Efficient, Language-Based Checkpointing for Massively Parallel Programs

Sanjeev Krishnan
Department of Computer Science,
University of Illinois, Urbana-Champaign.
email : sanjeev@cs.uiuc.edu
Phone : (217) 333-5827

Laxmikant V. Kale
Department of Computer Science,
University of Illinois, Urbana-Champaign.
email : kale@cs.uiuc.edu
Phone : (217) 244-0094

Abstract

Checkpointing and restart is an approach to ensuring forward progress of a program in spite of system failures or planned interruptions. We investigate issues in checkpointing and restart of programs running on massively parallel computers. We identify a new set of issues that have to be considered for the MPP platform, based on which we have designed an approach based on the language and run-time system. Hence our checkpointing facility can be used on virtually any parallel machine in a portable manner, irrespective of whether the operating system supports checkpointing. We present methods to make checkpointing and restart space- and time-efficient, including object-specific functions that save the state of an object. We present techniques to automatically generate checkpointing code for parallel objects, without programmer intervention. We also present mechanisms to allow the programmer to easily incorporate application specific knowledge selectively to make the checkpointing more efficient. The techniques developed here have been implemented in the Charm++ parallel object-oriented programming language and run-time system. Performance results are presented for the checkpointing overhead of programs

running on parallel machines.

Keywords : Checkpointing, rollback recovery, parallel object-oriented techniques, massively parallel software, object state reduction.

1 Introduction

Parallel computers are increasingly being used to satisfy the growing computational needs of scientific and commercial applications which are at the edge of computational feasibility. A large number of applications have been made tractable by massively parallel computers such as the TMC CM5, Intel Paragon, nCUBE/2 and many others. Such applications often have *long running times*, taking a few days or even longer for production runs, especially if run on smaller parallel computers which are more easily accessible.

Reliability on massively parallel machines is more difficult to ensure than in the sequential case, because an increase in the number of components causes the mean time to failure of the system as a whole to decrease. When an application aborts due to a fault, restarting it from the beginning is wasteful, and still provides no guarantee of the application executing to completion.

In order to reliably execute long running applications on systems which may fail, one approach is to keep a record (a *checkpoint*) of the program state at intervals. The checkpointed state is usually saved in stable storage. When a fault occurs, the program can be rolled back to the last checkpoint and restarted, thus ensuring forward progress if faults are less frequent than checkpoints.

In the massively parallel computing world, checkpointing and restart provides other important benefits. Often users run their applications in dedicated time slots on massively parallel computers in national laboratories and other supercomputer centers. Checkpointing and restart allows longer applications to be executed than a single time slot would allow. Another potential use is when different phases of a parallel program require different resources which are available on different machines. Instead of writing a separate program for each machine or occupying all the machines with a single "heterogenously" executing program, the program can be checkpointed on one machine and then restarted on another, thereby making the resources of the second machine available to the program.

1.1 Checkpointing for massively parallel systems

Although there has been extensive research into checkpointing and rollback recovery in the past few years in the context of distributed systems [KT87, BP93, Joh93], some new issues have to be considered in the context of massively parallel systems :

1. Most massively parallel systems do not support checkpointing at the operating system or hardware level, leading application programmers to implement checkpointing in their application, using existing OS mechanisms.

2. Portability of parallel applications across different parallel computers is an essential requirement because many real applications need to be run on more than one parallel computer. Proprietary solutions that depend on a machine-specific feature are hence not acceptable.

3. Massively parallel systems have a large number of processors, because of which the state of the parallel program is very large. So a large amount of stable storage would be required for saving the checkpointed state.

4. Processors in massively parallel systems are usually more tightly coupled, and have less autonomy. The processes in a parallel program are usually instances of the same program executing in a loosely synchronous, SPMD manner. Inter-processor communication latencies are small,

of the order of tens of microseconds on typical systems. So it is acceptable to synchronize the whole computation for checkpointing.

Thus massively parallel systems provide different opportunities for optimizing a checkpointing facility.

2 Goals of a Checkpoint and Restart facility for MPPs

Taking into account the different issues that arise while checkpointing massively parallel programs, we have formulated the following goals for a checkpointing facility for such a platform :

Space efficiency: Massively parallel machines have hundreds or thousands of processors, and for each checkpoint, every processor can potentially generate a checkpoint file. If each processor were to simply write out its entire memory space (equivalent to a "core dump" on Unix computers which can require Megabytes of storage), the total disk space requirement would be in Gigabytes for each checkpoint, which is clearly not practical. Hence a checkpointing facility should try to minimize the size of the checkpoint file.

Time efficiency: Checkpointing introduces a performance overhead, a major portion of which is the I/O time required for writing out the parallel program state to disk. The cost of coordinating processors at the beginning or end of the checkpoint also adds a small overhead to the checkpointing process. It is desirable to reduce the performance overhead of checkpointing and achieve time-efficiency. While the I/O overhead can be minimized by reducing the size of the program state that has to be saved, and also by using asynchronous parallel I/O, the coordination overhead can be reduced by overlapping computation with the coordination step.

Language-based: To date, most checkpointing facilities have been implemented in the operating system, sometimes with hardware support. This is not feasible for massively parallel programs because of the lack of vendor support for checkpointing and because of the need for portability. As a response to these two problems we have based our approach on the language and run-time system level. Moreover, language-based checkpointing can give significant advantages in terms of space efficiency and incorporation of application specific knowledge. Thus language-based checkpointing effectively meets the problems raised by the first three issues described in section 1.1.

Automation: A language-based checkpointing facility should automate the task of writing checkpointing code so that the programmer does not *have to* write code separately for each application. It should be possible to use the language-based checkpointing facility with the same ease as a hardware or OS based facility.

Programmer control: Recognizing that the application programmer sometimes knows best when to checkpoint and what part of the program state to checkpoint, the programmer should be allowed to control these decisions, *only if so desired.* In general, it is desirable to increase the efficiency of a checkpointing facility by allowing the programmer to provide application-specific knowledge which is not available to the operating system or hardware.

3 Our approach to checkpointing and recovery

In this section we describe our approach to achieving the goals motivated in the previous section. Our approach is oriented towards incorporating checkpointing and recovery at the language and run-time system level. Our discussion concentrates on the checkpointing phase; recovering from a checkpoint simply requires each processor to restore the program state from the checkpoint file, and does not require any coordination between processes. Since we have implemented our approach in the Charm++ language, a brief overview of Charm++ is first given. We emphasize, however, that the techniques we describe can also be applied to other languages.

3.1 The Charm parallel programming model

Charm and Charm++[Kal90, KK93] are explicitly parallel object-oriented programming languages based on C and C++, respectively. The basic unit of work in Charm++ is a *chare*, which is similar to a concurrent object. Chares are dynamically created and communicate with other chares by messages. There can be hundreds or thousands of chares on every processor. A chare type is like a C++ class, and contains data, functions and *entry points*, which are special functions where messages can be received. The Charm and Charm++ languages are supported by the Chare Kernel run-time system[FRS+91]. Charm and Charm++ programs run without change on shared as well as private address space parallel computers. Currently, the platforms supported include the TMC CM-5, Intel Paragon, IBM SP-1, nCUBE/2, workstation networks, Sequent Symmetry and Encore Multimax. The Charm run-time system is being ported to the Cray T3D and Convex Exemplar.

An essential feature of the Charm parallel programming model is *asynchronous message driven execution*. Recognizing that latency of remote communication is a significant cause of performance degradation, the Charm model is oriented towards overlapping communication and computation. All calls to the run-time are non-blocking and there are no "receive" calls. Remote accesses are performed by a split-phase transaction: a chare sends a request message to a second chare and returns control to the run-time. When the second chare replies, the run-time schedules the first chare for execution again. Thus when one chare is waiting for a message from a remote chare, another can be scheduled for execution. The run-time system maintains a pool of messages; user computations (entry-points) are executed on receipt of messages (hence the message-driven model), and once started, an entry-point is not pre-empted.

3.2 Invoking a Checkpoint

The best *location* (i.e. position in the parallel program) of a checkpoint is application dependent. Often the user may want to invoke a checkpoint at the end of one phase of a computation when the state of the program is smallest. The appropriate checkpoint *interval* (i.e. how often to checkpoint a program) depends on both the frequency of faults and the overhead due to a checkpoint. In addition to periodic checkpointing, the user may want to do on-demand checkpointing. Our checkpointing facility can be invoked by a run-time system call : "void CCheckPoint() ", which can be made from any part of the application code on any processor. It can also be invoked by the run-time system at intervals using the command line parameter "+interval" which specifies the checkpoint interval in seconds. The checkpoint phase is started as soon as control returns to the run-time system after the interval has passed.

3.3 Steps in coordinated checkpointing

As discussed in Section 1.1, in massively parallel systems, all processors usually run an instance of the same program, and the communication latencies between processors are small. Moreover, a fault on one processor usually makes the whole system unusable. Hence it is sufficient to coordinate all the processors to attain a consistent parallel program state. We employ a coordination strategy having the same high level steps as in [CL85]. However, our algorithm is more general in that it does not assume that messages are received in the order in which they are sent.

Figure 1 describes the our coordination strategy with a time-line diagram. The steps involved in a coordinated checkpoint are :

1. *Broadcast Initiate* : The coordinating processor broadcasts a message to all other processors to initiate a checkpoint.

2. *Stop all processing* : When a processor receives the "initiate checkpoint" message it stops all processing and does not send out any more application program messages; these messages are just received and logged.

3. *Start saving state* : Processors start saving their state (including memory state) to disk.

4. *Detect Quiescence* : A necessary condition for ensuring a consistent state of the parallel program (after all processors have stopped executing application code) is that all messages which have been sent must also have been received. In the absence of low level support for clearing the network and if machine independence is desired, the processors need to wait till all messages in transit are received. A scalable *quiescence detection* algorithm [SKR93] is used to detect this. Since message-order reversal is possible, marker messages cannot be used to indicate that a channel is cleared. Hence the quiescence algorithm is based on counting the total number of messages sent and received over all processors. When quiescence is detected, all pre-checkpoint messages have been received. The coordinating processor then broadcasts a Quiescence message to all processors whereupon they start saving their message states to disk.

5. *A final synchronization* : After all processors have received the Quiescence message, they inform the coordinating processor, which then broadcasts a "resume" message. This final synchronization is required to prevent post-checkpoint messages from reaching a processor before the Quiescence message[1]. On receiving the resumption messages, processors unfreeze their state and continue with the application.

One additional problem faced during checkpointing is that of logging messages which have

[1]The final synchronization ensures that all messages received before the Quiescence message are pre-checkpoint messages. Thus we do not need to tag messages as pre- or post-checkpoint messages.

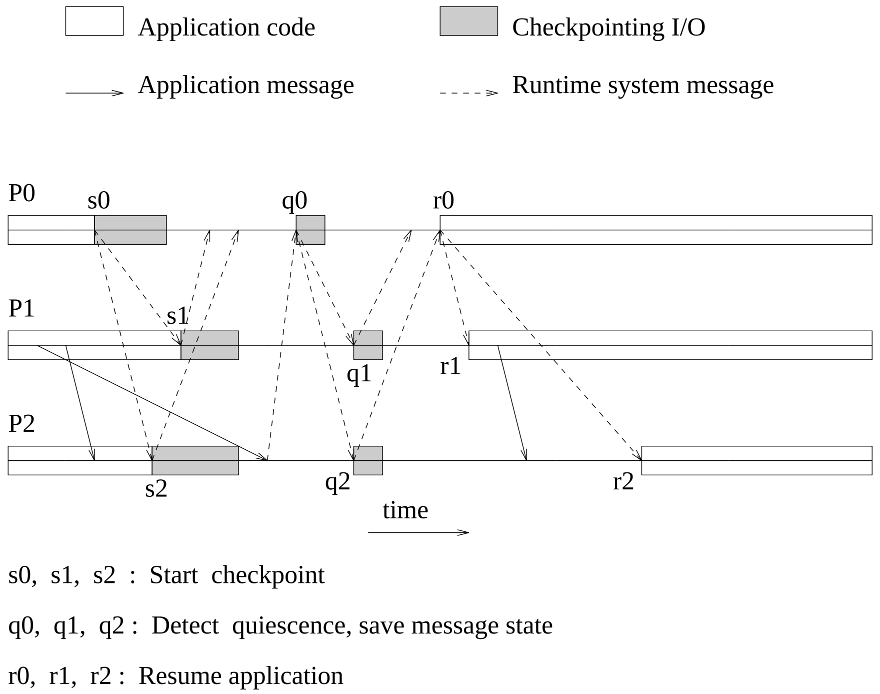

Figure 1: Time-line for processor coordination steps.

been received but are in system buffers. In the traditional SPMD style, the checkpoint code will have to execute a string of "receive" statements with wildcards and save the messages thus received. After restart, to deliver these messages back to the user program, the user code needs to be modified to pick up messages that were received but not processed before checkpointing, leading to significant code change just for the purpose of checkpointing. Charm++, however, prevents this problem because of its message-driven model. We have modified the Charm++ runtime system to receive messages from the network while checkpointing, enqueue them separately, save them in the checkpointed state, and deliver them back into the system queues for processing.

3.4 Reducing the state of a parallel program

The actual process of writing out the program state on each processor is now explained. For the purpose of checkpointing, we need to identify the state of a parallel program. Since there are no messages in transit after all processors have been coordinated, the state of the parallel program is simply the sum of the states of all processors. On each processor, the state includes the stack of procedure activation records; dynamically allocated memory (the heap); and registers including the program counter. Of these, the stack and the heap contribute the largest portion of the state. In this section we describe how to reduce the stack state, and the next section describes our techniques for reducing the state due to dynamically allocated objects in the heap.

The following observations can be made about the stack in the context of checkpointing :

- The stack state is temporary : often the stack state can decrease considerably in very little time when the program returns from a deep nest of procedure calls. Hence to reduce the stack state we should checkpoint the program at a location where it is in a shallow level of nesting.

- Method invocations in a object oriented program are usually fine- or medium-grained. This means that in order to reduce the stack state, it is acceptable to wait for a method invocation to complete before checkpointing the processor.

- Operations for saving and restoring the stack can be non-portable, since the stack pointer is usually in a machine register, and different compilers have different conventions for allocating and de-allocating procedure activation records.

As described earlier, the Charm model is a message-driven one, in which messages are directed to objects, and invoke methods in them. The run-time system thus has a pool of messages, and invokes a medium-grained method in a user-object for each of them. Based on this model and the observations in the preceding paragraph, we have designed the following scheme to reduce the stack state : a checkpoint is taken only when control has returned from a user-object to the run-time. This ensures that there is *no user stack state* when a consistent parallel program state is reached.

The Charm++ run-time system's stack state is known, and we have developed special code to save its data structures. Thus we not only attain much greater space efficiency, we can also use more portable techniques.

4 Reducing dynamically-allocated object state

In this section we describe how to reduce the size of the dynamically allocated heap state of a processor that needs to be saved. In an OS or hardware provided checkpointing facility, the heap state is saved by simply writing out to disk all pages in memory that are allocated to the heap. This is equivalent to a "core dump", and can occupy a large amount of storage because it is likely that the whole of a memory page is not used.

In our checkpointing facility, we have used our own memory-manager which keeps track of dynamic memory allocation, freeing, and management of free memory blocks. This provides a mechanism for saving objects in the program heap state : while checkpointing, the memory manager writes to disk only allocated memory blocks. While this first scheme alone is better than storing memory pages, it will not work if memory-block addresses change when the program is recovered from the checkpoint. This is because pointers to memory may not be valid after the program is restored. We now discuss a scheme which can modify pointer values.

Recognizing that the state of an object is known to its member functions, we can devise a second scheme, which can intelligently use information from the application about objects in the program. We define an object-specific function which saves the state of the object (called a *Pack* function), and another complementary function which restores the state of the object (called the *Unpack* function). The pack function transforms the object into a contiguous array of bytes, and the corresponding unpack function reconstructs the original structure from the byte array.

Pointer linked data structures such as linked lists or trees can packed by invoking the pack function on each item in the structure in a recursive manner. Thus a complicated structure can be packed by traversing all pointers in the structure. However, this cannot handle aliased pointers (more than one pointer to the same address) and structures with circularities such as a circular linked list : the latter can result in an infinite loop of pack functions. So we need a better scheme.

In our third scheme we combine the capabilities of the memory manager with the capabilities of pack functions in the following manner. The memory manager maintains the heap as a collection of *typed blocks*. A type is assigned to a block at the time it is allocated. While checkpointing, the memory manager uses the type field to invoke the correct pack function on the object. The type field is also saved; while recovering the program from the checkpoint the type field is again used to invoke the correct unpack function. The memory manager also invokes unpack functions in exactly the same order as the pack functions : this ensures that only a linear access is required to the saved disk file; also it obviates the need for storing each block's location, thereby saving some space.

In this third scheme, pointers to other elements in a pointer-linked structure are not followed : i.e. there is no recursive traversal of the pointer linked structure. Instead, a pointer in an object is written out as a special primitive data type by the pack function for that object. Now while recovering the program from the checkpoint, we need to make sure that the saved pointer is converted to a new valid pointer. The memory manager enables this by ensuring that the memory map, i.e. the relative addresses of allocated blocks remains the same when the program is recovered from a checkpoint. So a saved pointer needs to be simply shifted by a constant amount in order to be valid. The third and final scheme can thus handle all cases successfully.

Automatic pack function generation : The most efficient pack function for a data structure depends on its internal structure and semantics. However, correct pack functions can be automatically generated by the compiler for all data structures. We extended the Charm++ compiler to automatically generate pack and unpack functions for all complex data types. Pack functions for primitive data types such as integers, floating point numbers and pointers are provided by the run-time library. For complex data types (structures or classes in C++), since the compiler knows the type of each field, it generates a call to the appropriate pack function for storing that field. Figure 2 shows a C++ linked list class and automatically generated pack and unpack functions for it. With pack functions for every object, the entire pointer-linked object state of the Charm++ program can be saved to disk by calling the pack function for all the allocated memory blocks on a processor.

Providing programmer control with pack functions : Pack functions (either automatically generated or programmer defined) provide a powerful mechanism for minimizing the size of the saved application dependent object state. E.g. in many iterative computations involving operations on a matrix, each processor computes a new matrix on every iteration from the previous one. Often two matrices are maintained, so that the old matrix is not destroyed when the new one is computed, and each matrix is computed from the other on alternate iterations. Thus it is possible for the application programmer to reduce the size of the object state considerably by checkpointing only one of the two matrices. Again, programmers commonly allocate more memory than they need for arrays and other structures whose size is not known beforehand; in this case too, the saved data can be considerably smaller than the actual memory space allocated. Application programmers may write their own pack and unpack functions for selected objects when they desire to minimize the checkpointed object state. For the remaining objects the Charm++ compiler automatically generates pack functions. This method thus provides the application programmer complete control while at the same time automating the process to the extent desired.

```
class ListElement {                  =>      class ListElement {
        int data ;                                    int data ;
} ;
                                                public:
                                                    virtual void Pack()
                                                    {
                                                            PackInteger(data) ;
                                                    }

                                                    virtual void Unpack()
                                                    {
                                                            UnpackInteger(&data) ;
                                                    }
                                             } ;

class List {                         =>      class List {
        ListElement element ;                       ListElement element ;
        List *next ;                                List *next ;
} ;
                                                public:
                                                    virtual void Pack()
                                                    {
                                                            element.Pack() ;
                                                            PackPointer(next);
                                                    }
                                                    virtual void Unpack()
                                                    {
                                                            element.Unpack() ;
                                                            UnpackPointer(&next);
                                                    }
                                             } ;
```

Figure 2: Simple automatically generated Pack and Unpack functions for a linked list class. The Pack and Unpack primitive functions are part of the I/O library in the run-time system.

5 Performance results

We have implemented our checkpoint and recovery techniques in the Charm++ parallel programming system by modifying the Charm++ translator and the run-time system for adding checkpointing functionality. The checkpointing code is machine-independent. Our implementation has been tested on a network of Sun workstations, TMC CM-5, nCUBE/2 and the Intel Paragon. Performance results are presented here for a 64 processor CM-5. The programs for which checkpointing was evaluated are :

- Jacobi : a simple implementation of the Jacobi method for solving a 5 point stencil problem. The subdomain on each processor is a 16 x 16 grid.

- TSP : a branch-and-bound solution for the Traveling Salesperson Problem. The input is a 20 cities problem.

These programs were run without any source code change for the purpose of checkpointing, except for the actual checkpoint invocation using the "CCheckPoint() " call. Table 1 presents results for average per-processor checkpoint file size in Kilobytes and checkpointing overhead in seconds for the above Charm++ programs. One checkpoint file per processor was created on a single common disk. The overhead includes the time for the coordination steps in Section 3.4 as well as the actual sequential I/O time.

Program	Quantity	Program state size	Checkpoint file size
Jacobi	File size	36.0 K	7.0 K
	Overhead		15.4s
TSP	File size	41.5 K	14.5 K
	Overhead		21.5s

Table 1: Checkpointing overhead (seconds) and average per-processor file size (Kbytes) on a 64 processor CM-5.

From the results for file size it is clear that the pack function method is able to substantially reduce the size of the checkpoint file by saving only application specific objects and using application dependent information to further reduce file size.

The overhead for checkpointing is large, and is primarily due to the sequential I/O bottleneck of writing on a shared disk, and other operating system reasons. The I/O time was observed to be greater than 95% of the overhead for most runs. For these experiments we used simple Unix blocking I/O with no optimizations, since our emphasis in this work was on reducing file size.

6 Previous work

Plank and Li [PL94] have developed one of the only implementations of checkpointing on a multicomputer. An earlier version of our work [KK94] was completed around the same time as their work. Their work is restricted to the Intel i860 platform; in contrast our checkpointing facility is completely portable. They have demonstrated that consistent or coordinated checkpointing is a viable strategy for multicomputers because I/O overhead dominates the coordination overhead. They have used two optimizations – memory based checkpointing and on-line compression – to increase space- and time-efficiency. Both these techniques are orthogonal to our techniques : all of them can be profitably combined to increase efficiency.

There has been a lot of work on consistent or coordinated checkpointing algorithms, starting from Chandy and Lamport's algorithm [CL85]. Our coordination scheme is similar to theirs in requiring $O(N^2)$ coordination messages in the worst case. However it can handle message order reversal.

The concept of using object-specific functions to save and restore objects has been investigated to some extent in the context of I/O for objects and persistent objects. The C++ iostream library allows the programmer to define the "<<" operator for saving and the ">>" operator for restoring the object. Pack functions have also been used to transfer pointer-linked structures across processors in a private address space architecture. The Concert/C[AGKR94] system marshals pointer-linked structures by following pointers before sending them as remote procedure call parameters. However, there seem to be few implementations which automatically generate the pack- and unpack-functions. Our compiler automatically generates these functions for all objects. Most implementations also need complicated mechanisms to take care of pointer-linked structures : aliasing and circular linked structures are two difficult problems; the latter can cause an infinite loop of recursive save-function calls. Our strategy uses the idea of typed memory blocks to avoid these problems. Finally, there has been no previous work on the use of pack and unpack functions in the context of program state reduction for checkpointing.

Performance studies of consistent checkpointing on a distributed system have shown the overhead to be low. In [EJZ92] two techniques that decrease checkpointing overhead are explored. Incremental checkpointing involves checkpointing only modified pages in memory. In the copy-on-write technique, checkpointing I/O continues concurrently with the application, but when a page is written to, it is copied to a separate area in memory from where it can be asynchronously written to disk. While these techniques have been shown to decrease checkpoint file size and overhead, they require operating system support and are thus difficult to implement in a machine independent manner. Moreover, they operate at the level of operating system pages. In our scheme, we checkpoint typed objects in the application, which allows more efficient techniques to be used. Incremental checkpointing ideas can also be incorporated at an object level, thus combining the benefits of both.

In [LNP91] some issues in checkpointing multicomputer applications are surveyed. An efficient coordination algorithm is proposed for finding a consistent global state in which all messages are distinguished as being pre-checkpoint or post-checkpoint. For this the algorithm uses either tags in messages or checkpoint demarcation messages. The quiescence algorithm we use in Section 3.3 has comparable efficiency and is topology independent.

There is no previous work in issues related to language-based checkpointing that we know of. Also, little attention has been devoted to issues in using application information to make checkpointing efficient and easy to use. In [LF90] an adaptive checkpointing scheme is used where the position of the checkpoint can be set by the compiler depending on the size of the program state, in an attempt to decrease the size of the checkpoint file. However, their approach does not make use of application knowledge, so it saves the whole of the program state instead of only selected objects, and moreover, requires a "training run" so that the compiler can estimate the size of the program state at different points.

7 Conclusions and future work

In this work we have demonstrated that for massively parallel programs, language-based checkpointing is competitive with OS/hardware provided checkpointing in terms of ease-of-use, while providing significant advantages in portability and space and time efficiency. The specific contributions made in this work are :

- We have identified new issues that need to be considered for checkpointing massively parallel programs. These lead to different opportunities for optimization than do traditional considerations for checkpointing in distributed systems.
- We have described a checkpointing facility which is provided at the language and run-time system level. Thus it does not depend on operating system or hardware support, and is completely portable to different massively parallel computers. We believe our implementation is one of the first portable implementations of a checkpointing facility.
- We have designed a strategy which minimizes the stack state of the parallel program, reducing the size of the checkpoint file.

- We have presented the mechanism of pack- and unpack functions to efficiently checkpoint objects in the application, while at the same time giving the programmer the mechanism to use application specific knowledge to increase efficiency. We have also developed techniques to automatically generate pack functions for objects.

There are many interesting issues that have been raised by this work. We plan to explore issues in checkpointing a parallel program on one machine and restarting it on another. We also plan to improve time efficiency by using asynchronous parallel I/O.

References

[AGKR94] Joshua S. Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/c: Supporting distributed programming with language extensions and a portable multiprotocol runtime. In *Proceedings of the International Conference on Distributed Computer Systems*, 1994.

[BP93] N. Bowen and D. Pradhan. Processor- and memory-based checkpoint and rollback recovery. *Computer*, February 1993.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots : Determining global states of distributed systems. *ACM Transactions on Computer Systems*, February 1985.

[EJZ92] E. Elzonahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, October 1992.

[FRS+91] W. Fenton, B. Ramkumar, V.A. Saletore, A.B. Sinha, and L.V. Kale. Supporting machine independent programming on diverse parallel architectures. In *Proceedings of the International Conference on Parallel Processing*, August 1991.

[Joh93] D. B. Johnson. Efficient transparent optimistic rollback recovery for distributed application programs. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, October 1993.

[Kal90] L.V. Kale. The Chare Kernel parallel programming language and system. In *Proceedings of the International Conference on Parallel Processing*, August 1990.

[KK93] L.V. Kale and Sanjeev Krishnan. Charm++ : A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programmi ng Systems, Languages and Applications*, September 1993. (Also: Technical Report UIUCDCS-R-93-1796, March 1993, University of Illinois, Urbana, IL.

[KK94] Sanjeev Krishnan and L. V. Kale. Efficient, machine-independent checkpoint and restart for parallel programs. Technical Report 94-2, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1994.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, January 1987.

[LF90] C. Jim Li and W. K. Fuchs. Catch: Compiler assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault Tolerant Computing*, June 1990.

[LNP91] K. Li, J. Naughton, and J. Plank. Checkpointing multicomputer applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems*, October 1991.

[PL94] J. Plank and K. Li. Performance results of ickp – a consistent checkpointer on the iPSC/860. In *Proceedings of the Scalable High Performance Computing Conference*, June 1994.

[SKR93] Amitabh B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.