© Copyright by Sameer Paranjpye, 2000

A CHECKPOINT AND RESTART MECHANISM FOR PARALLEL PROGRAMMING SYSTEMS

BY

SAMEER PARANJPYE

B.Engr., University of Roorkee, Roorkee, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2000

Urbana, Illinois

To my Parents.

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor, Professor L. V. Kale. His generous support, guidance, encouragement and understanding enabled me to carry out this work successfully.

A special thanks to all my friends who have been an invaluable source of encouragement and enthusiasm. I would also like to thank the members of PPL for the many stimulating discussions and maintaining a pleasant working environment. A special thanks to Orion Sky Lawlor for his work on the pack/unpack library and to Milind Bhandarkar for putting up with endless questions, raising several important issues and reviewing this thesis.

Table of Contents

Chapter 1 Introduction **1**
1.1 Motivation 2
1.2 Thesis Objectives 3
1.3 Thesis Organization 3

Chapter 2 Checkpointing Programs **4**
2.1 Program State 4
2.2 Related Work 5

Chapter 3 Converse and Message Driven Execution **7**
3.1 Converse Modules 8
3.2 Messages and Queues 12
3.3 Utility Routines for Checkpointing 13

Chapter 4 Charm++ and Parallel Objects **16**
4.1 Issues in Checkpointing Charm++ 18
4.1.1 Object IDs 18
4.1.2 Dynamic Group Creation 20
4.1.3 Virtual Object IDs 21
4.2 The Object Manager 21
4.3 The Pack/Unpack Library 22

Chapter 5 Shrinking and Expanding **29**
5.1 Changing the Platform Size: Converse 30
5.1.1 Decreasing the Number of Processors 31
5.1.2 Increasing the Number of Processors 31
5.1.3 General Structure of Module Restart Routines 32
5.2 Charm++ and Virtual Processors 33
5.2.1 Difficulties in Charm++ Recovery 33
5.2.2 Virtual Processors 34

Chapter 6 The Checkpointing Process **36**
6.1 Checkpoint Initiation 36
6.2 Steps in the Checkpointing Process 37
6.3 Data Organization 38

Chapter 7 Conclusions **42**
7.1 Future Work 42

Appendix A Converse Interface **44**
A.1 Checkpoint Invocation 44
A.2 File and Directory Management 45
A.3 Utility Routines for Saving and Restoring Data 47

Appendix B Converse PseudoGlobals **50**

References **53**

List of Figures

3.1 Save and Restore functions for module *foo* 9

4.1 CkChareID definitions 19
4.2 A simple class declaration showing the `pup` method 25
4.3 Packing and unpacking a *foo* object 26
4.4 Declaration of the *bar* class showing the `pup` method. 27
4.5 Sample code for checkpointing and recovering a *bar* object 28

5.1 General form of restore routine for module *foo* 32

6.1 Time-line for processor co-ordination steps 39
6.2 Organization of Converse state on disk 41
6.3 Organization of Charm++ state on disk 41

B.1 An example code for global variable usage 52

Chapter 1

Introduction

Parallel computing has enjoyed considerable growth in recent years and parallel architectures are assuming an increasingly central role in information processing [8]. The stimulus behind this growth has been the ever increasing demand for compute power to solve complex problems in the shortest possible time. This sustained growth has also manifested itself in the evolution of a diverse design space for parallel machines. In order to exploit the benefits offered by the multitude of architectures available it is important for parallel applications to be developed in a machine independent fashion making only the most generally applicable assumptions about the underlying architecture. This requirement has been addressed by the developement of the *Converse* parallel runtime system and the *Charm++* parallel programming language. In addition, the scale of parallel applications has been increasing, it is not uncommon for parallel programs to run on a thousand processors and for long periods of time. Applications such as these demand a high degree of reliability from the machine and the runtime system. Parallel programming systems augmented with checkpoint and restart facilities furnish a mechanism for ensuring reliable execution of programs despite system failures. The checkpointing and restart facility described in this thesis is intended to be an extension to Converse and to the Charm++ runtime system. The checkpointing framework at the Converse level also permits checkpointing modules for other programming languages and paradigms (that have been ported to Converse) to be easily integrated into the runtime system.

1.1 Motivation

Checkpoint and restart techniques allow programs to make progress in the face of recurring system downtime. The program state is periodically recorded on stable storage during execution. In the event of a system failure, the computation may be restarted and continued from the 'frozen' state on disk. Several factors motivate the need for a general-purpose checkpoint and restart mechanism for (massively) parallel computing systems. Parallel computers have traditionally been employed to satisfy the needs of science and engineering applications at the edges of computational feasibility [16]. Such applications typically have long running times, ranging from several hours to days at a time, and thus require the systems they run on to provide long periods of continuous failure-free time [20]. In addition, these applications tie up the systems resources preventing other less demanding applications from running, thereby decreasing system throughput and increasing average turnaround time. The increase in the number of system components causes the mean time between failures of the system to decrease. Parallel programs must also successfully contend with this increase in the number of points of failure. There are exacting demands on the capability of parallel computing systems to execute long running, compute intensive tasks to completion. Checkpoint and restart techniques endow parallel programs with a higher degree of fault tolerance than otherwise possible thus enabling complete runs despite non-continuous failure-free time. These facilities provide a number of other benefits in the context of parallel computing systems. Large compute intensive tasks may be checkpointed allowing other programs to run, thereby increasing the average turnaround time of the system. Programs may execute on different machines during a single run, and also 'shrink' and 'expand' being checkpointed after running for a while on a system with a large number of processors, then continuing on smaller, more easily available systems, and vice versa.

1.2 Thesis Objectives

In this thesis we describe the design and implementation of a checkpoint and restart facility targeted at Charm++ – an object-oriented parallel programming language based on C++ and Converse–a parallel runtime system that enables parallel implementations of a variety of programming languages and paradigms and facilitates interoperability among various paradigms. The main objectives of the thesis were:

- To implement a checkpoint and restart facility for Converse, which allows checkpointing modules for different parallel languages to supply callbacks that will be invoked at checkpoint time.

- To implement a checkpoint and restart facility for the Charm++ programming language.

The checkpoint and restart functionality implemented is fully integrated into Converse and Charm++ and is machine independent in nature.

1.3 Thesis Organization

The thesis consists of 7 chapters. Chapter 2 discusses checkpoint and restart techniques in general, the notion of program state we use and presents an overview of related work in the field. Chapters 3 and 4 describe the issues encountered in developing checkpoint and restart mechanisms for the Converse runtime system and the Charm++ parallel programming language respectively. They include brief discussions of the major design issues. In addition to the overall architecture, they also describe the alternate designs considered and the rationale behind choosing one over the other. Chapter 5 deals with the problem of changing the number of processors that a program runs on after a checkpoint. Chapter 6 gives a step by step description of the algorithm used to checkpoint Converse/Charm++ programs. Conclusions and future work are discussed in chapter 7.

Chapter 2

Checkpointing Programs

Checkpointing a running program involves saving enough of the program state on stable storage so that a *recovery point* [20] is established. It must be possible for the program to continue from the recorded state and run correctly to completion. A simplistic approach to checkpointing might save the entire memory image of the program and reload the same upon restart. A better approach might save only the global and static variables and the contents of dynamically allocated memory. Care must be taken to identify exactly what parts of the program are critical to enabling restart. Extensive research has gone into developing checkpointing and restart mechanisms that are space and time efficient and as transparent to the programmer as possible. The next section describes our view of what constitutes a recovery point. This is followed by a discussion of related work in the field.

2.1 Program State

The Converse runtime system is organized in the form of *modules*. The state of a Converse program consists of the global and static variables defined and controlled by each module. These include *message handlers* [18], runtime system statistics and several message queues, among other things. In order to checkpoint a Converse program we must save to disk the state of each module. It is possible that when a checkpoint is initiated, a large number of messages are in transit. We must ensure that all messages are accounted for (i.e. have been

received at their intended destinations, if not processed) before we initiate a checkpoint. This requires a distributed *quiescence detection* algorithm [24], [9].

The Charm++ runtime system is in effect a complex Converse module. However, this complexity and the number of new abstractions it introduces and that it is not an integral part of Converse warrants that it be treated differently from other Converse modules and should define its own checkpointing behavior. A Charm++ program at runtime consists of a number of concurrent objects [17] and its state is completely defined by the states of each of these objects and a few global readonly and pseudoglobal variables. Checkpointing a Charm++ program involves saving the state of each object on disk. The structure that the Charm++ runtime state has due to its being object based makes it easier to checkpoint than Converse.

2.2 Related Work

There is a substantial body of research dealing with checkpoint and restart issues for sequential as well as parallel programs. Checkpointing and restart for sequential programs under UNIX is described in [20]. Compiler assisted checkpointing is examined in [6]. Hardware support for checkpointing and fault tolerance such as Sheaved Memory [25] and Virtual Checkpoint Architecture [2] has also been studied. Most recent work is in the area of checkpointing distributed programs, for example [5, 11, 19, 22]. Message passing systems complicate rollback recovery due to the possibility of *rollback propagation* [10]. If a process that sends a message rolls back to a state before it sent the message, the receiving process must also rollback to a state prior to the receipt of the message. A sequence of rollback propagations of this kind may push the program back to its initial phases thereby losing most of the useful work done before the failure. Such cascading rollback propagation is called the domino effect [21]. Existing approaches to checkpointing for message passing systems are classified into coordinated, uncoordinated and communication induced checkpointing [12].

In a message passing system, each process may perform checkpoints independently of the other processes, this approach is called uncoordinated or independent checkpointing. The advantage of this autonomy is lower runtime overhead and the reduction in state information stored because each process can be made to checkpoint at times when the state is small. However, such an approach is susceptible to the domino effect, and requires maintainance of multiple checkpoints by each process and periodic garbage collection to remove aging and unuseable checkpoint information. Coordinated checkpointing involves a global synchronization among the participating processes so that a system wide consistent state is established. Coordinated checkpointing simplifies recovery and garbage collection and reduces the stable storage overhead of checkpointing. The simplest approaches to coordinated checkpointing block all communication between processes until the checkpoint is complete [7, 26]. Nonblocking schemes for coordinated checkpointing put messages recepients in charge of maintaining checkpoint consistency. A *distributed snapshot* algorithm for a nonblocking checkpoint protocol is described in [5].This allows the existence of non-FIFO channels between the communicating processes. In *communication induced checkpointing* techniques each process performs checkpoints upon the receipt of special messages from other processes, this approach does not require global synchronization and is thought to scale better than coordinated checkpointing. These methods are further classified into *model-based checkpointing* and *index-based coordination.* Several domino effect free models for checkpointing and communication have been proposed. Models in which all message receive events precede all message sending events in every checkpoint interval have been shown to be domino effect free in [23]. A model of this kind, called an MRS model, can be maintained by taking an additional checkpoint before every message receive that is not separated from its previous message send by a checkpoint [1, 28].

Chapter 3

Converse and Message Driven Execution

This chapter describes Converse–a parallel runtime framework. Converse abstracts the underlying architecture from the programmer and supplies him with a uniform and consistent interface on a wide variety of platforms, and the requirements that the design of Converse places on the checkpointing facility. Converse has been ported to most commercial parallel machines and networks of workstations [27].

Converse is an interoperable framework for combining multiple languages and their runtime libraries into a single parallel program. Its software architecture allows developers to integrate multiple separately compiled modules possibly written in different languages without adversely impacting performance. The framework has been verified to support message-passing systems, thread based languages and message driven parallel object oriented languages [14].

The Converse machine model treats the parallel machine as a collection of nodes where each node consists of a number of processors (≥ 1) that share memory. Each processor may have multiple threads running on it. These threads share the same address space but have different stacks. Computational entities (threads, objects etc.) on a processor communicate with entities on other processors by means of messages [18].

The design of Converse is based on the necessity to efficiently handle the different con-

currency models presented by single-threaded modules, message-driven objects and thread based modules. As a result the design of Converse is component based. A parallel language implemented on top of Converse should use only the components that it needs. Each Converse module has state associated with it in the form of variables that it defines and controls, in addition there are several *pseudoglobal* variables (see Appendix B) that form part of the runtime state. To successfully checkpoint and restart a Converse program we must save to disk and restore the state of each module and the states of the pseudoglobal varibles. The next section is devoted to descriptions of the Converse modules, the state they encapsulate and the steps taken to checkpoint and recover that state. This is followed by a discusson of message queues and issues in checkpointing them. The chapter closes with a brief discussion of a library of utility routines that we provide for checkpointing. For a more detailed description see Appendix A.

3.1 Converse Modules

Each Converse module encapsulates a piece of the runtime-system functionality. This functionality is embodied in the form of pseudoglobal variables defined by the module and functions for manipulating them. Each module also defines an initialization routine that is called upon system startup. This initialization routine is usually called *ModuleName*`Init`, and is typically called from the main system initialization routine `ConverseCommonInit`. To provide checkpoint and restart functionality, each module has been augmented by two additional functions, called *ModuleName*`Save` and *ModuleName*`Restore`. These routines embody the modules checkpoint and restart functionality respectively. To be checkpointable, a module must provide these two functions. The *Save* functions provided by the modules are callbacks that are invoked at checkpoint time. For modules essential to the functioning of Converse, calls to these functions are explicitly encoded in a function `CcpSaveModules()`, that is called at each checkpoint. Other modules can register their *Save* functions with the checkpointing

facility by calling `CmiRegisterCheckpointFn(CcpFn)` during initialization, with a pointer to the *Save* funtion as the parameter.

The initialization routines do not need to change except for the addition of a call to the *Restore* function at restart time. The `_IS_RESTART()` macro indicates whether the program is starting for the first time or recovering from a checkpoint and is used to determine whether the *Restore* routine is to be called. File management is completely handled by the checkpointing facility, modules do not need to be aware of where their data is stored in the checkpointed state or how the checkpoint data is organized. A function that saves a modules state only needs to call `CcpBeginSave`, when it begins and `CcpEndSave`, when it ends. Similiarly, the restore function needs to call `CcpBeginRestore`, when it begins and `CcpEndRestore` when it ends. This is best illustrated with an example. Figure 3.1 shows the general structure of save and restore routines written for module *foo*.

```
void fooSave(void)
{
  CcpBeginSave(``foo'');
  //Save module data
  .
  .
  .
  CcpEndSave();
}
```

```
void fooRestore(void)
{
  CcpBeginRestore(``foo'');
  //Restore module data
  .
  .
  .
  CcpEndRestore();
}
```

Figure 3.1: Save and Restore functions for module *foo*

A brief description of Converse modules and their checkpointing behavior follows:

- **Converse Statistics** - This module keeps track of runtime system statistics such as memory usage, message send and receive numbers etc. The statistics module is not critical to the operation of Converse. Its checkpointing behavior consists of saving a number of counters to disk and restoring the values of these counters when the program restarts.

- **Converse Conditional Callbacks** - Converse includes a mechanism that allows the programmer to insert hooks that are invoked when the system reaches a specific state or according to certain timing constraints. A conditional callback of this kind consists of a function pointer and a generic pointer to the argument. Since pointers to functions are not in general checkpointable (if the program is recompiled after a checkpoint, the functions may not be loaded at the same virtual addresses), we do not attempt to save their values to disk. Instead, we save only the arguments to the callbacks. Any module that registers a conditional callback and requires the callback to be restored at restart time must supply along with the function pointer and pointer to argument information that enables the arguments to be checkpointed. This consists of two handles - the module name and the function name, and pointers to pack and unpack functions for the arguments to the callbacks. At checkpoint time the arguments are packed and saved. When the program restarts it is the responsibility of the modules that registered the callbacks to supply the function pointers together with their handles so that the callbacks are restored correctly.

- **Converse Handlers** - This is the module that controls the assignment of handler numbers. A handler must have the same number on every processor. Also, in the interests of flexibility, it is desirable to make it possible to add and remove modules from a program once it checkpoints. To make this possible. Handler numbers are treated specially. Every Converse module registers certain handlers with the `CmiHandler` module. Before registering its handlers each module makes a call to `CmiRegisterModule` and registers itself with the `CmiHandler` module. Checkpointing for the `CmiHandler` module is done only on processor 0. Each module name is stored along with the starting handler number for that module. Upon restart, handler number registration doesn't need to change at all. Each module makes the same registration calls as in normal startup. When a module X registers itself at restart the `CmiHandler` code checks if

module X was part of the program prior to the last checkpoint. If so, it looks up the starting handler number for module X prior to the checkpoint and assigns handler numbers to X starting from this value.

- **The Converse Scheduler** - The Converse scheduler (`Csd`) module manages the scheduler queue. Messages in the scheduler queue may need to be packed at checkpoint time, but only the modules that added these messages can perform this packing correctly. The job of the `Csd` module at checkpoint time is to deque all the messages in the schedulers queue and string them onto another FIFO queue. All modules that add messages to the scheduler queue examine this FIFO and checkpoint the messages that they inserted into it. At restart time the `Csd` module does nothing. The modules that 'owned' messages in the scheduler queue prior to checkpoint re-insert them for processing.

- **Converse Client Server** - The state of this module consists of a linked list of handler numbers and names. The checkpointing behavior consists of saving and restoring this linked list.

- **Converse Random Numbers** - The `Crn` module manages a stream of random numbers. At checkpoint time it saves the current state of the stream and restores it at restart time.

- **The Seed Balancer** - The function of the seed balancer is to load balance the creation of message driven objects. When a message requesting the creation of an object is sent out, it first goes to the seed balancer. The seed balancer then determines, according to some strategy, where the object should be created. The strategy may be, for instance that the object is created upon the currently least loaded processor. This is one of the modules that inserts messages into the schedulers queue. The checkpointing behavior of the seed balancer thus consists of retrieving its messages (a module can identify

messages that it inserts into the scheduler queue by examining the message header) from the scheduler queue (actually the FIFO queue that the `Csd` module creates), packing them if necessary and then saving them to disk. At restart these messages are recovered and re-inserted into the scheduler queue.

3.2 Messages and Queues

A Converse message is essentially a sequence of bytes. The first few bytes of the message contain a 'handler number' that specifies a function ('handler') that processes the message. Converse maintains a mapping of handler numbers to function pointers in a table. There are two kinds of messages in Converse, messages that come over the network and locally generated messages, and these messages reside in different queues. A scheduler runs on every processor, all messages handlers in the system are registered with the scheduler, when a message arrives on any queue; the scheduler examines the message and invokes the appropriate handler. There are several message queues that form part of the Converse runtime, most of them are processor level queues, there are some node level queues as well, and any processor on a node may handle a message from the node queues.

The various queues and their functions are described below:

- **CmiLocalQueue:** All local messages, i.e. messages generated on a processor that are sent to itself reside in this queue.

- **CmiNonLocalQueue:** This is an abstraction provided by the Converse machine interface. Converse provides a `CmiGetNonLocal`, function that is the interface to this queue and removes a message from it. All non-local processor messages reside in this queue when they first arrive over the network.

- **CmiNonLocalNodeQueue:** This is similar to the CmiNonLocalQueue, but is meant to store non-local messages meant for a node (i.e. messages that may be handled by

any processor on a node), when they arrive over the network.

- **CsdSchedQueue, CsdNodeQueue**: These are two additional 'scheduler' queues that can store prioritized messages and messages that require packing and unpacking, non-local messages of this type appear in the non-local queues and are then moved to these queues. The CsdSchedQueue is a processor level queue and the CsdNodeQueue is a node level queue.

The treatment of these queues at checkpoint and restart time is fairly simple. When a checkpoint is initiated, the checkpointing algorithm starts with saving the state of all the Converse modules. It then waits for all messages in transit to arrive at their destinations. Every message once it arrives at its destination processor resides in one of the above mentioned queues. To checkpoint these queues, we simply pluck all the messages out of each queue andsave them to disk. The scheduler queues are treated a little differently. Since, messages in these queues may require packing and unpacking, checkpointing of these messages is delegated to the modules that inserted them in the queue. Each module inspects the scheduler queue for messages that it inserted and saves them to disk, packing them if necessary.

At restart time, local queue messages are simply picked up and re-inserted into the local queue. However, it is not possible to insert messages into the non-local queues, non-local messages are therefore also inserted into the local queue. Scheduler messages are recovered by their controlling modules and may be inserted into the scheduler queues or into the local queue.

3.3 Utility Routines for Checkpointing

As stated earlier, to aid the checkpointing of Converse modules, all the file management is handled by the checkpointing subsystem. No module ever needs to know what file(s) it is

writing data to, what format the data is stored in, the way its state information is stored relative to the data of any other module etc. The only thing that is required of a modules checkpointing and recovery code is that data items be retrieved in the same sequence that they were stored on disk.

If a module absolutely must have some information about the file that its data is stored in, it can retrieve the file pointer (`FILE *`) for its data file at any time during checkpointing or recovery, by calling the `CcpGetFp` function. Further, the checkpointing subsystem provides a set of primitives for storing and retrieving data. These primitives deal with all basic data types and a few more complex data structures. The utility routines provided are:

- `CmiSaveChar, CmiReadChar` - These functions store and read a single character from the checkpoint file being processed. A module can use this if it needs to store single characters that are part of its state. This, however is an unlikely situation. The major uses of these functions are in defining formatting constructs used in more complex functions such as `CmiSaveArray, CmiReadArray` etc.

- `CmiSaveInt, CmiReadInt` - These functions store and read a single `int` value from the checkpoint file being processed. Functions similar to this are available for other built-in numeric datatypes in C. These functions have names of the type `CmiSave`*dataType*, where *dataType*, is a numeric datatype such as `float` or `double`.

- `CmiSaveArray, CmiReadArray` - This is perhaps the most versatile pair of functions available. They are intended to save and recover arrays of arbitrary types. Parameters to these functions allow the programmer to chose from a rich set of options. For instance one may specify an array of statically or dynamically allocated objects to be saved. It is also possible to indicate that an array is **sparse**, (i.e. not all elements need to be saved and restored) so that only elements that satisfy certain programmer specified criteria for 'non-sparseness' are dealt with.

- `CmiSaveString, CmiReadString` - These functions deal with saving and restoring C-style null-terminated strings of characters. Since strings are just character arrays, their implementation consists simply of calls to the analogous functions for Arrays with appropriate parameters.

- `CmiSaveList, CmiReadList` - This pair of functions deals with saving and recovering singly linked lists of objects. Singly linked lists occur frequently enough in code to merit the inclusion of these functions among the utility routines

- `CmiSaveBytes, CmiReadBytes` - Functions in this pair deal with raw sequences of bytes and assume no semantics for the data that is passed to them. They simply save and recover sequences of bytes to and from stable storage.

Chapter 4

Charm++ and Parallel Objects

Charm++ is an object oriented parallel programming system based on C++. It is explicitly parallel in nature and provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message driven; a Charm++ program in execution is a collection of concurrent objects of various types (described in more detail later) that communicate by sending messages to one another [15]. Charm++ programs can freely use most object-oriented and generic programming features of C++, including multiple inheritance, polymorphism, overloading, strong typing and templates.

Every Charm++ object is a regular C++ object, i.e. an instance of a C++ class. Concurrent and replicated objects in Charm++ have several special attributes that are provided and supported by the run-time system.

A Charm++ object belongs in one of the following categories:

- Chares (Concurrent objects) - Chares are the most important entities in a Charm++ program. Unlike ordinary C++ objects, Chares can be created asynchronously on remote processors and special methods, called *entry methods* on these objects may be invoked asynchronously from remote processors. An entry method is invoked by sending a message to the Chare.

- Chare Groups - Chare groups are special concurrent objects. Each chare group is a collection of chares with one branch on every processor. All members of a chare group

share a globally unique identifier and messages may be broadcast to the whole group or to a specific branch.

- Chare Nodegroups - Chare Nodegroups are similar to groups and except that instead of having one branch on every processor, nodegroups have one branch on every SMP node that the program runs on.

- Chare Arrays - Chare Arrays are generalized collections of Chares, however these collections are not constrained by the underlying architecture, chare arrays can have any number of elements and this number may change at runtime.

- Sequential objects - These are regular C++ objects, except that they may not have static data members. These objects are local to a processor and the Charm++ runtime has no knowledge of their existence, they are typically members of other concurrent objects.

- Messages (Communication objects) - These are entities that constitute the arguments to asynchronously invoked methods of concurrent and replicated objects. A Charm++ message consists of an *envelope* followed by the message body. The envelope is used by the Charm++ runtime and stores message attributes such as the message type, source processor etc. Entry methods that handle the messages deal only with the message body.

- Readonly objects - A readonly object represents a global variable in the computation. Charm++ does not allow mutable global variables in the computation in order to keep programs portable across a wide range of platforms. Readonly objects are a way to provide a way to share data amongst all the objects involved in a computation. In cases where the size of the readonly object is not known at compile time, readonly *messages* may be used. Readonly messages are declared as pointers. Pointers to messages are the only readonly pointers allowed.

Checkpointing a Charm++ program involves saving the state of each object involved in the computation, together with any unhandled messages and readonly objects. The design of the Charm++ runtime does not lend itself to checkpointing easily, this is largely owing to the presence of a large number of pointers that form part of the runtime state. The effort to develop an effective checkpointing strategy for Charm++ programs was therefore first directed at developing techniques to eliminate these pointers from the runtime system. The main role of these pointers is as identifiers for objects that exist during the execution of the program. To make it possible to checkpoint these identifiers so that they correctly distinguish objects when the program restarts, all pointers were replaced by indices into tables. The tables store the actual pointers to the objects, upon restart the object pointers are restored at the same indices in the table that they occupied prior to the checkpoint. The operation of tables that form part of the runtime system is supervised by an entity called the *object manager*, one instance of which exists on every processor. The next section presents some examples of runtime system identifiers containing pointers and explains how these were made persistent across checkpoints. This is followed by a description of the object manager and its operation. The chapter concludes with a discussion of the pack/unpack library used to checkpoint and recover the state of Charm++ objects. Most of the difficulties are encountered in the state of the runtime system proper and not in the state of the user program which typically consists of a number of concurrent objects.

4.1 Issues in Checkpointing Charm++

4.1.1 Object IDs

Concurrent objects (objects possessing methods that can be invoked asynchronously) in a Charm++ program require identifiers that are unique across all processors. *Chares* are identified by handles called `CkChareIDs`. A `CkChareID` is a system defined structure and is essentially a global pointer that uniquely designates the object across all processors. A

`CkChareID` as defined in the Charm++ runtime cannot persist across checkpoints and therefore has to be modified. Figure 4.1 shows the definitions of the persistent and non-persistent `CkChareID` structures.

```
typedef struct {
  int onPE;
  int magic;
  void *objPtr;
} CkChareID;

(a)
```

```
typedef struct {
  int onPE;
  int magic;
  unsigned int objNum;
} CkChareID;

(b)
```

Figure 4.1: CkChareID definitions. (a) **Non-persistent CkChareID** - the object is identified by processor number and a pointer to the object. This handle loses its validity upon restart. (b) **Persistent CkChareID** - the object is now identified by processor number and object number, the object number is an index into a table and is assigned by the *object manager*. This number remains unchanged upon restart.

A `CkChareID` is a structure that contains as members a processor number and a pointer to a dynamically allocated object. Together these two members suffice to identify the object uniquely among all the processes that constitute the Charm++ program. Even if the object is correctly restored when the program restarts there is no guarantee that memory for the object will be allocated at the same heap address, therefore object pointers in `CkChareIds` will be invalid at restart time. A possible solution is to augment the `CkChareID` with an object number that is assigned by an object manager and refresh the object pointer at restart by querying the object manager on the appropriate processor. However, this solution has the drawback that it leads to additional message passing upon restart. In addition, messages to objects cannot be sent using the pointers until we are certain that they have been refreshed. We chose therefore to replace the object pointer by an object manager

assigned object number. The only overhead of this solution is the additional table lookup that occurs every time a message is sent to an object and this is a small constant cost.

4.1.2 Dynamic Group Creation

A *Group* is a replicated object, one branch of which exists on every processor. A Group is identified by an integer called the groupID. The groupID is assigned when the group is created, since multiple group creations may occur concurrently there must exist a mechanism to ensure that different groups are not assigned the same groupID. This is accomplished by making a specific processor (in this case processor 0, since it is guaranteed to exist) responsible for assigning groupIDs. All requests for groupIDs are therefore serialized. Group creation starts with an asynchronous invocation of the group class' constructor by some object. The constructor must be invoked on all processors (i.e. the message argument must be broadcast to all processors). However, before the messages are sent a groupID must be obtained. For this purpose the message argument is stashed away on the calling processor and a *request message* is sent to processor 0. The *request message* holds a pointer to the constructor argument. When processor 0 receives the *request message* it replies with a *number message* that contains the groupID. Processor 0 also copies the pointer to the constructor argument from the *request message* to the *number message*. When the reply is received by the requesting processor it uses the pointer in the *number message* to access the constructor argument and then broadcasts it to all processors along with the groupID. This scheme impedes the ability to checkpoint because if a checkpoint is initiated while a group creation is in progress, then the pointers in the request and number messages will be invalid when the program restarts. We circumvent this difficulty by replacing the pointer in the messages in the group creation protocol by indices into a table of messages. These indices remain valid across checkpoints.

4.1.3 Virtual Object IDs

A virtual ID is similiar to a `CkChareID` but is closer in nature to a proxy for a Chare rather than being a simple handle to a Chare. When a Chare creation message is sent there is no way to find out the `CkChareID` of the chare being created, unless the Chare itself reports its identity to the creating object when it comes into existence. This causes a problem if a programmer wants to send messages to a chare immediately after invoking the constructor without waiting for it to actually come into being. Virtual IDs alleviate this problem, when a chare is created a virtual ID is created for it, messages may be sent to the virtual id immediately after the asynchronous constructor invocation. The virtual id queues up the messages until the real chare comes into existence, it then redirects all the messages it has queued up and all subsequent messages to the real chare. A `CkChareID` that is a virtual id contains instead of an object pointer a pointer to a *proxy*, the *proxy* is responsible for queueing and redirecting messages. Pointers to *proxies* that reside in `CkChareID`s do not persist across checkpoints and have now been replaced by indices into a table of such proxies.

4.2 The Object Manager

The object manager is responsible for the management of all the pointer tables that exist on a processor. It presents an interface that is a facade to three tables that contain pointers to Chares, Virtual IDs and Messages. The object manager supplies methods for adding pointers to the tables, retrieving pointers and querying the tables for attributes of pointers, for instance whether a particular table index is occupied or not, whether the object it points to is still alive or has expired. In addition the object manager is completely responsible for co-ordinating the checkpointing of all chares in the system and for checkpointing all the proxies and pending group creation messages in the system.

4.3 The Pack/Unpack Library

The pack/unpack or "pup" library is a collection of efficient and elegant classes that enable the state of Charm++ objects to be checkpointed and recovered from disk. The **pup** library can be extended to provide services to any operation that requires a traversal of the object state (typically a traversal over the objects data members).

To checkpoint a Charm++ program, we need to save the state of all the concurrent objects in the system to disk. The state of sequential objects is subsumed by the state of the concurrent objects since, in a Charm++ program, every sequential object is a member of or is pointed to by a member of a concurrent object. Chares and replicated objects may contain dynamically allocated data the size of which varies at runtime. All of this data also needs to be saved to disk and faithfully recovered at restart. Also concurrent objects such as chare array elements need to migrate at runtime. To accomplish this, the state of the object must be 'packed' into a memory buffer; the memory occupied by the object released on the processor where the object resides; the object state transported to the new processor where the object is to be migrated and the object recreated at the new location. Migration of array elements for load balancing was supported by the Charm++ runtime system before the checkpointing project was undertaken [13, 3, 4]. We chose to view checkpointing as a variant of migration. When a checkpoint is made, the Charm++ objects are seen as 'migrating' to disk, and upon restart they return to their respective processors. To migrate an object its data needs to be 'packed', i.e. serialized either into a memory buffer or to disk and then 'unpacked' into memory.

Migration and checkpointing for objects can be handled in several ways. A possible approach is to require each class to implement `static` **pack** and **unpack** methods. If an object is required to migrate to another processor while the program is executing, the *class method* **pack** is invoked with a pointer to the object as argument. The pack method allocates a memory buffer large enough to hold all the objects data and then proceeds to serialize the

objects data into the memory buffer. The memory buffer is then inserted into a message and sent to the processor where the object is to be migrated. When this message containing the object state is received by the new processor, a new instance of the class is created by calling a special *migration constructor*. The migration constructors task is to simply create an uninitialized instance of the class. The **unpack** class method is invoked with pointers to the migration message and the newly created raw object. The unpack method proceeds to stuff the new object with data from the old one. At the end of the unpack method, migration is complete.

With class methods available for packing to memory and unpacking from it, it is easy to implement methods that serialize objects to disk and recover object state from disk. A **serialize** method could be as simple as a wrapper around the pack method, it just grabs the contiguous memory buffer produced by the pack method and writes it to disk. The problem with this approach is that for objects whose members reference large quantities of dynamically allocated memory a contiguous buffer needs to be produced for writing the object state to disk, which is wasteful. Secondly, this approach creates difficulties in the way of implementing platform independent checkpointing for objects. Another approach to writing a serialize method does not use the pack method. Rather the serialize method is completely self-contained and writes the object state to disk. The problem with this approach is code duplication, both the pack and the serialize method must perform a traversal of the objects members and perform different operations on the members.

The above discussion suggests that the **pack** and the **serialize** methods have a common 'skeleton', with only the actual operation the methods perform on the data members being different. The **pup** library has been designed with this intent. The programmer of a particular class only needs to implement a single method, called `pup`. The `pup` method takes a single parameter, which is an instance of a *packer/unpacker* or *pupper*. The nature of puppers shall be dealt with subsequently. The role of this method is to perform a traversal of the object state. The actual operations that need to be performed on the data members

are executed by the pupper.

The **pup** library contains the following important classes:

- `class PUP::er` - This class is the abstract superclass of all the other classes in the system. The `pup` method of a particular class takes a reference to a `PUP::er` as parameter. This class has methods for dealing with all the basic C++ data types. All these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.

- `class PUP::packer` - The abstract superclass of all classes that 'pack' objects. It is not clear here what packing means, but it may be considered as any operation that performs a non-destructive transformation on the objects state, i.e. the 'packing' operates on the data that constitutes the object state and creates a different representation of that state. The object does not change as a result of this operation. Introduces additional methods `isPacking` and `isUnpacking` that may be used to query the class to determine its mode of operation

- `class PUP::unpacker` - The abstract superclass of all classes that 'unpack' objects. Unpacking is the opposite of packing as described in the previous item. An unpacking operation works with a 'raw' (uninitialized) object and a some representation of the object state. The process of unpacking involves a traversal of the object state, at each step of the traversal, part of the object state is 'converted' from the given representation into a piece of memory holding the right bit pattern. When the unapcking is complete, the entire object state has been recovered.

- `class PUP::sizer` - This is a subclass of the `PUP::er` class. It's function is to determine the size (in bytes), of the object that it operates on.

- `class PUP::toMem` - This is a subclass of the `PUP::packer` class. The role of this class is to pack the object it operates on into a preallocated contiguous memory buffer. The

most general way to pack an object into a memory buffer is to invoke `pup` on the object with an instance of `PUP::sizer` to determine the size of the object. Then a buffer of the required size is allocated and `pup` is invoked again with an instance of `PUP::toMem` that has been initialized with the allocated buffer.

- `class PUP::fromMem` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given contiguous memory buffer.

- `class PUP::toDisk` - This is a subclass of the `PUP::packer` class. The role of this class is to save the state of the object it operates on into a disk file. To serialize an object to disk `pup` in invoked on the object with an instance of `PUP::toDisk` that has been initialized with a file pointer.

- `class PUP::fromDisk` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given disk file.

Figure 4.2 shows the shows a class declaration that includes a `pup` method:

```
class foo {
 private:
  bool isBar;
  int x;
  char y;
  unsigned long z;
  float q[3];
 public:
  void pup(PUP::er &p) {
    p(isBar);
    p(x);p(y);p(z);
    p(q,3);
  }
};
```

Figure 4.2: A simple class declaration showing the `pup` method

The routine in Figure 4.3 presents an example of how an instance of the *foo* class may be packed and unpacked from a memory buffer.

```
int main()
{
  //Build a foo
  foo f;
  f.isBar=false;
  f.x=102;f.y='y';f.z=1234509999;
  f.q[0]=(float)1.2;f.q[1]=(float)2.3;f.q[2]=(float)3.4;

  //Collapse f into a memory buffer
  //Allocate a buffer for the foo object
  PUP::sizer s;
  f.pup(s);
  void *buf=(void *)malloc( s.size() );
  //Pack f into preallocated buffer
 {PUP::toMem m(buf);f.pup(m);}

  //Unpack the foo object
  foo f2;
  {PUP::fromMem m(buf);f2.pup(m);}
}
```

Figure 4.3: Packing and unpacking a *foo* object

The following more complex example shows how an instance of the *bar* class may be serialized to disk and then recovered from stable storage. The *bar* class has an instance variable of type *foo*. To pack/unpack or checkpoint/recover an object of type bar we must apply the same operation to the instance variable `foo f`. This is accomplished by having the `pup` method of the bar class invoke `pup` on the member of type foo with the *pupper* passed to it. Figure 4.4 shows the declaration of the *bar* class, including the `pup` method. Figure 4.5 shows sample code illustrating how instances of the *bar* class may by checkpointed to disk and how their state may be restored from disk files.

```
class bar {
 public:
  foo f;
  int nArr;//Length of array below
  double *arr;//Heap-allocated array

  bar() {}
  bar(int len) {nArr=len;arr=new double[nArr];}

  void pup(PUP::er &p) {
    f.pup(p);
    p(nArr);
    if (p.isUnpacking())
      arr=new double[nArr];
    p(arr,nArr);
  }
};
```

Figure 4.4: Declaration of the *bar* class showing the `pup` method.

```
int main()
{
  //Build a foo
  foo f;
  f.isBar=false;
  f.x=102;f.y='y';f.z=1234509999;
  f.q[0]=(float)1.2;f.q[1]=(float)2.3;f.q[2]=(float)3.4;

  //Build a bar
  bar b(2);
  b.f=f;//Just copy our old f.
  b.arr[0]=876543.21;
  b.arr[1]=-345.67;

  //Write the bar to disk
  {
    FILE *f=fopen("bar.bin","wb");
    PUP::toDisk d(f);
    b.pup(d);
    fclose(f);
  }
  bar b2;
  {
    FILE *f=fopen("bar.bin","rb");
    PUP::fromDisk d(f);
    b2.pup(d);
    fclose(f);
  }
  return 0;
}
```

Figure 4.5: Sample code for checkpointing and recovering a *bar* object

Chapter 5

Shrinking and Expanding

In the previous chapters we have described in detail the functionality provided by the checkpointing subsystems written for Charm++ and Converse. The discussion so far has been conducted under the implicit assumption that when a Converse/Charm++ program checkpoints, it is restarted on the same number of processors that it was originally running on. The checkpointing subsystem described in this thesis also gives Converse/Charm++ programs to *shrink and expand*. This means, that once a program checkpoints it may be restarted on a smaller or larger number of processors than it was originally running on. This is an important benefit for parallel programs. Large parallel machines are often not available to a program for exclusive use. With the ability to change the size of the set of processors that the program runs on a program can use all of a large parallel machine for as long as it is available and then simply checkpoint and restart using a smaller set of processors. If all parallel programs running on a machine had the ability to shrink and expand after a checkpoint, an operating system scheduler could implement a swapping scheme that took advantage of this ability. The throughput of parallel machines could be substantially increased by implementing such a scheme. Providing parallel programs with the ability to shrink and expand implies that they must somehow be made independent of the number of processors that they run on. This is not in general possible at all levels, the lowest level message passing and routing algorithms and libraries must know the size of the platform that they are dealing with. But it is possible to create the illusion of platform size independence at higher levels,

particularly when dealing with Charm++ objects. A concurrent object in Charm++ should not need to know exactly what processor another object that it communicates with resides on. At the same time some abstractions such as that of object groups are tightly bound to the platform size, and these are necessary in certain situations particularly when dealing with data exchange between modules. Making parallel programs execute on platforms of different sizes raises a whole new set of issues. This chapter describes the strategies employed to deal with these issues. The next section describes the behavior of Converse modules with varying numbers of processors. This is followed by a discussion of the behavior of Charm++ programs in such a scenario and the notion of *virtual processors*.

5.1 Changing the Platform Size: Converse

When a parallel program checkpoints and then restarts on a different number of processors, the main questions that arise are as follows. If the number of processors that the program is running on decreases after restart then the program state that existed on the 'lost' processors must somehow be distributed and integrated into the state existing on the remaining smaller number of processors. If the number of processors increases, then the variables that embody the program state must be meaningfully initialized on the new processors that are added. We have seen that the state of a Converse program is embodied by the variables defined and controlled by the various Converse modules, and by the state of the several message queues in the system. There is a clear separation between the states of different modules. Though the state of a module, may be referenced and even modified by another module, for the purposes of checkpointing and recovery, the state of the runtime is split up among the modules. Every piece of data has exactly one module 'in charge' of it. Though the separation is not formally enforced by means of programming constructs (e.g. modules are not objects or namespaces), it is natural and logical. The next two subsections discuss the strategies employed at restart when the number of processors decreases and increases respectively.

5.1.1 Decreasing the Number of Processors

The strategy used when the size of the platform that a program is running on decreases is fairly simple. Every module 'knows' what part of the runtime state it is in charge of. In general, it is not possible for the checkpointing subsystem to determine how state data for a module on a 'lost' processor is to be integrated with the data on a currently existing processor. Also, care must be taken to ensure that messages that exist at checkpoint time go to the right places when the program restarts. The checkpointing subsystem therefore only determines the general scheme for redistributing state, the details of integrating the data from two instances of the same module are left to the module itself. The overall data redistribution scheme is as follows:

Let x be the number of processors prior to checkpoint and let y, $y \leq x$, be the number of processors at restart. Then at restart processor p, takes charge of the data of all processors numbered, $p, p+y, \ldots, p+ky$, where:

$$k = \begin{cases} \lfloor \frac{y}{x} \rfloor & , \quad p < x \bmod y \\ \lfloor \frac{y}{x} \rfloor - 1 & , \quad p \geq x \bmod y \end{cases}$$

So when processor p is recovering from the checkpoint, each Converse module on p reads in its own data and also reads in the state of its instances that existed on processors $p + y$ etc.

5.1.2 Increasing the Number of Processors

When increasing the number of processors a program runs on, no data redistribution is done. Each processor that existed prior to the checkpoint reads in its own state from disk. All the new processors added must however initialize the runtime state. Usually default initialization suffices for most modules, but in some cases some special actions may need to be taken. Since no redistribution is done, the role of the checkpointing subsystem in this case is minimal.

It leaves the initialization to the modules. All that the checkpointing subsystem provides is the ability for a module to determine whether its processor existed before the checkpoint or not. If it did it can read in its own data, if it did not, it may perform initialization any way it desires.

5.1.3 General Structure of Module Restart Routines

The form of a restart routine for a module now takes the form shown in Figure 5.1, for module *foo*. Note the differences between this restart routine and that shown in Figure 3.1. If `CcpBeginRestore` returns -1, the module infers that its processor did not exist prior to the checkpoint. If its processor did exist it goes through the `do` loop, reading in the state data for all instances of itself that existed on its own processor p and on processors $p + y$ etc. If module *foo* is reading in the state data of its instance on processor $p + ky$, then the function `CcpNextPe` makes available for reading data belonging to *foo* that existed on processor $p + (k + 1)y$ if that processor existed before the checkpoint. In such a case it returns an integer greater than or equal to zero. If the module has read all the data it needs to `CcpNextPe` returns -1.

```
void fooRestore(void)
{
  if (CcpBeginRestore(``foo'') < 0) return;
  do {
  //Restore module data
  .
  .
  .
 } while (CcpNextPe() >= 0);
  CcpEndRestore();
}
```

Figure 5.1: General form of restore routine for module *foo*. This is when shrinking and expanding is permitted

5.2 Charm++ and Virtual Processors

Recovering the state of the Charm++ runtime on a different number of processors raises the same issues as with Converse. These issues are however much more complex in the context of Charm++, so that the straightforward techniques applied to recovering Converse modules run into several difficulties. In order to use the same techniques successfully, several changes to the Charm++ runtime were required. We now proceed to describe some of these problems. This is followed by a discussion of *virtual processors* and how they are used to simplify the recovery of Charm++ programs when the platform size changes.

5.2.1 Difficulties in Charm++ Recovery

Consider, the object manager, as described in Section 4.2, the object manager is an entity one instance of which exists on every processor. The object manager is responsible for holding pointers to all `Chares` that exist at runtime and for co-ordinating the checkpointing and recovery of these objects. A straightforward way to recovering the state of the object manager when the number of processors decreases is to have each existing instance take ownership of the objects belonging to the other processors. This, however leads to difficulties in passing messages to Chares. Chares are identified by `CkChareID`s, which are global pointers. `CkChareID`s include the processor number for the object. If the object manager were to simply take ownership of objects on another processor, then `CkChareID`s cached by objects might no longer be valid. A different message routing scheme would have to be devised for sending messages to Chares whose ids contain processor numbers that no longer exist.

Another problem arises when recovering data belonging to object groups. When recovering the state of an object an uninitialized instance of the class is created and then the `pup` method is invoked. The `pup` reads in the objects data from disk and initializes the object as needed. Recovering object groups on a different number of processors creates additional difficulties. We need to either redistribute the data of all branches that existed before the

checkpoint among the currently existing branches when the number of processors decreases or to meaningfully initialize newly created branches when the platform size increases. Doing this for Converse modules is fairly easy, but objects groups, for instance, the array managers can encapsulate extremely complex functionality and combining the state of these objects can be a difficult task. It is possible to write `combine` methods for classes that represent groups, but the code complexity required can easily lead to hard to diagnose errors. It also puts additional responsibility on the programmer and goes against one of the main goals of the checkpointing subsystem which is to minimize programmer intervention.

5.2.2 Virtual Processors

To circumvent the difficulties mentioned above an additional level of abstraction was added to the Charm++ runtime system by the introduction of *virtual processors*. A virtual processor is simply a wrapper C++ class or *facade* around all the state in the Charm++ runtime that exists on a processor. The major components of which are the object manager which handles the pointers to Chares and messages, the *groupTable* which manages all interaction with object groups and the quiescence detection state. A quiescence detection module exists as part of both Charm++ and Converse because in a situation where modules written in several different languages need to interact it may be desirable to detect quiescence w.r.t Charm++ alone and not w.r.t the system as a whole. Instead of interacting directly with these components of the runtime state, the Charm kernel instead interacts only with the virtual processor which forwards all requests to the appropriate component.

Virtual processors are a simple mechanism to evade the difficulties highlighted in the previous subsection. A physical processor has one or more virtual processors residing on it. A Charm++ program running for the first time has exactly one virtual processor assigned to each physical processor. A Chare, instead of being bound to a physical processor now belongs to a virtual processor. Regardless, of whether the platform size changes or not it ownership of a Chare is retained by the virtual processor that it was created on. Due to this,

`CkChareID`s remain valid even after restarting on a different number of processors.

The problems with writing `combine` methods for groups are also removed. A branch now belongs to a virtual processor and only recovers data that belongs to it. In the event that the number of physical processors increases new virtual processors are created on the added physical processors and new branches for all existing groups are created on these processors. These branches contain no meaningful state data when they are first created and need to be explicitly initialized. No messages are sent to these branches until they have been initialized. Every group class needs to supply an entry method that can be used to initialize empty branches. The semantics of this method are not otherwise constrained in any way. It may take any kind of message as argument and may have any name the programmer chooses. However, to make this scheme work each group class also needs to provide a method called `getBranchInitMsg`, whose signature is `void* getBranchInitMsg(int &eIdx)`. This method should return a pointer to an initialization message that must be sent to all new branches. The parameter `eIdx`, is an output parameter that is used to specify the *entry index* of the method to be invoked on the new branches. Group class writers need not be concerned with what the entry index means. All that is required of them is that `getBranchInitMsg` initialize `eIdx` with the return value from `CProxy_`*ClassName*::`ckIdx`*MethodName*, where *ClassName* is the name of the group class and *MethodName* is the name of the method used to initialize the new branches.

Virtual processors also provide additional benefits for Charm++ programs, the additional degree of freedom that they provide when restarting a program is accessible at runtime also. It now becomes possible to shrink and expand a Charm++ job at runtime.

Chapter 6

The Checkpointing Process

We now present an overview of the entire process of checkpointing a Converse/Charm++ program. Most of the details of the process have already been covered in previous chapters. This chapter is meant to give a step by step outline of the process filling in miscellaneous details that have been omitted in previous portions of the text. The steps involved in a co-ordinated checkpoint are also described in [16]. The chapter includes a section describing the layout of the program state on disk.

6.1 Checkpoint Initiation

The most suitable locations (i.e. points in the execution of a parallel program) for creating checkpoints are application dependent. It is usually suitable to checkpoint a program at the end of a phase in the computation when the program state is the smallest. Also, the frequency with which checkpoints are invoked depends on the frequency of faults and the space and time overheads imposed by checkpointing. Programmers may also want to initiate checkpoints 'on-demand'. The checkpointing subsystem can be triggered by a call to the `CmiCheckpoint` function. The signature of this function is:

```
void CmiCheckpoint(CcpType t, CcpWhen w, unsigned int deltaT).
```

The `CcpType` argument is either `CCPQUIT` or `CCPPERIODIC` or `CCPONCE`.This signifies whether the program should stop execution after the checkpoint or if it should checkpoint

periodically or if it should checkpoint once and then continue execution. The period is given in milliseconds by the argument `deltaT`. The `CcpWhen` argument is either `CCPNOW` or `CCPAFTER` specifying whether the checkpoint is to be initiated immediately or after an interval given by `deltaT`. Checkpointing may also be triggered from the command line using the "+checkpoint s" parameter where s specifies the period in milliseconds. We can also specify that the program stop after the checkpoint by using the "+ccpquit" parameter.

6.2 Steps in the Checkpointing Process

A checkpoint may be initiated on any processor at any point in the program. At this point a message is sent to processor 0, which acts as the co-ordinating processor. This choice is arbitrary, any scheme could be used to determine the co-ordinating processor. The co-ordination strategy employed has the same general steps as in [5]. However, our scheme is more general in that it does not assume in-order delivery of messages.

A time line depicting the coordination steps is shown in Figure 6.1. A steps involved are as follows:

1. **Initiate checkpoint** - Once a call is made to the checkpointing subsystem triggering a checkpoint, the co-ordinating processor is informed. The co-ordinating processor then broadcasts a message to all processors in the system asking them to checkpoint.

2. **Suspend Processing** - When a processor receives a message initiating a checkpoint, it suspends processing of all subsequent messages. Neither does it send out any new messages. All messages that are recieved over the network are queued up to be saved to disk.

3. **Begin Saving State Data** - The processors start their state data 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 [24] 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. **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. On receiving the resumption messages, processors unfreeze their state and continue with the application. 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.

6.3 Data Organization

We now describe the layout of a Converse/Charm++ programs state on stable storage. Several possible schemes for organizing program state on disk were considered. Broadly all of these could be classified into two categories. Those that store the entire programs state as a single disk file and others that use a hierarchy of files and directories. For concurrent programming systems the latter were seen to be superior. Storing the state as a single file results in either increased complexity of the checkpointing and recovery protocols or a loss

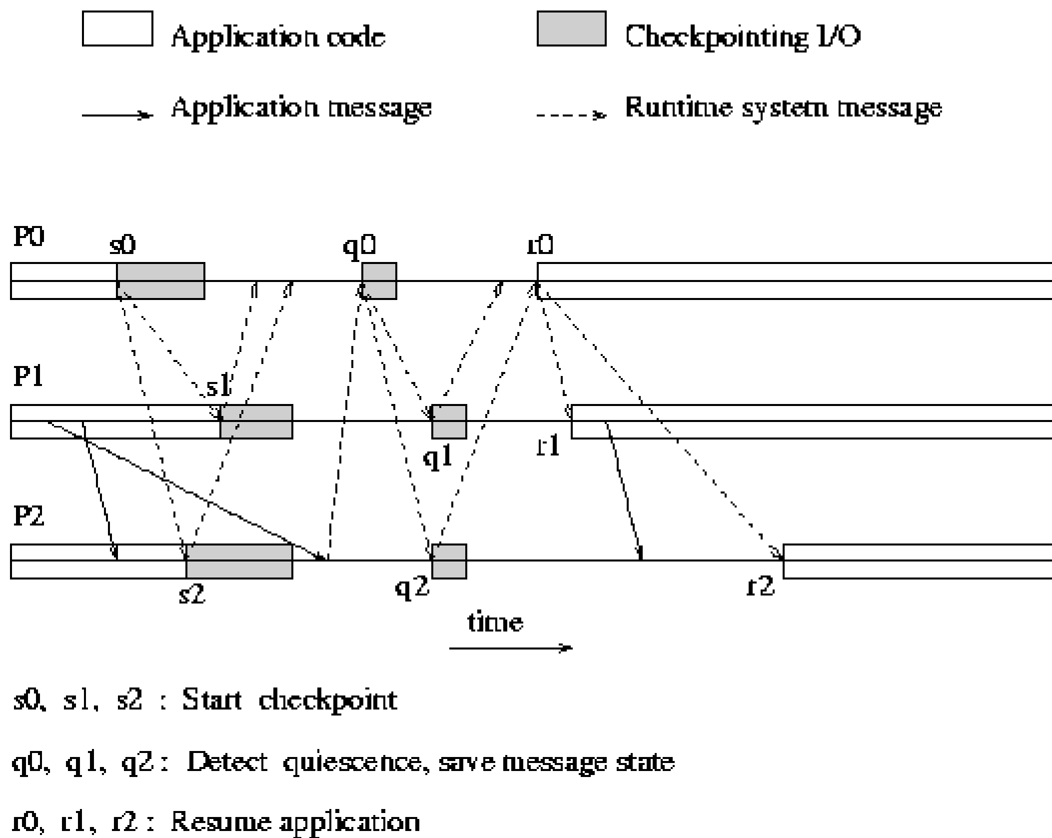

Figure 6.1: Time-line for processor co-ordination steps

in concurrency. If complete concurrency during checkpointing is desired (i.e. any process can write to the file any time it is ready with data) then we need to maintain a large number of indices into the file as pointers to the data stored by each process. In addition, allowing multiple processes to write to the same file at the same time requires the implementation and enforcement of a suitable locking protocol. In the absence of a shared file system it also requires that the checkpointing subsystem transport the data of each process to the right processor. On the other hand, if these difficulties are to be avoided then we must serialize the saving of data to disk by the different processes which poses another set of problems. The tasks involved in checkpointing are greatly simplified by moving the complexity to the file system. We choose the checkpointing protocols to be simple and have a high degree of

concurrency. Data layouts for Converse and Charm++ are now described:

- **Converse Data Layout** - To checkpoint Converse state, the user must have a directory named `CCPConverse` in each processes' working directory, on every machine that the program runs on and that has an independent filesystem. If several machines share a filesystem, the checkpoint directory needs to be created only once. Converse uses this directory as the root for all its checkpoint files and directories. Each processor saves its data in a directory called `CCPPE[n]`, where n is the processor number. These directories are one level down from `CCPConverse`. Module data is stored under each of the processors directories, with a different file for each module. In addition the `CCPConverse` directory also holds a file called `ConverseGlobals` that stores the programs global data. Figure 6.2 shows the data layout for Converse state.

- **Charm++ data layout** - For Charm++ state, the user must create a directory named `CCPCharm` in the same locations as `CCPConverse`. This directory is used as the root checkpointing directory by the Charm++ checkpointing subsystem. This directory holds several files, one for each virtual processor named `VirtualPE[n]`, where n is the virtual processor number. In addition it holds a file names `CharmGlobals` for storing Charm++ global data. Figure 6.3 shows the data layout for Charm++ state.

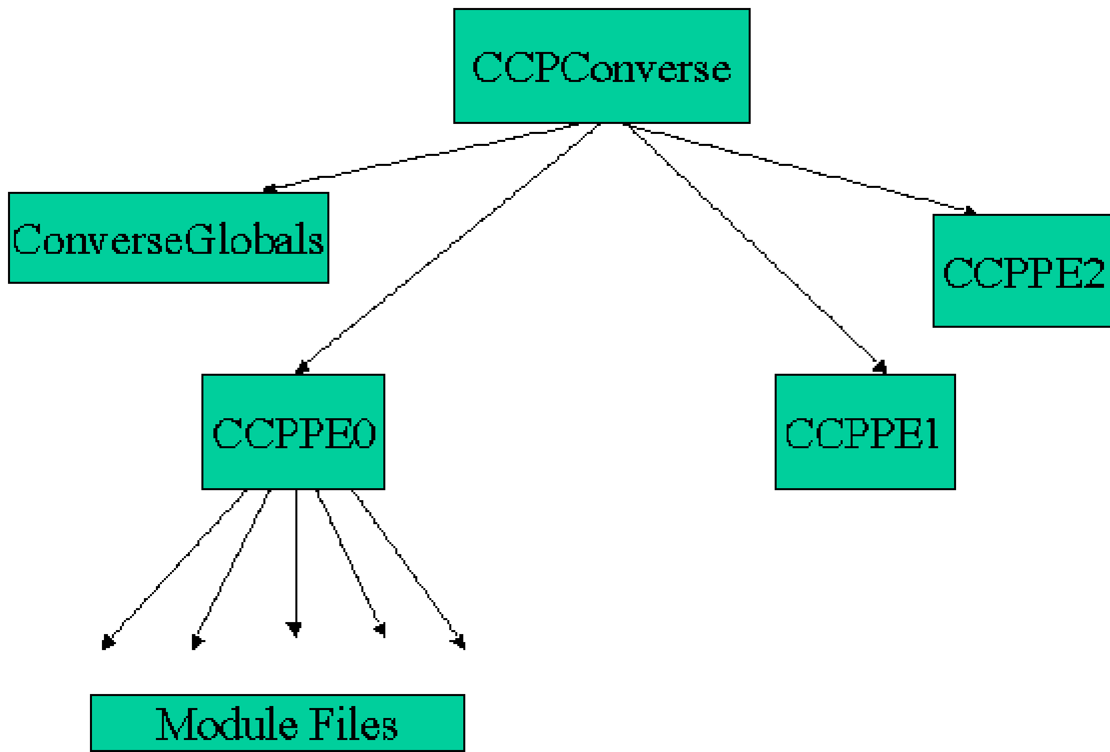

Figure 6.2: Organization of Converse state on disk

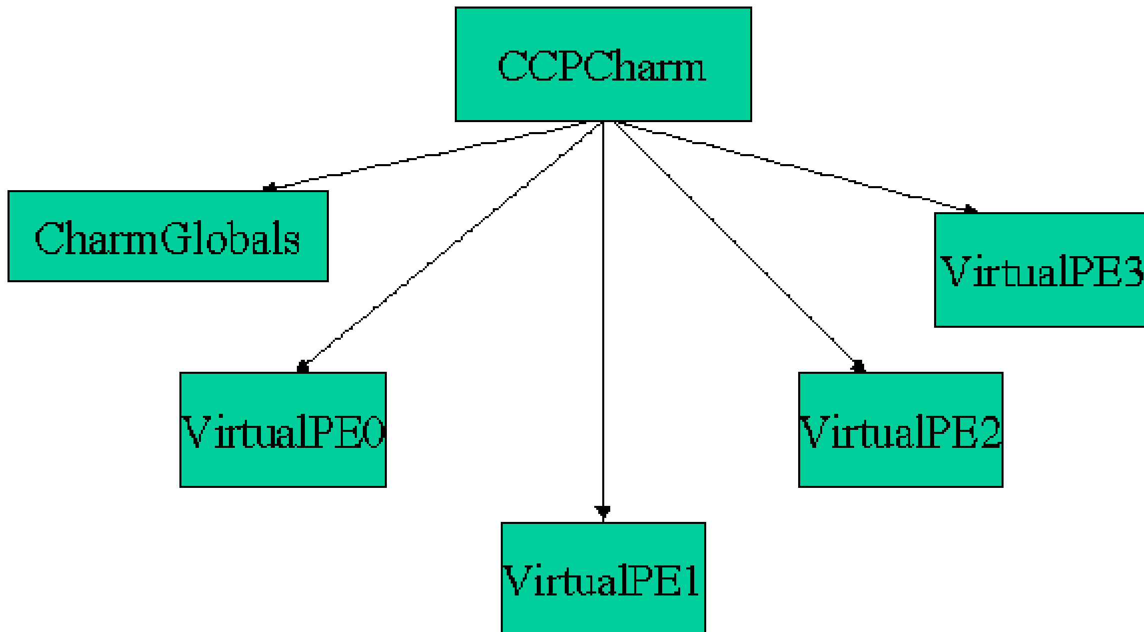

Figure 6.3: Organization of Charm++ state on disk

Chapter 7

Conclusions

We have presented here the design of a general purpose checkpointing facility for the Converse runtime system and Charm++ parallel programming system. The checkpointing subsystem provides several benefits to Converse/Charm++ programs. Among these are increased fault tolerance and the ability to 'shrink' and 'expand'. The ability to change the platform size after a checkpoint also provides a way to increase throughput of parallel machines running these programs. The checkpointing facility at the Converse level is extensible and checkpointing subsystems written for other parallel languages can be easily integrated into the program. The checkpointing subsystem provides the programmer with a uniform and consistent interface and is portable across all the platforms supported by Converse. Other strengths of the checkpointing facility are ease of use and low overhead.

7.1 Future Work

The area of checkpointing and restart for parallel programming systems presents several avenues for further exploration. The facility described in this thesis can serve as a foundation for developing more sophisticated checkpointing subsystems. We have described how Charm++ objects are required to have `pup` methods for the purposes of checkpointing and migration. The checkpointing subsystem can be further enriched by developing compiler support for automatically generating these methods, so that programmer involvement is fur-

ther reduced. Automatic generation of `pup` methods for statically allocated objects is fairly straightforward. Doing the same for objects that contain pointers into the heap requires support from the memory management subsystem. Converse includes a rudimentary memory management module that can be developed further to make this possible. Another possible attractive enhancement is making the checkpoint data platform independent, so that a program may checkpoint on say a cluster of Linux machines and then restart on a machine like the Origin2000. Addition of these capabilities would make the checkpointing subsystem a powerful tool that programmers can use to make their applications more resilient to system downtime and better able to exploit available computational resources.

Appendix A

Converse Interface

The programming interface for the Converse checkpointing subsystem has routines for triggering checkpoints, creating files and directories for checkpointing and saving and retrieving checkpoint data. Routines for triggering checkpoints are of interest mostly to application developers. File and directory management facilities oriented towards developers of checkpointing subsystems for parallel languages and programming systems. Checkpointing subsystems for parallel languages may use the routines for saving and accessing data or they may provide their own. Declarations for all of these functions occur in `converse.h`. These functions are described below:

A.1 Checkpoint Invocation

- `void CmiCheckpoint(CcpType type, CcpWhen when, unsigned int deltaT)`

  This function is used to invoke a checkpoint. The first parameter `type` may take the values `CCPQUIT` specifying that the program must stop after it checkpoints or `CCPPERIODIC` specifying that the program must checkpoint at regular intervals of time or `CCPONCE` specifying that the program should checkpoint once and then continue execution. The `when` argument may take the values `CCPNOW` specifying that the checkpoint must be invoked immediately or `CCPAFTER` specifying that the checkpoint must be taken after a given interval of time. The `deltaT` argument is used when a timing parameter is

needed be it a delay or a period. It represents time in milliseconds. In circumstances where both are required the same value is used for both. Programmers should be careful when using `deltaT` as both a delay and a period. In particular, small values of delay should not be specified.

- `void CmiRegisterCheckpointFn(CcpFn fn)`

  Though not an invocation function, a call to `CmiRegisterCheckpointFn` is used by a checkpointing module for a parallel language to register its primary checkpointing routine with the Converse checkpointing subsystem. The `fn` argument should have the type `CcpFn` given by: `typedef void (*CcpFn) (void)`

A.2 File and Directory Management

- `void CcpSetChkptRoot(char *dirname)`

  This function sets the root directory for checkpointing to `dirname`. Each checkpointing subsystem involved will typically have its own root directory under which it saves all its data. The `dirname` parameter may specify either an absolute pathname or relative to the processes' working directory.

- `void CcpMkDir(char *dirname)`

  This function creates a directory with name `dirname` under the checkpoint root directory and sets it as the current checkpointing directory. A checkpointing subsystem needs to be aware of two special directories its root checkpointing directory `CcpRoot` and the current checkpointing directory. The current checkpointing directory is always located under the directory tree rooted at `CcpRoot`. All I/O is done with files residing under the current checkpointing directory. It may be changed by calls to `CcpMkDir` or `CcpSetDir`; `dirname` must be a relative pathname.

- `void CcpSetDir(char *dirname)`

  This function sets the current checkpointing directory to `dirname` which must be a path relative to the current `CcpRoot`.

- `int CcpOpenFile(char *fname, char mode)`

  The `CcpOpenFile` function is used to open a file in the current checkpointing directory. The `fname` parameter gives the name of the file. The `mode` parameter may take the values 'r' or 'w' for *read* and *write* mode respectively. Upon failure, the function returns a value less than zero.

- `CcpCloseFile()`

  This is a macro used to close the current checkpoint file.

- `void CcpBeginSave(char *fname)`

  This is used to open a file for writing to at checkpoint time, `fname` specifies the filename. The file is opened in the current checkpoint directory.

- `void CcpEndSave(void)`

  At checkpoint time, used to close the current checkpoint file.

- `int CcpBeginRestore(char *fname)` Unlike the *Save* functions this routine is Converse specific. Other checkpointing subsystems should use `CcpOpenFile` instead. At restart time, this function opens for reading a file in the current checkpoint directory. In the case that a program expands after a checkpoint and the processor on which this function is called did not exist before the checkpoint it returns -1.

- `CcpEndRestore(void)`

  Also Converse specific, used at restart time to close the current checkpoint file.

- `int CcpNextPe(void)`

Another Converse specific function, `CcpNextPe` is used to take care of cases where a program shrinks after a checkpoint. A module recovering its data calls `CcpPe` every time it finishes recovering the data for an instance of itself. If no more instances remain `CcpPe` returns a value less than zero.

- `FILE* CcpGetFp(void)`

  This function returns a pointer to a FILE structure that represents the current checkpoint file.

A.3 Utility Routines for Saving and Restoring Data

All of these routines operate on the current checkpoint file.

- `void CmiReadInt(int *pi), void CmiSaveInt(int i)`

  Used to read and save an integer respectively.

- `void CmiReadChar(char *pch), void CmiSaveChar(char ch)`

  Used to read and save a character respectively.

- `void CmiReadDouble(double *pd), void CmiSaveDouble(double d)`

  Used to read and save a double respectively.

- `void CmiReadFloat(float *pf), void CmiSaveFloat(float f)`

  Used to read and save a float respectively.

- `void CmiReadBytes(void *p, int nBytes)`
  `void CmiSaveBytes(void *p, int nBytes)`

  Used to read and save an arbitrary sequence of bytes to disk, `nBytes` specifies the number of bytes to be processed starting at the address `p`.

- `void* CmiReadArray(void *array, int eltSize, CmiArrayEltReadFn readFn)`

  This function is used to read generic arrays of arbitrary sized elements from disk. The `array` parameter is a pointer to a chunk of memory where the elements must be stored. If `array` is `NULL`, the appropriate amount of memory will be allocated. The `eltSize` parameter specifies the size in bytes of each array element. In some cases where the array is an array of pointers a `CmiArrayEltReadFn` must be provided whose type is given by: `typedef void (*CmiArrayEltReadFn) (void *);`. The `readFn` must allocate memory for the element and set the pointer it is passed to reference the freshly allocated array element. A `readFn` may also be used if specialized processing of data while reading is required. A pointer to the array read in is returned.

- `void CmiSaveArray(void *array, int eltSize, int numElts,`
  `CmiArrayEltSaveFn saveFn, CmiSparseCheck isSparse)`

  This function is used to save generic arrays of arbitrary sized elememts to disk. The `array` parameter is a pointer to a chunk of memory where the elements are stored. The `eltSize` parameter specifies the size in bytes of each array element and `numElts` gives the number of array elements. In some cases where the array is an array of pointers a `CmiArrayEltSaveFn` must be provided whose type is given by: `typedef void (*CmiArrayEltSaveFn) (void *);`. The `saveFn` is passed a pointer to the array element to be saved. A `saveFn` may also be used if specialized processing of data while saving is required. Sometimes arrays are *sparse*, i.e. several elements contain trivial values which may not need to be stored. If this is the case the programmer may supply a `CmiSparseCheck`, whose type is given by: `typedef int (*CmiSparseCheck) (void *);`. A `CmiSparseCheck` is passed a pointer to an array element and should return a non-zero value if the element is sparse and zero otherwise. Every element is then checked with `isSparse` to determine if it needs to be saved or not.

- `char* CmiReadString(void), void CmiSaveString(char *s)`

These functions are special cases of the array routines and are used to read and store C-style strings. `CmiReadString` returns a pointer to the string read in.

- `void* CmiReadList(CmiListEltReadFn readFn)`

  Used to read singly-linked lists of arbitrary structures from disk, `CmiReadList` returns a pointer to the list that is read in. A `CmiListEltReadFn` whose type is given by: `typedef void* (*CmiListEltReadFn) (void *);` must be passed to this function. The function has been designed with the view that elements of linked lists have a member that points to the next element. The `readFn` is passed a pointer, it must read in an element and set the 'next' member of this element to its parameter. It must then return a pointer to the element just read in. A problem with this function is that lists are read in reversed.

- `void CmiSaveList(void *list, CmiListEltSaveFn saveFn)`

  Used to read singly-linked lists of arbitrary structures from disk, `CmiSaveList` must be passed a pointer to the list to be saved. In addition a `CmiListEltSaveFn` must also be passed. The type of the `saveFn` is given by: `typedef void* (*CmiListEltSaveFn) (void *);` The `saveFn` is passed a pointer to a list element. It is expected to save the element to disk and return a pointer to the next element in the disk. The list functions are therefore simple iterators over a singly linked list.

Appendix B

Converse PseudoGlobals

Different vendors are not consistent about how they treat global and static variables. Most vendors write C compilers in which global variables are shared among all the processors in the node. A few vendors write C compilers where each processor has its own copy of the global variables. In theory, it would also be possible to design the compiler so that each thread has its own copy of the global variables.

The lack of consistency across vendors, makes it very hard to write a portable program. The fact that most vendors make the globals shared is inconvenient as well, usually, you don't want your globals to be shared. For these reasons, we added "pseudoglobals" to Converse. These act much like C global and static variables, except that you have explicit control over the degree of sharing.

Three classes of pseudoglobal variables are supported: node-private, process-private, and thread-private variables.

Node-private global variables are specific to a node. They are shared among all the processes within the node.

Process-private global variables are specific to a process. They are shared among all the threads within the process.

Thread-private global variables are specific to a thread. They are truely private.

There are five macros for each class. These macros are for declaration, static declaration, extern declaration, initialization, and access. The declaration, static and extern specifications have the same meaning as in C. In order to support portability, however, the global variables must be installed properly, by using the initialization macros. For example, if the underlying machine is a simulator for the machine model supported by Converse, then the thread-private variables must be turned into arrays of variables. Initialize and Access macros hide these details from the user. It is possible to use global variables without these macros, as supported by the underlying machine, but at the expense of portability.

Macros for node-private variables:

```
CsvDeclare(type,variable)
CsvStaticDeclare(type,variable)
CsvExtern(type,variable)
CsvInitialize(type,variable)
CsvAccess(variable)
```

Macros for process-private variables:

```
CpvDeclare(type,variable)
CpvStaticDeclare(type,variable)
CpvExtern(type,variable)
CpvInitialize(type,variable)
CpvAccess(variable)
```

Macros for thread-private variables:

```
CtvDeclare(type,variable)
CtvStaticDeclare(type,variable)
CtvExtern(type,variable)
CtvInitialize(type,variable)
CtvAccess(variable)
```

A sample code to illustrate the usage of the macros is provided in Figure B.1. There are a few rules that the user must pay attention to: The `type` and `variable` fields of the macros must be a single word. Therefore, structures or pointer types can be used by defining new

types with the `typedef`. In the sample code, for example, a `struct point` type is redefined with a `typedef` as `Point` in order to use it in the macros. Similarly, the access macros contain only the name of the global variable. Any indexing or member access must be outside of the macro as shown in the sample code (function `func1`). Finally, all the global variables must be installed before they are used. One way to do this systematically is to provide a module-init function for each file (in the sample code - `ModuleInit()`. The module-init functions of each file, then, can be called at the beginning of execution to complete the installations of all global variables.

```
File Module1.c

typedef struct point
{
  float x,y;
} Point;


CpvDeclare(int, a);
CpvDeclare(Point, p);

void ModuleInit()
{
  CpvInitialize(int, a)
  CpvInitialize(Point, p);

  CpvAccess(a) = 0;
}

int func1()
{
  CpvAccess(p).x = 0;
  CpvAccess(p).y = CpvAccess(p).x + 1;
}
```

Figure B.1: An example code for global variable usage

References

[1] A. Acharya and B. Badrinath. Checkpointing distributed applications on mobile computers. In *Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems*, pages 73–80, Austin, Texas, September 1994.

[2] N. Bowen and D. Pradhan. Virtual checkpoints: Architecture and performance. *IEEE Transcations on Computers*, 41(5):516–525, May 1992.

[3] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[4] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. Technical Report 99-03, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, May 1999.

[5] K. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.

[6] C.Li, E. Stewart, and W. Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, 24(10):871–886, October 1994.

[7] C. Critchlow and K. Taylor. The inhibition spectrum and the achievement of causal consistency. In *Proceedings of the 9th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 31–42, New York, NY, 1990. ACM Press.

[8] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1999.

[9] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–6, 1980.

[10] E. N. Elnozahy, D. B. Johnson, and Y. M. Wang. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-96-181, Carnegie-Mellon University, October 1996.

[11] E. N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The Performance of Consistent Checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, Houston, Texas, 1992.

[12] Bob Janssens and W. Kent Fuchs. Experimental evaluation of multiprocessor cache-based error recovery. In *Proceedings of the 1991 International Conference on Parallel Processing*, volume I, Architecture, pages I–505–I–508, Boca Raton, FL, 1991. CRC Press.

[13] L. V. Kalé, Milind Bhandarkar, and Robert Brunner. Load balancing in parallel molecular dynamics. In *Fifth International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1457 of *Lecture Notes in Computer Science*, 1998.

[14] L. V. Kalé, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[15] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[16] Sanjeev Krishnan and L. V. Kale. Efficient, Language-Based Checkpointing for Massively Parallel Programs. Technical report, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, January 1995.

[17] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *The Charm++ Programming Language Manual, Version 5.0*, April 1999.

[18] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *Converse Programming Manual*, Jan 1999.

[19] J. Plank. *Efficient Checkpointing on MIMD Architectures.* PhD thesis, Princeton University, June 1993.

[20] J.S. Plank, M.Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter 1995 Technical Conference*, New Orleans, January 1995.

[21] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[22] R.Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[23] D. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[24] 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.

[25] M. Staknis. Sheaved memory: Architectural support for state saving and restoration in paged systems. In *Proceedings of the 3rd Symposium on Architectural Support' for Programming Languages and Operating Systems*, pages 96–102, April 1989.

[26] Y. Tamir and C. Equin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, Bellaire, MI, August 1984.

[27] Krishnan Vardarajan. Communication library for parallel architectures. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1999.

[28] Y. Wang. Maximum and minimum consistent global checkpoints and their applications. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 86–95, Bad Neuenahr, Germany, 1995.