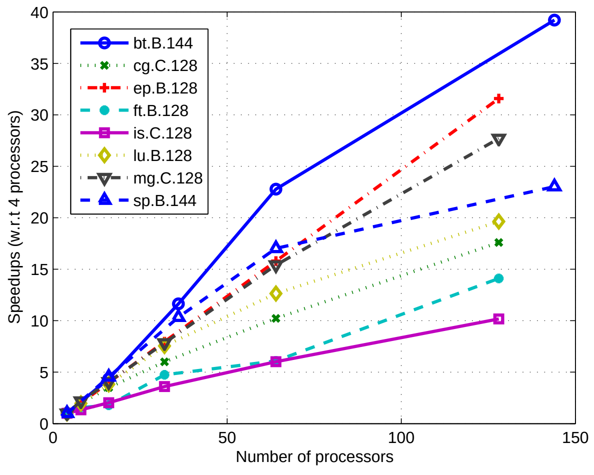
To understand the overheads of over-decomposition, we measured how much time was spent in different routines, the number of context switches, etc., as the degree of over-decomposition was varied. For lack of space, we consider only lu. Table 2 shows

	Velocity					Lemieux				
	$O_f = 1$	$O_f = 2$	$O_f = 4$	$O_f = 8/9$	$O_f = 16$	$O_f = 1$	$O_f = 2$	$O_f = 4$	$O_f = 8/9$	$O_f = 16$
bt.B	291.0	N/A	275.6	285.9	281.7	211.7	N/A	182.5	188.4	203.5
cg.C	176.0	147.6	194.8	206.0	332.1	140.7	138.2	161.8	170.9	205.2
ep.B	26.7	26.7	26.7	26.7	26.7	20.5	20.5	20.5	20.5	20.5
ft.B	56.0	56.2	56.8	56.3	57.3	44.8	45.5	43.5	41.2	41.8
is.C	20.0	21.0	21.9	21.0	21.0	19.9	19.6	19.0	18.7	20.8
lu.B	161.5	160.3	190.4	231.3	271.9	92.4	107.7	121.0	149.3	168.2
mg.C	54.5	71.5	65.5	70.0	75.4	34.7	41.7	37.0	40.0	43.2
sp.B	253.5	N/A	262.0	285.5	332.9	121.1	N/A	134.4	165.3	188.7

Table 1. Variation of execution time(in seconds) with the degree of over-decomposition(O_f) ranging between 1 and 16 on 16 physical processors. For bt and sp, O_f can only take values of 1, 4, 9 and 16 because these benchmarks must run on a square number of virtual processes. The other benchmarks must run on a power of 2 number of threads, so for these codes, O_f can take values of 1, 2, 4, 8 and 16.



(a) Velocity



(b) Lemieux

Figure 3. NAS speed-ups on Lemiuex and Velocity clusters

the measurements for various degrees of over-decomposition. The overheads attributed to code instrumentation for making globals thread-local was negligible(less than 1%) and so we do not show that.

Most of the time is spent computing and communicating (T_{bench}). Computing time may go up as the degree of over-decomposition is increased because the data cached by a given thread may be evicted by the execution of other threads. The amount of communication can also increase, as is shown in Table 2. The cost of context-switching between threads contributes a small amount to the overheads. The time spent blocking for messages initially decreases and then increases. The decrease may be attributed to the ability to switch to some other thread which is ready to compute. For $O_f > 4$, this benefit is offset by the increase in communication to the extent that the application cannot find enough computation to overlap with the communication.

It is important to note that the assignment of logical MPI processes to physical processors can affect performance; for example, the number of network messages depends on this assignment. In allocating threads to processors, we used the block distributions for bt, ep, ft, is and mg. We used block-cyclic distributions for bt, lu and sp since these performed better than the block distribution. We are experimenting with more elaborate heuristics such as co-locating threads that communicate a lot with each other.

6.3 Performance of Checkpointing System

In this section, we measure the overheads incurred by our checkpointing system. Since these overheads are orthogonal to those incurred by over-decomposition, we restrict the over-decomposition factor to 1 for this subsection. We evaluated our system on 4, 16, 64 and 256 processor jobs. For lack of space, we show running times for only some of the configurations we explored.

Table 3 shows the sizes of checkpoints for 64 processor configurations on Lemiuex. Figures 4(a) and (b) shows the running times on Lemieux and Velocity respectively. For each benchmark, we show the running time of the original application ($Baseline$), the running time of the instrumented application without taking any checkpoints ($PC^3(0ckpt)$), the running time of the instrumented application taking one checkpoint ($PC^3(1ckpt)$), and the running time of the application taking one checkpoint and restarting immediately from it($PC^3(1ckpt + restart)$). The overhead of instrumentation can be determined by subtracting the time for $Baseline$ from the time for $0ckpt$. The overhead of taking a checkpoint can be inferred by subtracting the time for $0ckpt$ from the time for $1ckpt$. The time spent in restarting from a checkpoint can be obtained by subtracting the time for $1ckpt$ from the time for $1ckpt + restart$.

O_f	T_{tot}	T_{bench}	T_{block}	T_{csw}	N_{csw}	N_{loc_msg}	$AvgSiz_{loc\_msg}$	N_{nw_msg}	$AvgSiz_{nw\_msg}$
1	161.5	117.9	43.6	0.0	0	0	N/A	53500	2.9K
2	160.3	119.3	38.5	2.5	67420	50508	3.1K	78750	2.0K
4	190.4	142.1	42.7	5.6	254815	202024	1.5K	104000	1.5K
8	231.3	165.8	55.5	10.0	596097	505056	1.3K	154000	1.1K
16	271.9	185.0	68.9	18.0	1341041	1212120	0.8K	204000	0.8K

Table 2. Detailed performance breakdown for lu on 16 physical processors on Velocity for various degrees of over-decomposition(O_f). The total execution time(T_{tot}) is broken down into times spent in benchmark execution (T_{bench}), blocking on network messages (T_{block}) and context switches (T_{csw}). All times are in seconds. Also tabulated are the number of context switches(N_{csw}), the number and average size of local or intra-process messages(N_{loc_msg} and $AvgSiz_{loc\_msg}$) and the corresponding values for network of inter-process messages(N_{nw_msg} and $AvgSiz_{nw\_msg}$).

bt.C	cg.C	ep.D	ft.C	is.C	lu.C	mg.C	sp.C
76.5	42.5	1.1	117.5	37.8	16.5	456.9	40.1

Table 3. Checkpoint sizes in MB per processor for 64 processor runs on Lemieux.

As can be observed from the graphs, the instrumentation overheads are negligible for all the benchmarks. This implies that the checkpointing system causes little slowdown to the normal execution of the program. The overheads of state-saving depend only on the size of the checkpoints and the frequency of checkpointing. In these measurements, all checkpoints were stored on the local disk of each processor. In general, the overheads are small and are noticeable only for mg and ft on the larger configurations since the size of their checkpoints is large relative to the execution time of the application. For typical checkpointing frequencies such as once per hour, we expect state-saving costs to be manageable.

6.4 Migration Flexibility

Heterogeneity To test the porting of parallel jobs between different platforms, we restarted Lemieux checkpoints on the Windows cluster, using IBP [13] to move data between the sites. Figure 4(b) shows the time required to restart 64 processor Lemieux checkpoints locally on Velocity and run to completion, compared to the time required if native Velocity checkpoints were used. The cost of translating the state can be seen to be negligible. In our experience, the major bottleneck in this style of utility computing is the cost of transferring data between sites which varied from a few minutes to many hours depending on the size of the checkpoints. To reduce this cost, it is imperative to reduce the amount of saved state, and to find faster solutions for transferring data between platforms.

Restarting on Different Number of Processors To demonstrate the flexibility afforded by restarting on different numbers of processors, we used 16 processors on the Velocity cluster to start up jobs with 128 or 144 logical MPI processes, and took checkpoints in the middle of their execution. We used those checkpoints to restart on 8, 12, 16, 20, 24, 28 and 32 Velocity processors. We repeated the same experiments on the Lemieux cluster. Figure 5 shows the speed-up relative to 8 processors as the number of physical processors is increased. Apart from cg on Velocity, the execution times decrease as more processors are used. For ep which is embarrassingly parallel, we obtain near perfect speedups. For the other benchmarks which have more communication, the speedups obtained are less than perfect and the exact value depends on the nature of communication and synchronization, which do not scale perfectly for the benchmark.

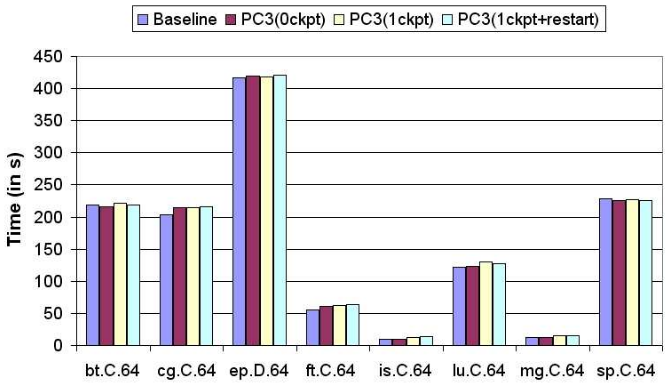
For cg on Velocity, we obtain speedups when the load (consisting of 128 threads) is perfectly balanced as is the case with 8, 16 and 32 processors. However, the performance of the application deteriorates when the load is not perfectly balanced. We suspect that this may be attributable to the base MPI implementation on the platform because the performance of cg on the base MPI is itself somewhat anomalous as can be seen in Figure 3. This anomaly is not observed on Lemieux.

7. Related Work

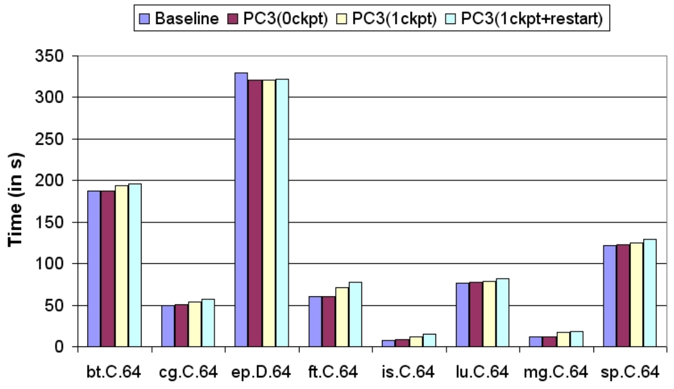
The PC^3 system described in this paper builds upon the C^3 system [2] which integrated application-level checkpointing with a coordinated non-blocking protocol to checkpoint MPI applications. The C^3 checkpointer used mechanisms to ensure that objects were located at the same virtual memory addresses on restart thus permitting objects to be copied without need for pointer translation on the same architecture. PC^3 extends the C^3 checkpointer by using type information to restart on a different architecture, a feature that C^3 could not support. The coordinated non-blocking protocol described in [2] was extended to support collective MPI operations in [3]. A scalable implementation of the protocol was provided in [15]. This paper describes three significant advances over our previous work, (a) the design of a lightweight virtual MPI system that supports efficient over-decomposition using threads allowing restart on a different number of processors, (b) the incorporation of techniques for enabling heterogeneous Application-Level Checkpointing, and (c) experimental results that demonstrate that (a) and (b) do not add significant overheads to the base application.

AMPI [6] and TMPI [18] are examples of other systems that provide support for over-decomposition. AMPI relies on the CHARM++ framework to support automatic load-balancing. To support transparent thread migration, it requires that a given thread always start at the same address and so it partitions the virtual address space across all the threads. For applications with a large number of threads, this could become a limitation, especially on 32 bit platforms. Since, our system recomputes all the pointer addresses, it does not have this restriction. Also, our pre-compiler provides support for portable checkpointing in a transparent fashion, a feature that none of these systems currently support. TMPI changes the internals of MPICH to support thread endpoints. It cannot leverage other implementations of MPI that may be efficient for the target platform. Further, it does not provide any support for checkpointing.

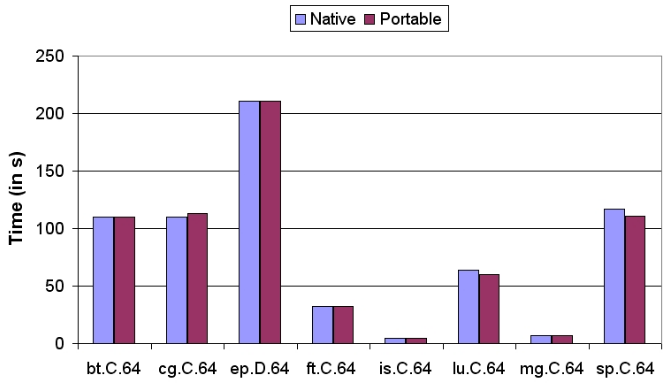
Alvisi et al [10] is a good survey of work on checkpointing and related techniques like message-logging. Most checkpointing systems in the literature use system-level checkpointing (SLC), so they do not provide a solution to the problem of moving state between heterogenous platforms. The most commonly used checkpointing system in grid environments is Condor [11], which takes system-level checkpoints of sequential programs. Condor provides support for heterogeneous migration, provided the application is written in Java, by leveraging the portability of the JVM. This approach is problematic for legacy high performance codes written in



(a) Velocity



(b) Lemieux



(c) Lemieux to Velocity

Figure 4. Overheads of checkpointing on 64 processors. Figures (a) and (b) shows the overheads on Lemieux and Velocity respectively. Figure (c) compares the overhead of restarting on Velocity from a checkpoint taken from Lemiuex as opposed to a native checkpoint.

C and Fortran. CoCheck[17] leverages SLC to checkpoint MPI applications by providing mechanisms to coordinate distributed state. However, it has not been integrated with the JVM approach to provide a solution for heterogeneous migration of MPI programs.

Previous systems for portable checkpointing of C programs include Porch [14], April [4], HPCM [8], MigThread [9] and Tui [16]. These systems use type-information similar to PC^3 although the details of their implementation are different. Of these, Porch, April, HPCM and MigThread use an automated pre-compiler approach. The Tui system relies on modifications made to the compiler to extract type information. With the exception of April, none of these systems handle MPI programs. The April compiler was combined with a fault-tolerant MPI implementation to checkpoint MPI applications for the Legion system. However, support for over-decomposition was not provided.

SRS [19] is a checkpointing infrastructure that uses the Internet Backplane Protocol [13] for storing and migrating checkpoints. Its allows the programmer to manually specify the data that needs to be shared as well as its distribution. On recovery, the system uses this information to recover on a potentially different number of processors. The Dome system [1] is a C++ library of data-parallel objects that supports checkpointing and restart using application level checkpointing. All objects that need to be checkpointed should be defined as special Dome objects capable of saving their state in portable XDR format. Legion [7] is another system that provides naming and location services so that applications programs can be written without any awareness of where objects reside. To allow objects to be migrated between heterogenous resources, Legion requires that every object provide `save_state()` and `restore_state()` methods. The legacy applications that we are targeting are not written using these frameworks.

8. Conclusions

In this paper, we have described the PC^3 system for migrating C MPI programs in computational grids. PC^3 provides an efficient virtual MPI that uses threads as its endpoints to enable efficient redistribution of load on restart to a different number of processors. The PC^3 pre-compiler automatically transforms the input program to save its own state making its use transparent to the application programmer. The system uses type information to save the state of the MPI processes in a portable manner, and uses either barrier or non-blocking protocols for coordinating the MPI state. Our experimental results indicate that the overheads of checkpointing are less than 2% on average for the benchmarks used in the paper. The overheads caused by over-decomposition, though high for some applications, provides them with additional flexibility for migration.

In future work, we plan to explore the effect of alternative scheduling policies on performance. We also plan to study how communication pattern information can be exploited to obtain better strategies for load distribution. We would like to evaluate our system on applications that have more irregular and asynchronous communication. Our eventual goal is to extend our system so that it can handle all of MPI in an efficient and scalable manner.

The sizes of MPI checkpoints can get very large as the problem sizes and the number of processors are increased(as high as 100GB). The major bottleneck to mobility is in transferring these large checkpoints over the network. One way to reduce the checkpoint size is to save only the live objects. Further, some objects may be recomputed cheaply on recovery and so need not be included in the checkpoint. We are investigating static analyses that can identify such objects. We are also interested in technological solutions for improving transfer bandwidth.

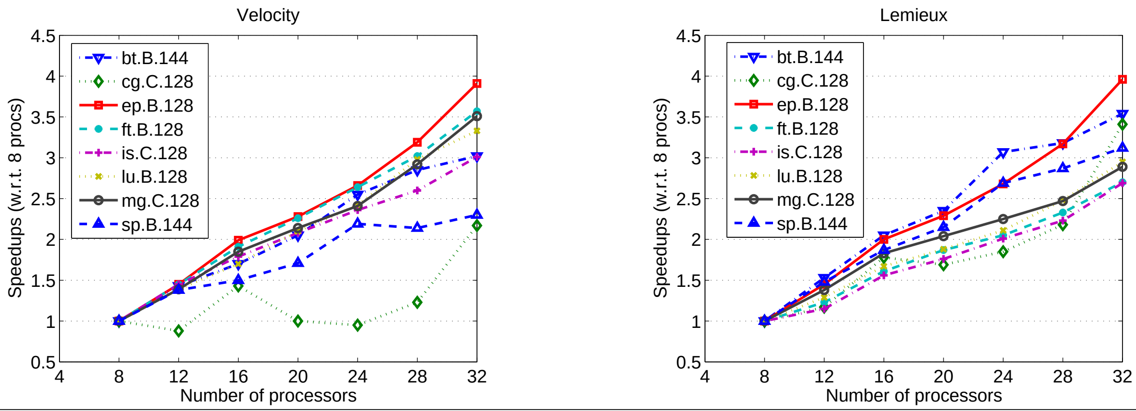


Figure 5. Speedups (relative to 8 processors) obtained by restarting on different numbers of physical processors. The times measured are the execution times after restoring data from a checkpoint taken midway in the application run originally on 16 processors.

References

[1] Adam Beguelin, Erik Seligman, and Peter Stephan. Application Level Fault Tolerance in Heterogeneous Networks of Workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[2] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated Application-level Checkpointing of MPI Programs. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP 2003)*, 2002.

[3] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective Operations in an Application-level Fault Tolerant MPI System. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[4] Adam Ferrari, Steve J. Chapin, and Andrew S. Grimshaw. Heterogeneous Process State Capture and Recovery through Process Introspection. In *Cluster Computing*, volume 3, 2000.

[5] Tom Goodale, Gabrielle Allen, Gerd Lanfermann, Joan Mass, Thomas Radke, Edward Seidel, and John Shalf. The Cactus Framework and Toolkit: Design and Applications. In *VECPAR*, 2002.

[6] Chao Huang, Orion Lawlor, and L. V. Kale. Adaptive MPI. In *Languages and Compilers for Parallel Computers (LCPC)*, 2003.

[7] Michael J. Lewis and Andrew Grimshaw. The Core Legion Object Model. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996.

[8] Kasidit Chanchio and Xian-He Sun. Data Collection and Restoration for Heterogeneous Process Migration. In *Softw., Pract. Exper.*

[9] Hai Jiang and Vipin Chaudhary. Process/Thread Migration and Checkpointing in Heterogeneous Distributed Systems. In *HICSS 2004*.

[10] 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.

[11] J. Basney M. Litzkow, T. Tannenbaum and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report 1346, University of Wisconsin-Madison, 1997.

[12] NASA. NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB/`.

[13] James S. Plank, Micah Beck, Wael R. Elwasif, Terry Moore, Martin Swany, and Rich Wolski. The Internet Backplane Protocol: Storage in the Network. In *NetStore99: The Network Storage Symposium*, Seattle, WA, 1999.

[14] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogenous Architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[15] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and Evaluation of a Scalable Application-level Checkpoint-recovery Scheme for MPI Programs. In *Supercomputing 2004*, Pittsburgh, PA, November 6–12 2004.

[16] Peter Smith and Norman C. Hutchinson. Heterogeneous Process Migration: The Tui System. *Softw., Pract. Exper.*, 28, 1998.

[17] Georg Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[18] H. Tang and T. Wang. Optimizing Threaded MPI Execution on SMP Clusters. In *Proceedings of 15th ACM International Conference on Supercomputing*, pages 381 – 392, 2001.

[19] S. Vadhiyar and J. Dongarra. SRS - A Framework for Developing Malleable and Migratable Parallel Software. *Parallel Processing Letters*, 13(2):291–312, June 2003.