SYSTEM SUPPORT FOR CHECKPOINT AND RESTART OF CHARM++ AND AMPI APPLICATIONS

BY

CHAO HUANG

B.S., Tsinghua University, 2001

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, 2004

Urbana, Illinois

Abstract

As both modern supercomputers and new generation scientific computing applications grow in size and complexity, the probability of system failure rises commensurately. Making parallel computing fault tolerant has become an increasingly important issue. Checkpoint/restart mechanism provides for fault tolerance capability as well as other benefits for parallel programmers. This thesis describes the on-disk checkpoint/restart mechanism for Charm++ and Adaptive MPI programming framework, its motivation, design, and implementation. This mechanism has proven to be useful in practice and can also be extended to implement other fault tolerant techniques.

Acknowledgments

I thank my advisor, Professor L. V. Kalé, for his inspiring guidance throughout my graduate research at the Parallel Programming Lab. I also appreciate the invaluable help from my fellow group members at PPL, especially Gengbin Zheng and Orion Sky Lawlor. It has been a truly enjoyable journey to learn and work at PPL. Last but not least, I would like to thank my family and friends for being incredibly supportive.

Table of Contents

List of Figures

Chapter 1

Introduction

Parallel computing has been playing an increasingly significant role in scientific and engineering research. Some problems, for instance protein unfolding simulation, by themselves are too large to be solved by any single machine. On the other hand, researchers keep increasing the size and/or resolution of their applications in order to obtain results of higher research value. Challenged with the need for higher computing capability, people created larger parallel machines to handle the ever growing needs of parallel applications. Nowadays it is quite common for a parallel program to run on a thousand-processor machine for weeks or months to solve a problem. Besides the increasing scale, many applications consist of algorithms that are hard to parallelize in conventional programming paradigms. There arises the necessity for new parallel model and supporting run-time systems (RTS) to ease this difficulty. Charm++/AMPI[16, 18, 13] has addressed this issue with its virtualization model. Its model has made an effective separation between the concerns of the programmer and the RTS. With virtualization, the adaptive overlapping between computation and communicationis enabled. In fact, the methodology is so successful that it has led to a Gordon Bell Award for "difficult to parallelize" application in 2002[20].

As the size of new parallel machines multiplies, the probability of system failure increases commensurately. As a result, methods to ensure that programs "survive" infrastructure failures become a hot topic. There are different types of survival. For example, the system may be able to automatically recover from the problem without affecting the behavior of applications. In other situations, the run-time system or the application itself are required to be involved in the recovery. The checkpoint/restart mechanism described in this thesis is aimed at providing a reliable execution of parallel programs in Charm++/AMPI at the run-time system level.

1.1 Motivation

Checkpoint and restart mechanism is an important effort toward fault tolerance. It provides the programmer with the capability to take snapshots of the application, periodically or on command. The checkpoint mechanism does not necessarily log every single bit of the information in the system at that moment; only the essential data are saved on more reliable storage. On occurrence of system failure, these checkpointed data would be used to restore the application to the previous checkpoint, and the forward progress of the application can resume from that point.

The checkpoint/restart mechanism in Charm++/AMPI has benefits beyond fault tolerance. For example, if we can restart the parallel program on different number of processors, the system can immediately work around the failure. This is ideal in the typical scenario where 1 processor out of 1,000 has failed; the surviving 999 will not have to wait for the dead processor to come back; the application can simply restart on the surviving processors.

In this thesis we present a checkpoint and restart mechanism that fully supports the above described features.

1.2 Contributions

The contributions of this thesis include:

- A checkpoint/restart mechanism for Charm++ and Adaptive MPI run time system. This mechanism can be used in any parallel program written in Charm++ or AMPI on all supported platforms.

- Part of the work of this thesis is used for further fault tolerance efforts in Charm++/AMPI. Examples are two ongoing projects - in-memory checkpoint mechanism and message-logging based fault tolerance mechanism.

1.3 Organization

This thesis is organized as follows. In the first chapter we provide the motivation for this thesis. Chapter 2 includes an introduction to the checkpoint/restart mechanisms. Checkpoint/restart mechanisms for Charm++ and AMPI will be described in detail in Chapter 3 and 4 respectively. In the last Chapter a conclusion will be drawn and the future work will be discussed.

Chapter 2

Checkpoint and Restart Mechanism

2.1 Features of Checkpoint/Restart Mechanism

Checkpoint/restart is a mechanism to preserve the state of a running system and to restore the system from the saved information. In parallel programming, checkpoint/restart mechanism must record and reconstruct the states of all the involved processing elements.

The most important benefit that checkpoint/restart provides is fault tolerance. By taking checkpoints periodically or on command, the system is capable of recovery after infrastructure failures. Periodic checkpoint is a straightforward solution to regular system failures. In other cases, especially for a system with fault monitoring mechanism, checkpoint schedule can be adjusted with the frequency of system failures to save on the overhead of checkpointing in failure-free execution.

In the context of parallel programming, checkpoint/restart mechanism is usually more complicated than in sequential programming. Part of the complexity is due to the communications between processors. At the time of system failure, not only the machine states but also all the messages in flight are lost. Decisions must be made on whether and how to save the flying messages in the wire. Some restrictions on when the checkpoint might be taken might be imposed for this reason.

The capability to reorganize the load in the application on recovery is a significant optimization. A parallel computing platform usually consists of a large quantity of nodes and components. Typically, the system fails with only one or two of its components ceasing to function. In this case, having all the nodes idle while waiting for the one or two components to be fixed is very inefficient. If we are able to reorganize the load in the recovery phase, we have the flexibility of adapting the application to the new environment after recovery. The example of being able to restart on different number of processors has been described in the previous chapter.

Besides fault tolerance, checkpoint/restart can also contribute to more efficient scheduling on parallel platforms. With checkpoint/restart capability, a long running job that requires a large amount of system resource can be saved to disk and suspended during busy periods and restarted when the system is less busy. This technique is called Gang Scheduling[10] and is used in various contexts such as operating system. With this flexibility in scheduling, the overall throughput and usability of the supercomputing platforms can be improved.

2.2 Categories of Checkpoint/Restart Mechanism

In this section we discuss different categories of checkpoint/restart mechanism.

2.2.1 Application vs. Run-time System Level

Checkpoint/restart mechanism can be implemented at different levels in the system. Some choose to hard code the checkpoint mechanism in their application. The programmers write code to decide which part of the data structures to save during a checkpoint as well as where and how to save them. Smooth restarting is also a consideration. This approach can be very efficient, because the programmers understand their applications' needs the best. However, there are a few problems with application level checkpoint/restart. First, it is usually prohibitively difficult to modify the code of past applications to do checkpoint/restart. Second, this approach holds the programmers responsible for the correctness and portability of the application's checkpoint/restart mechanism, while this job could be automated in a more generic way.

As an alternative, checkpoint/restart mechanism can also be implemented at Run-Time System (RTS) level. The RTS has access to all kinds of data structures to be checkpointed. Another main advantage is its generality. One does not have to make major changes to the code, especially for hard-to-modify legacy code, to make it checkpointable. Generality could be a disadvantage too, when the RTS saves more than necessary data, due to the lack of knowledge specific to an application. This problem can be solved by allowing the programmers to give hints to the RTS as to what is the essential data in their applications.

2.2.2 Coordinated vs. Uncoordinated

Parallel checkpoint/restart has additional complexity of coordinating multiple nodes in the system. In coordinated checkpointing, the processors exchange control messages to ensure a consistent global state. It can be implemented as a blocking barrier on all involving processors. This approach is simple to implement, with its obvious correctness and transparency. At restart phase, all the processors just roll back to the latest global checkpoint and start the program. The other flavor of coordinated checkpointing does not require the global barrier. Each processor continues after their checkpointing without blocking. Chandy and Lamport[5] suggested an algorithm to ensure a consistent global state without a global barrier. This method alleviates the cost of synchronization but does not totally remove it.

The simplicity of coordinated checkpoint comes at the price of synchronization overhead, especially in large scale parallel systems. To eliminate the overhead and make checkpointing scalable, uncoordinated checkpointing removes the requirement for control message exchange among processors. Each processor decides when to checkpoint independently. The key is to ensure global consistency at the restart phase because all the checkpoints are not necessarily consistent. In order to do this, some processors might be rolled back more than once to reach a good global state, due to the lack of coordination among the checkpoints. Potentially this type of mechanism may suffer from rollback propagation or domino effect[22]. Another issue is garbage collection. Since all the checkpoints are potentially necessary in a restart, the checkpoint files may grow infinitely in space. The system has to figure out a way to reclaim checkpoint space[26] in such a system.

Other than the categories discussed above, there is the issue of form in which checkpoints are recorded. Checkpoint files are usually saved to reliable storage such as hard disks. As the price of reliability, hard disks have relatively higher I/O overhead, thus lowering the efficiency of checkpoint/restart. Especially, when checkpoints are taken periodically, the life of a checkpoint can be transient and does not deserve the high overhead. To meet the demand of lower overhead and acceptable reliability, the memory of peer nodes in a parallel system can be used. With good design of the algorithm, in-memory checkpointing or in-memory combined with on-disk checkpointing, enjoys the advantage of low cost yet high reliability.

2.3 Related Work

Checkpoint/restart mechanism in parallel computing has been a very popular research topic. CoCheck[24] sits on top of message passing library and implements its functionality in its own MPI library tuMPI. A special process is used to coordinate the checkpointing, triggering the processors to save their states as well as incoming messages until all processors have finished doing so. At restart phase, "receive" operations need to first look at the saved messages for any match. This restricts when the checkpoint can be taken and sometimes may change MPI's semantics of synchronous communication.

CLIP[6] is another project implemented on top of message passing paradigm and it is specifically built for Intel Paragon. They claim to be a *semi-transparent* mechanism because the user is expected to make minor changes to invoke the checkpoint procedure. Also it is the programmer's responsibility to make sure that it is invoked at an appropriate time. Because a totally transparent implementation usually involves the operating system and can be very difficult to implement, this tradeoff does make sense in many cases. A notable point is that CLIP is built on top of a compiler-based checkpointer libckpt[21].

An actively ongoing project called MPICH-V[7] features multiple fault tolerant protocols. Built on MPICH core, MPICH-V[2] and MPICH-V2[3] based their protocols on uncoordinated checkpoint. To ensure the consistency, messages are logged in both protocols. In the first version, *Channel Memory* is devised to process the communication from one PE, reducing the bandwidth by half. In the second version, a communication daemon is created for each computing node to reduce this overhead. Another version of this project MPICH-V-CL take coordinated checkpoint following Chandy-Lamport algorithm.

Similar to CLIP project, LAM/MPI team based their checkpoint/restart module on a kernel-level checkpoint/restart mechanism called BLCR, or Berkeley Lab's Linux Checkpoint/Restart[9]. It is on the kernel level, and therefore it is limited to Linux platforms. The low level implementation hides the checkpointing from the application developer, and the overhead introduced by the LAM/MPI Checkpoint/Restart Framework[23] is shown to be very small.

Chapter 3

Charm++ Checkpoint/Restart Mechanism

3.1 Charm++

Charm++ is an object-based, data-driven parallel programming language. It provides a machine-independent Run-Time System(RTS). A program written in Charm++ can be compiled and run on many of the most powerful supercomputers without modification of the code. Some of the supported platforms include Lemieux at Pittsburgh Supercomputing Center(PSC), Origin2000 at National Center for Supercomputing Applications(NCSA), IBM SP supercomputer and the BlueGene/L, as well as clusters of Linux workstations including SMPs and single processor nodes. Charm++ is built on an underlying framework called Converse [15, 17]. Converse provides a machine-independent layer for message passing, message handling and scheduling, as well as basic thread functionalities.

Before we introduce Charm++ programming model and its components, let's first look at a fundamental concept in Charm++: **processor virtualization**.

3.1.1 Processor Virtualization

The idea of processor virtualization is to let the programmer divide the work into chunks, and let the system map these entities to physical processors. The number of chunks, or virtual processors, is typically much larger than the number of physical processor, and is usually independent of the latter.

Processor virtualization allows for a set of benefits. The first benefit is better software engineering. Virtualization let the programmer focus on the logical interaction among the parallel jobs while neglecting the configuration of physical processors. This is in compliance with the principle of good coupling in software engineering: program entities are coupled only when they are logically connected. In contrast, MPI's processor-centric model often hinders the programmer from following this principle. Secondly, efficiency and flexibility are improved for parallel programs. In the object-oriented model, adaptive overlapping between communication and computation comes naturally, while automatic load balancing and checkpointing/restarting can be done in an easier way.

3.1.2 Programming Model

Charm++ is object-based and data-driven, meaning that control of the program is dictated by messages sent from object to object. An object sends a message to another through the recipient's "entry point", or a specially registered function invoked by remote messages. The remote invocation is asynchronous: a function call returns immediately after sending out the message, without waiting for the response to come back. This idea of asynchronous remote invocation makes adaptive overlapping of computation and communication possible. For example, when an object needs some

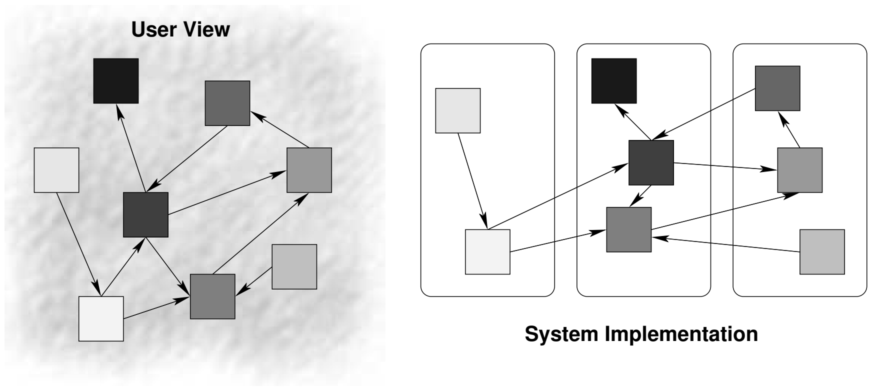

Figure 3.1: Charm++ maps objects onto processors

Charm++ allows computation to be divided among objects that are mapped to physical processors at runtime. On the left, the programmers focus on designing the interactions among parallel objects. To the right, Charm++ RTS maps the objects onto physical processors and supports the interactions. The communication relationships, as illustrated by arrows between the objects, are preserved in the mapping. Charm++ can automatically balance the load on the processors by moving objects between processors. (Figure taken from [16]) data from another object, it sends out a request and while the data is being prepared on the other side or in flight in the internetwork, the object can make full use of the CPU time to perform some useful computation.

The run-time system maps objects onto processors, so it is also responsible for routing message from object to object. A mapping mechanism maintains a default mapping and records all changes in object location. This means even after an object moves away from its default (or home) processor to another processor, messages addressed to it will still be able to reach it. Object migration allows for automatic load balancing, by which workloads can be dynamically balanced among processors at run time. Taking this idea one step further, migrating objects onto and back from disk files is the basic idea of checkpointing and restarting.

On the recipient's processor, a message scheduler receives and buffers the message and determines which object the message is intended for. If the recipient has migrated off the processor during load balancing, the message is forwarded to the correct location. One thing to note is that in Charm++ the messages are not guaranteed to arrive and be scheduled in the same order as they are sent.

3.1.3 Parallel Objects

There are several different kinds of parallel objects in Charm++. To checkpoint/restart a Charm++ program, we will have to take care of each type. Therefore in this section we give a brief introduction of them. [8]

- **Chare**

  Chares are concurrent objects with methods that can be invoked remotely. They are no different than ordinary C++ object except for their special methods known as entry methods or entry points. The entry methods are registered in an interface ".ci" file, where the object class is also declared. Also a "proxy" can be created for this chare. A proxy's role is just as its name suggests. It is a delegation of its corresponding chare or other types of parallel objects on all processors. So after a proxy is assigned for the chare, some entry methods registered for it, Charm++ supports the invocation of these entry methods on the proxy from any processor in the system.

There is a special variant of chare called Mainchare. A mainchare's default constructor is the entry point to the user code in a Charm++ program. Mainchare is usually used as a general control center, maintaining some global data. One limitation of this use is that a mainchare has only one copy and it resides only on processor 0.

- **Chare Array**

  Chare arrays are arbitrarily-sized collections of chares. The entire array has a globally unique identifier, and the array ID can be used to derive a proxy for this chare array. Chare arrays are different from regular C array, in that the elements are not necessarily sequentially ordered in a linear space. Essentially the elements are a collection of objects, each with a unique index of type which can be a single integer for 1D array, several integers for a multi-dimensional array, or an arbitrary string of bytes for user defined index type (e.g. a binary tree index).

  Array elements can be dynamically created and destroyed on any processor, and messages for the elements will still arrive properly. Array elements can be migrated at any time, allowing arrays to be efficiently load balanced. Array elements can also receive array broadcasts and contribute to array reductions. To implement these functionalities in an efficient way, Charm++ devised a special type of chare arrays called Groups which is described in detail next. A system group Array Manager is created for each chare array and the array manager maintains information such as which indices have objects created and where they are (resident processor and object pointer) and controls the message routing, object migration and so on.

- **Group and Nodegroup**

  Groups are a special kind of chare arrays: they have exactly one element on each physical processor. Groups are useful in several ways. First, groups can maintain global data in a distributed fashion. They do the same job that the mainchare does, but providing a local copy to each processor improves the scalability of the program. For this reason, groups are also called Branch Office Chares (BOC). For example, when an object on a random processor needs some global data, it is able to fetch it on the local group element to save on communication overhead. Groups are also used to hold processor specific data. The Charm++ programming model alleviates the programmer's responsibility of always thinking of physical processors, but the run-time system is fully aware of processor specific information. In the Charm++ RTS, groups manage the information like which chare array elements currently reside on a local processor.

Nodegroups are a variant of groups, with one element on one node of SMP machines. The same way that groups improve scalability at the processor level, node groups do the job on the node level. As an example, the reduction tree in the SMP version of Charm++ is formed on both processor level with group and on node level with nodegroup to further improve scalability.

In the run-time system, two tables are maintained: one for all groups, system and user-defined, and the other for all nodegroups. Each entry of the table includes information like group name, group ID, its constructors and the object pointer. From these central tables, the system has access to all groups and nodegroups, and via array managers, to all chare arrays. Individual chares, however, are not included in any system table, and thus are not accessible to the system without an explicit object pointer.

- **Readonly Data**

  Readonly data are not really parallel objects. Their values are assigned in user mainchare at start-up and broadcast to all processor as read only. Proxies of the mainchare, other chares, and chare arrays are typically declared as readonly data in the interface file.

3.1.4 Program Flow

A Charm++ program starts with initialization of various variables, data structures and modules in the run-time system. For example, tables for groups and nodegroups are created and initialized here. Some system modules such as the load balancing module and the Charm tracing module are initialized too.

Following initialization is registration. In this phase, system modules and user modules are registered with Charm++ on processor with rank 0 of each node. A module is a collection of related objects, functions and data bundled together. One module may also include other modules to complete its functionality. For instance, “CkArray”, the module for array managers, includes the definition of array manager and its entry methods, definitions of a few helper functions, and also modules that implement reduction and location management.

After initialization and registration, a few things are done from processor 0 for the whole system. First constructors of mainchares are invoked. There may be more than one mainchare in the system, because many modules may want to set up their utilities before anything really happens, and a mainchare's constructor is their best choice. The checkpointing module uses this technique to create a special group used by the checkpoint mechanism called checkpoint manager, with a branch on each processor. Then readonly data are collected using a PUP'er (described in Section 3.1.5) and broadcast to all processors. After all these are done, function _initDone() is invoked to do some clean-up work.

After the run-time system has initialized itself, the user mainchare's constructor is invoked, and thus the user code starts execution. Typically, user code creates a bunch of parallel objects like chare arrays and groups. These objects interact with one another by communicating messages, and messages drives the flow of the execution.

Let us take a stencil computation as an example. In this problem, we have a matrix of data, and every time step we want to update each matrix element's value to be the average of its old value with its neighbors' values. The problem simulates heat dissipation and other scientific computations. To parallelize this problem, we divide the matrix into many small chunks, and the run-time system will map these chunks onto physical processors available. Each chunk will need the border data from its neighbors to do its computation. So a timestep starts with all chunks sending out border data to their neighbors. Depending on the destination of the messages, the message passing can be as simple as a message copy, or as complicated as data transfer across the interconnect. Consequently, the border data do not arrive at the destinations at the same time. Some lucky chunks get the data they need and starts computation early, while others are still idle waiting. When the late chunks get their turn do the computation, the early chunks might be preparing for the next round of communication. The example illustrates the typical control flow of a Charm++ program: message-driven and asynchronous. Thereby, Charm++ is enabled to render the full power of processor virtualization.

After all work is done, CkExit() is called and the program is terminated.

3.1.5 PUP Framework and Object Migration

In order to implement object migration, the run-time system needs a way to collect all the data, static and dynamic, and serialize them into a buffer before departure from the source processor, and vice versa after arrival at the destination processor. The RTS of Charm++ provides a collection of efficient and elegant classes that enable parallel objects to be serialized and deserialized. This collection of classes is called Pack/UnPack or the PUP Framework. Other than to serialize/deserialize an object, the PUP framework can also be used in any functionality that requires traversal of an object's data members.

With the PUP framework, object migration can be as simple as 3 steps: 1) packing the object's member data into a buffer, 2) shipping this buffer to the destination processor, and 3) reinitialize the object with the data unpacked from the buffer on the new processor. Similarly, to checkpoint a program, we need to pack up the states of all the parallel objects and ship them onto the disk, and restarting is read the object states from the disk and migrate them back onto processors. Therefore, checkpointing/restarting can be viewed as a variant of objection migration between the processors and the disks.

The motivation behind PUP framework, a generic way to traverse an object's data members is to eliminate code duplication. Consider what one needs to do to migrate an object. First the object needs to be sized, so that the total size of buffer can be decided. Second the actual data items in the object should be packed one by one. We will have different operations depending on the different destination: memory or disk. And this is also true to the procedure of restoring data from the buffer. All these operations share the same essential form: traversal of data members, and that is exactly what PUP classes are doing.

The PUP framework (namespace) consists of the following set of classes:

- `PUP::able`

  This is the base class of all system-allocatable objects, which chares, chare array elements, groups and nodegroups all belong to. This class declares an important function `PUP::able::pup` that takes a `PUP::er` as parameter. What this call does is to use the `PUP::er` to traverse its data members. To make use of PUP framework, a `PUP::able` subclass is expected to implement its pup routine, which specifies how the class data are to be traversed. To make the format simpler, an operator "pipe(|)" is defined as calling the pup routine.

- `PUP::er`

  `PUP::er` is the abstract superclass of all other traversing classes in this framework. When a `PUP::able` object calls its pup function, a `PUP::er` object is passed in to do the desired job. This class has methods for dealing with all basic C/C++ data types, expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method. When the programmer uses a `PUP::er`, he/she does not have to know what operation is applied on the member data of the object. Polymorphism provides for the elegant format and versatile functionalities of this framework.

- `PUP::sizer`

  A subclass of `PUP::er`, `PUP::sizer` goes through the data members and finds the total number of bytes to pack. For example, before the system preallocate the buffer to hold the flatten data, the sizer is used to get the buffer size in bytes.

- `PUP::mem`

  This is the abstract superclass of binary memory buffer packing/unpacking classes. It deals with the interaction between object data and the binary memory buffer. It does not discriminate which direction, i.e. packing or unpacking, is the interaction, but just implemented several fundamental functions like seek for other memory operations. `PUP::mem` comes in two flavors, `PUP::toMem` and `PUP::fromMem`. `PUP::toMem` packs the object it operates on into a preallocated, presized memory buffer. The most general way to pack an object into memory buffer is to invoke pup on the object using an instance of `PUP::sizer` to determine the size of the object, then allocate a buffer of required size and invoke again with an instance of `PUP::toMem` that has been initialized with the allocated buffer. `PUP::fromMem` unpacks the state of the object it operates on from a given contiguous memory buffer.

- `PUP::disk`

  This subclass of `PUP::er` is much like `PUP::mem` other than it deals with different type of media - disk file. While `PUP::mem` takes a preallocated memory buffer as parameter, a instance of this class does its job on an opened disk file. It is not responsible for opening or closing the file. It also provides operations like start seek, seek and answer current position. As usual, there are two variants: `PUP::toDisk` and `PUP::fromDisk`.

```
class HelloClass{
  public:
    int x;
    char y;
    double arr[4];
    float *darr;
    init(int x_) {
      x=x_;
      darr=new float[x];
    }
    void pup(PUP::er &p) {
      p|x;
      p|y;
      p(arr,4);
      if(p.isUnpacking())
        darr=new float[x];
      p(darr, x);
  }
};
```

Figure 3.2: A simple class declaration showing the `pup` routine

In general, `PUP::er` is a collection of calls that represent the object data on different format and on different media. Now here we show an example of how PUP framework is used in Charm++ programming. Assume in the program we have class named HelloClass, and it has several data members. Here in Figure 3.2 how to write a typical pup routine for it.

As you can see, the pup routine goes through all the data members with different type. The programmer does not have to worry about the problem of alignment like how many bytes of padding should be placed after the char to correctly align the following double array. The PUP framework will do it automatically. Also the issue of endianness is taken care of. The programmer does not even have to think about the direction of packing or unpacking, except for the case of dynamically allocated data. If the `PUP::er` is unpacking, a new array should be dynamically allocated on the destination processor, just as on the original processor by the constructor.

In Charm++, when the object migrates, the pup routine on the object is automatically invoked

```
int main(void){
  /* create and initialize object hello */
  HelloClass hello;
  hello.init(8);
  hello.y = 'a';
  hello.arr[0] = 0.1;
  hello.arr[1] = 0.2;
  hello.arr[2] = 0.3;
  hello.arr[3] = 0.4;
  for(int i=0;i<8;i++)
    hello.darr[i] = (double) i;

  /* flatten hello into a memory buffer.
     first size it */
  PUP::sizer s;
  hello.pup(s);
  void *buf = malloc(s.size());
  /* then pack hello into buffer */
  PUP::toMem mto(buf);
  hello.pup(mto);

  /* create an empty object hello2,
     then restore it from the memory buffer */
  HelloClass hello2;
  PUP::fromMem mfrom(buf);
  hello2.pup(mfrom);
}
```

Figure 3.3: A program illustrating the use of `PUP::toMem/fromMem` routines

so that all data items are sized and flattened on source processor, and then on arrival, the object data is restored from the flattened format. This procedure is done in Charm++'s Array Manager, the system group that manages an array including its migration. Here Figure 3.3 shows how an instance "hello" of HelloClass is created, initialized, and packed into preallocated memory buffer, and then another uninitialized instance "hello2" get restored directly from the buffer. So afterwards hello2 should have exactly the same data as hello.

For checkpoint/restart purposes, the subclasses `PUP::toDisk/fromDisk` will be used. The programmer opens a file and pass the file pointer into the `PUP::er`'s constructor to initialize it. In Figure 3.4 is an example of such code.

```
FILE* fGroups = fopen("Groups.dat","wb");
PUP::toDisk pGroups(fGroups);
CkPupGroupData(pGroups);
fclose(fGroups);
```

Figure 3.4: Example of `PUP::toDisk` used in checkpointing

3.2 Charm++ Checkpointing

To checkpoint a Charm++ program, we need to save the state of the whole program. As described in Section 3.1.2, a Charm++ program relies on the following object states: Readonly data, chares, mainchares, groups, nodegroups and chare arrays. Other than the state of parallel objects, we also need to take care of the control flow; we want to be able gain control of the program as well as yield control when checkpointing is done.

Now let's take a closer look at the first requirement: saving the state of objects. Readonly data exists on all processors, but all copies are the same, so one copy of all readonly data should suffice. Chares can be created anywhere and do not register themselves on any system table. There is currently no way to find all the chares in the system and save them. This is acceptable because in typical Charm++ programming, Chares are created to hold temporary data and do not require persistent existence. Mainchare is an exception. Especially, user mainchare usually holds important control data like the number of timesteps, current counts of objects, so there is a necessity to checkpoint/restart mainchares. On the other hand, mainchares reside only on processor

0 and has a mainchare table, so that all these operations can be done locally on processor 0.

Groups (and nodegroups)[1] have a branch on each processor or node. However a local copy of each branch is not necessarily required. Due to the different uses of groups, there are different ways to checkpoint/restart them. Sometimes groups are used to hold same data in a distributed way, and thus all branches are the same. Obviously only one copy on disk file is enough. Groups may also hold processor-specific data, which then may or may not be easy to reconstruct. If the processor-specific data is difficult to recompute, a local copy of each branch is on demand.

Chare arrays are even more complicated than groups. They are accessible only through their array managers, which will be identified hopefully during the processing of the groups. Then each branch of the array manager will take care of their local array elements.

The procedure of checkpointing a Charm++ program has the four steps.

1. **Synchronization**

   Since our approach is blocking coordinated checkpointing, the concurrent processes need to reach a global barrier before a checkpoint can be taken. This requires the programmer is expected to decide the point where nothing important (e.g. computation of timestep results) is going on and no message is flying in the wire. Before the interface function is called on one processor (the "master processor") to invoke the following steps, the programmer prepares a "callback" function so that the control of program can re-enter user code after checkpoint or restart is done. This callback function in Charm++ is an object and passed into the interface function `CkStartCheckpoint`.

2. **Saving Readonly Data**

   Because readonly data is spread on all processors and they are all same, we can take a snapshot of readonly data from the master processor. The readonly data items are accessed from a global system table `_readonlyTable[_numReadonlies]`.

3. **Saving Mainchares**

   There are 2 things expected from the programmer when the he/she wishes to save the main-

[1]Groups and nodegroups have little difference except for their locations. For the sake of simplicity, groups refers to both groups and nodegroups hereafter.

chare. 1) The master processor must be processor 0; 2) The programmer should write a PUP routine for the mainchare and declare it as [migratable] in the .ci file.

From processor 0, the system has access to the table holding the mainchare list. Object is taken out one by one from the list, pup routine executed with the `PUP::toDisk` PUP'er, and all mainchares are saved.

4. **Saving Groups and Nodegroups**

   From this step on, the control is passed to all branches of the system group that is devoted to checkpointing `_sysChkptMgr` with a broadcast from its proxy. On each processor, the local branches of groups are saved, in the similar fashion that mainchares were saved. First the number of groups and number of nodegroups are saved, then a loop iterates over all groups. For each group, its globally unique group ID, name, and constructor's handlers are all packed in, and then the object pointer is grabbed and finally the object's member data are packed in.

5. **Saving Chare Arrays**

   Again we iterate over all groups, to identify the location manager, the helper group keeping track of array elements' locations. On a location manager, an iterator is created to go through every location of each element. Here we use location of array element because Charm++ allows user to bind two arrays together, meaning they have same size and their corresponding elements always migrate together, i.e. always have the same location. This is implemented by treating the bound elements as one "location", and the locations are managed by the separate location manager, not any single array manager. Therefore, it makes more sense to iterate through all locations and pack corresponding array elements into disk file.

6. **Callback**

   After the above steps, all the significant program states are saved. Another global barrier is reached to make sure all the checkpoint manager branches have finished their job, and then the callback is invoked to return the control to user code.

3.3 Charm++ Restarting

Simply put, restarting is the reverse process of checkpointing. Readonly data and parallel objects in the program are restored from disk files one by one, and the callback is invoked to restart the user code. However, there are a few issues to deal with to make restarting possible.

First issue is the *control flow.* Unlike checkpointing, where user code is paused and checkpoint code kicks in, restarting a Charm++ program is actually a modified way of starting a brand new program. In fact, to restart a program from a previous checkpoint, the user simply adds a "`+restart DIR`" option onto the command line, with `DIR` pointing to the checkpoint directory. There is no need for another system group like checkpoint manager, however, because right after initialization and registration (refer to Section 3.1.4), the function CkRestartMain() takes over the startup on all processors. Readonly data are restored locally, therefore the broadcast is saved. Mainchares are restored on processor 0, with whatever data they had at the moment of checkpointing. And last groups, nodegroups and chare arrays are restored on their original processor.

Secondly, Charm++ checkpoint/restart mechanism provides the programmer with the flexibility of restarting on different number of processors from checkpointing. This flexibility is important when the failed processor is not expected to recover very soon. The program can restart on N-1 surviving processors and continue its job.

This flexibility also gives rise to potential problems for parallel object restoring. Groups, for example, will not have the same number of branches, and their data might become inconsistent. To remedy this, there are two alternatives: 1) the copy from processor 0's branch could be restored on all processors; or 2) each branch recomputes their data. Chare arrays also suffer from this, since some element's home processor might be gone in a system failure. An easy cure is to redistribute the elements on surviving processors, or simply to transfer the workload of the dead processor(s) to some random surviving processor(s).

Last but not least, the process of "restoring" a parallel object is worth some explanation. Restoring is more complicated than checkpointing since we have to create an object before we can fill the saved data back in. For this special purpose, we use a special type of constructor called a Migration Constructor. A migration constructor is typically used to create an empty object ready for unpacking. Initially it is meant for when the object just arrived at a new processor after migration. In other words, migration constructors and pup'er routines are required for object migration as well as checkpoint/restarting.

The programmer can decide if a certain array or group is checkpointed or not. For example, if the migration constructor is not provided, there is no way to restore this object, so this object will be skipped in checkpoint/restart. This enables helpful optimization to programmers. During the execution of a typically parallel program, especially scientific computation, there will be multiple arrays and groups created to do the computation. Many objects compute the interim results only to pass to the next phase. This kind of objects are not worth the space when the data set they are working on is huge. In Charm++ checkpoint/restart framework, the programmer can choose not to save them by not providing the migration constructor and pup'er routine.

3.4 Performance Analysis

Because checkpoint/restart for parallel programming involves more than one processor in the parallel system, to evaluate the efficiency of checkpointing/restart we have to consider two aspects: size of data checkpointed/restored and the number of processors involved in the procedure. For both checkpoint and restart, the total overhead consists of two parts: parallel disk I/O and synchronization overhead. The first factor is related to the total size of data, and the second is affected by the number of processors in the system.

We now demonstrate the performance of checkpoint/restart mechanism on a stencil calculation (Jacobi) problem. The program updates elements in a matrix with the average value of the element and its immediate neighbors. The version we use does 1D decomposition on 2D data. The platform for experiments is NCSA Platinum IA-32 Linux cluster, which is comprised of 512 dual 1GHz Intel Pentium-III processors, with 1.5GB of RAM on each node, connected with 100Mbit Ethernet. For the sake of simplicity, we measure only the checkpoint time.

Here we show two series of data from our experiments. Figure 3.5 is the average (over 10 runs) checkpoint overhead against the total data size (ranging from 10MB to 320MB) for different number of processors (N = 8,16,32). For each fixed number of processors N, the amount of data to checkpoint is equal to the total data size divided by N. As demonstrated in the figure, the average overhead is linear to the total data size, which also implies that the synchronization overhead is

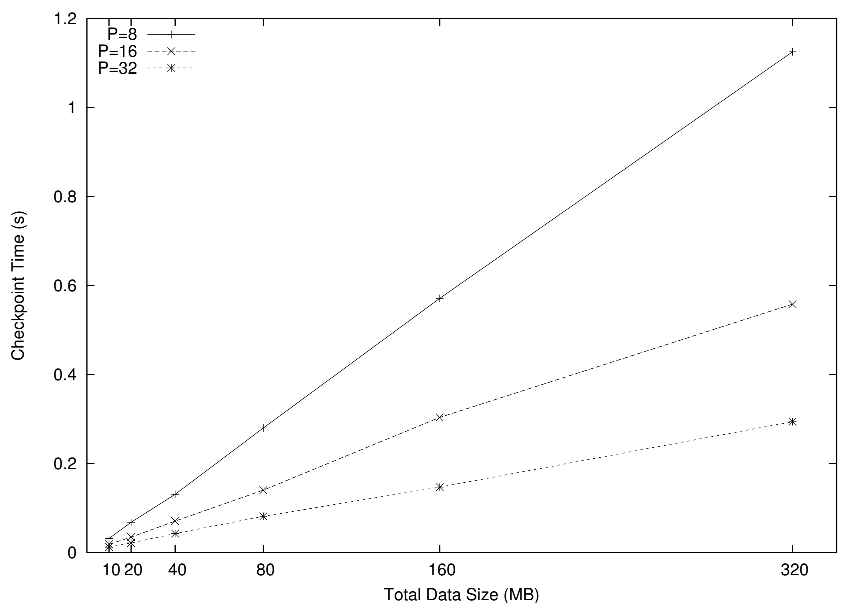

Figure 3.5: Checkpoint Overhead against Total Data Size

not a dominant factor in the checkpoint overhead.

Figure 3.6 visualizes the average checkpoint overhead against the number of processors (N = 4 - 128) for different size of data (total size = 10MB, 20MB, 100MB). In this figure, it can be observed that the overhead almost decreases linearly with the number of processors. This indicates the checkpoint mechanism scales very well. There are some exceptions in the figure too, especially when the checkpoint data is small. For example, when total checkpoint size is 10MB and N=64 and 128. In that case, the checkpoint size on each processor is too small to compare with the increased synchronization overhead. The conclusion can be drawn that the size of checkpoint should be reasonably large, especially when the total number of processor is large.

If we take a look at the bandwidth of the checkpoint, the trend is a slight decrease with increasing number of processor. For 4 nodes, the average checkpoint bandwidth is as high as 41.1 MB/s, and this metric drops to 30.3 MB/s in the case of 32 nodes. The increasing synchronization overhead for larger system contributes to this result.

For restart, the overhead is similar to the checkpoint overhead, because basically it is the reverse procedure. When the data is approximately evenly distributed on the processors, the checkpoint/restart mechanism scales well with the total data size as well as the number of processors.

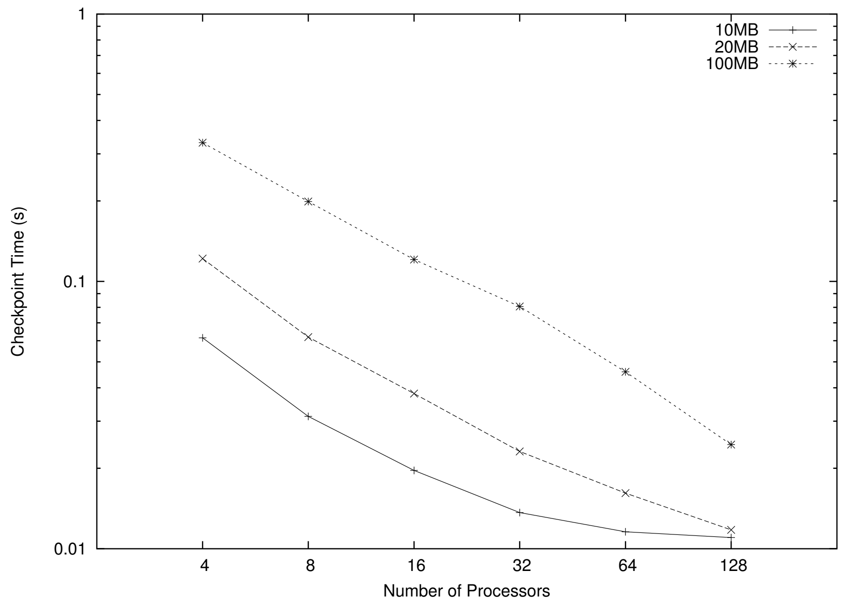

Figure 3.6: Checkpoint Overhead against Number of Processors

Chapter 4

AMPI Checkpoint/Restart Mechanism

4.1 Adaptive MPI

Adaptive MPI(AMPI)[13] is an adaptive implementation of the MPI Standard on top of Charm++. The underlying Charm++ empowers AMPI of the same collection of benefits including adaptive overlapping, automatic load balancing, as well as automatic checkpoint/restart capability.

4.1.1 MPI Standard

Message Passing Interface[11, 12] is the *de facto* standard of message passing programming. The standard specifies a set of APIs for parallel processors to communicate through message passing. It includes topics such as point-to-point communications, collective operations, one-sided communications, parallel I/O. The standard also specifies language bindings of the APIs for C, C++ and Fortran. It is important to note that MPI is the specification of a set of standard APIs, not a parallel programming language or a parallel library. There are many implementations of MPI Standard from both the industry and academia.

The MPI Standard covers a wide range of topics in parallel programming in the message passing paradigm. The Standard, however, explicitly states that a collection of topics are not included, for example, explicit shared-memory operations. Also, MPI Standard avoids anything that requires more operating system and underlying framework support, which includes checkpoint/restart mechanisms.

According to MPI Standard, MPI programmers have to take the responsibility of checkpointing/restarting their own programs if the platform does not support this feature. When the program has clear data structures holding important results, it is easy to tell what to save and how to restart. MPI programs often contain complicated structures so that it would be difficult to decide what exactly to save and how to save them. Thanks to the run-time support from underlying Charm++ system, AMPI provides automatic on-disk checkpoint/restart capability.

4.1.2 AMPI Implementation

AMPI is built on Charm++, and uses its communication facilities and threading model. As illustrated in Figure 4.1, AMPI takes advantage of the concept of virtualization introduced by Charm++. An AMPI program is based on Virtual Processors (VPs), several of which can be mapped onto one physical processor. Virtualization enables the same powerful features as Charm++, including automatic load balancing and adaptive overlapping.

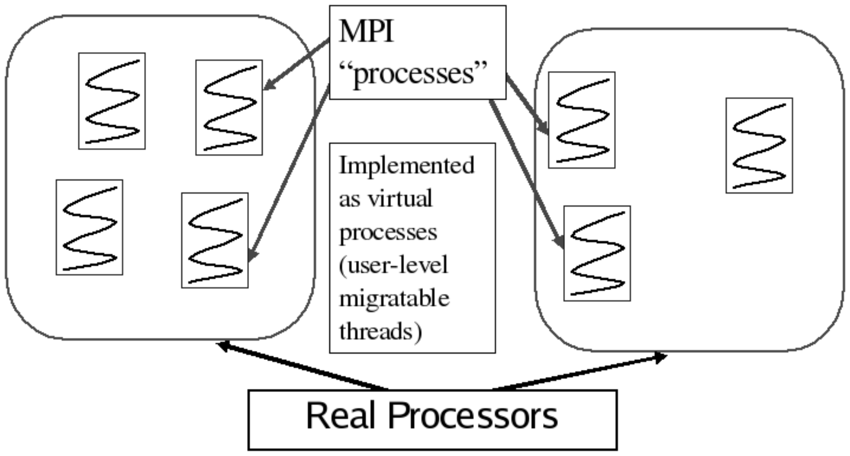
MPI "processes"
Implemented as virtual processes (user-level migratable threads)
Real Processors

Figure 4.1: AMPI Implementation

An MPI process is implemented as a user-level thread, several of which can be mapped onto one single physical processor. This virtualization enables several powerful features including automatic load balancing and adaptive overlapping. (Figure taken from [13])

AMPI implements its MPI "processors" as Charm++ "user-level" threads bound to Charm++ communicating objects. Message passing between AMPI virtual processors is implemented as communication among these Charm++ objects, and the underlying messages are handled by the Charm++ run-time system. The threads used by AMPI are light-weight user-level threads in

Charm++, and this thread module is called TCharm.

Threaded Charm(TCharm)[19] was created to support several frameworks that are based on Charm++ run-time and share the common feature of using load-balanced, migratable threads. It provides threads facilities including suspending, resuming, and migrating a thread. For example, if a VP blocks with a receive operation, the corresponding TCharm thread in AMPI would suspend itself. As the CPU is yielded to other threads while this one is suspended waiting, the overall efficiency is improved. These threads are light-weight, with a context switch time of less than 1 microsecond[13].

The migratability of TCharm threads is the key feature that makes checkpointing AMPI programs possible. When a thread migrates, its states and data in the stack and heap are brought along. There arises the problem of pointer reference. Because most of the pointer references use absolute virtual memory address rather than relative displacement, the application would lose the correct information after being migrated to a new processor as the memory locations are all changed. This is solved by a technique called "isomalloc"[1]. The basic idea is to preserve a range of virtual address for all the processors, so that after migration, the threads will have exactly the same address.

In practice, an MPI program starts with a specified number of TCharm threads, and then two chare arrays of the same size are created: `ampiParent` and `ampi`. The array `ampiParent` has one element for each AMPI VP to hold the VP-specific data, while the array `ampi` is responsible for the communications. Obviously, `ampi` is playing the role of communicator in the MPI Standard. Therefore, when the program creates new communicators, new arrays of the same class `ampi` are created. No matter how many arrays are created, they are all bound together, meaning the corresponding elements always migrate together. Typically in an AMPI program, there will be elements of multiple `ampi` arrays corresponding to one location, but to one location there is always only one element of `ampiParent` array and one element of TCharm array. For this reason, `ampiParent` is responsible for the overall control of the bound elements of the location.

4.2 Checkpoint/Restart for AMPI

Since an AMPI program is in essence a Charm++ program, checkpointing an AMPI program is no more difficult than checkpointing a Charm++ program. The only issue is the threads. Currently, a TCharm thread can be saved only when it is suspended. How can a thread suspend itself and then checkpoint itself? The answer is asynchronous remote invocation. The extension function `MPI_Checkpoint` starts with `ampiParent`. Every element of `ampiParent` will suspend its bounded thread, but right before it does that, a callback is sent out like a homing pigeon. Due to the atomicity of Charm++ entry methods, the callback will not be invoked until the thread is suspended. After the threads are suspended and program control flow is paused, the callback invokes the `ampiParent::Checkpoint` entry method in which another callback devised for the program to re-enter to user-code after checkpoint/restart is passed into the `CkStartCheckpoint` call, and the program is checkpointed like a Charm++ program.

At restart, the reverse procedure is taken. The arrays are restored and the threads resumed, and the program is restarted from the saved callback.

A TCharm thread's PUP'er routine writes out the thread's whole stack and its heap allocated data, and therefore the checkpoint data size is usually larger than the amount of data allocated in AMPI programs. To minimize the overhead, the programmer can set the stack size by a run-time option `+stacksize SIZE` to fit the actual size of data used in the application.

4.3 Performance Analysis

The performance of AMPI checkpoint/restart should also be similar to that of Charm++'s. The only difference is due to the complication of AMPI implementation. As described in the previous section, before an AMPI Virtual Processor can be checkpointed, the underlying user-level thread has to be suspended. This added overhead might decrease the efficiency.

Here we show the performance of AMPI checkpoint/restart mechanism on an MPI version of the same stencil calculation (Jacobi) problem. This program performs 3D decomposition on 3D data, and naturally it needs k-cubed number of processors. Thanks to the virtualization in AMPI, we are able to run the experiments on any desired number of processors. The experiment was carried out on the same platform: NCSA Platinum IA-32 Linux cluster with 512 dual 1GHz Intel Pentium-III processors and with 1.5GB of RAM on each node, connected with 100Mbit Ethernet.

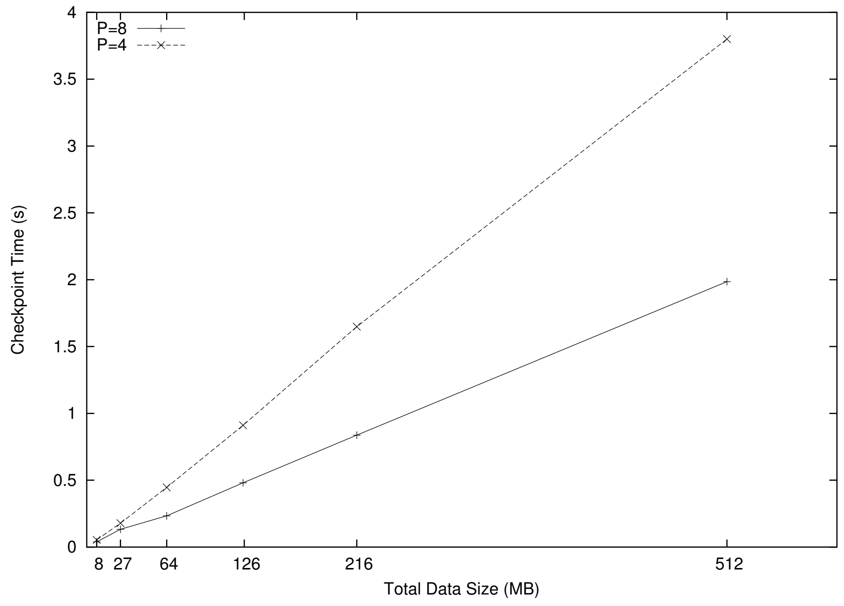

Figure 4.2: Checkpoint Overhead against Total Data Size for AMPI

Data shown in Figure 4.2 is the average checkpoint overhead against the total data size (from 8 - 512 MB) for different number of processors (N = 4,8). For each fixed number of processors N, the amount of data to checkpoint is equal to the total data size divided by N. Similar to that of Charm++, the average overhead of AMPI checkpointing is approximately proportional to the size of data to checkpoint. The average bandwidth, on the other hand, is lower than that of Charm++. As a comparison, on 4 processors, the average I/O bandwidth for Charm++ is 41.1 MB/s and AMPI 35.2 MB/s. On 8 processors Charm++ hits 36.7 MB/s and AMPI scores 30.8 MB/s. On average, AMPI checkpoint/restart has a 10% to 15% lower average I/O bandwidth.

Chapter 5

Conclusions and Future Work

We have demonstrated the On-Disk Checkpoint/Restart mechanism for the Charm++/AMPI parallel run-time system, its motivation, design and implementation. This approach is advantageous in its high reliability and versatility. It not only is a fault-tolerant effort, but also can be used for improving parallel job scheduling.

This piece of work is the first step in the effort to improve the fault tolerance of parallel programs. It has been proven useful in practice, and more importantly, it has been and will be helpful to new projects on this subject. The disk file based checkpoint/restart mechanism, along with other fault-tolerance mechanisms, will eventually form a versatile fault-tolerance framework for parallel programming in Charm++/AMPI.

5.1 Discussion and Future Work

The checkpoint/restart mechanism described in this thesis makes it possible for the parallel program to restart after an infrastructure failure. The system can further avoid any down-time by detecting the failed node and resurrect it without restarting the whole program. This requires more work on detecting failures and maintaining consistent state among processors. Although this approach is subject to potential problems like cascading rollback as described in Chapter 2, it improves effeciency by restoring only those nodes that really need to be restored. There are many such protocol developed in the parallel programming area, and one is under investigation for Charm++/AMPI framework. Research work on this topic is underway, and the preliminary results are presented in [4].

In this thesis, our checkpoint is based on disk files. There are choices of disks in a parallel system. Local disk might be the fastest, but as the node fails, it is very probable that the local disk on the failed node becomes inaccessible, rendering the checkpoint effort useless. Network file systems, on the other hand, can be considerably slower than local disks. For instance, experiments show that using NFS in Charm++ checkpointing is 5 to 10 times slower than writing local disks. Another concern is the reliability of the NFS server. Distributed file systems like RAID can be used to improve the storage reliability and I/O throughput, especially when the optimization techniques such as striping and staggering [14, 25] are adopted.

As discussed in the first Chapter, an alternative can be to write checkpoint into memory instead of disk files. This is especially feasible in parallel programming because we have access to multiple nodes in the system. Peer relationships can be formed and checkpoints deposited to peer's memory. Memory bandwidth is usually much larger than disk I/O bandwidth, and the possibility of a failed disk module can be removed from our consideration of system reliability. Recent work of [27] has proven this an effective and efficient approach.

Finally, checkpoint/restart mechanism should be flexible enough to suit the user's needs. For instance, the user should have better control of what portion of data and states are to be checkpointed and which part of the system is worth more frequent checkpointing. This not only improves the usability of the programming tool, but also boosts the performance and efficiency while saving resources. Another improvement can be automatic detection of failure and automatic restart of the program. The project described in this thesis includes some preliminary efforts, and more work on this direction is necessary in the future.

References

[1] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[2] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djailali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of SuperComputing 2002*, Baltimore USA, November 2002.

[3] Aurlien Bouteiller, Franck Cappello, Thomas Hrault, Graud Krawezik, Pierre Lemarinier, and Frdric Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on the pessimistic sender based message logging. In *Proceedings of SuperComputing 2003*, Phoenix, USA, November 2003.

[4] Sayantan Chakravorty and L. V. Kale. A fault tolerant protocol for massively parallel machines. In *FTPDS Workshop for IPDPS 2004*. IEEE Press, 2004.

[5] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 3(1):63–75, February 1985.

[6] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, November 1997.

[7] Laboratoire de Recherche en Informatique. Mpi implementation for volatile resources. http://www.lri.fr/ gk/MPICH-V/.

[8] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CHARM (4.5) programming language manual*, 1997.

[9] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart, 2002.

[10] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling: Issues and approaches. In Dror G. Feitelson and Larry Rudolph, editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 1–18. Springer, 1995.

[11] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994. http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[12] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[13] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[14] Hai Jin and Kai Hwang. Distributed checkpointing on clusters with dynamic striping and staggering. In *7th Asian Computing Science Conference (ASIAN'02)*, Hanoi, Vietnam, December 2002.

[15] L. V. Kale, 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.

[16] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[17] Robert Brunner L. V. Kale, Milind Bhandarkar and Joshua Yelon. Multiparadigm, Multilingual Interoperability: Experience with Converse. In *Proceedings of 2nd Workshop on Runtime Systems for Parallel Programming (RTSPP) Orlando, Florida - USA*, Lecture Notes in Computer Science, March 1998.

[18] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of ACM 2001 Java Grande/ISCOPE Conference*, pages 21–29, Stanford, CA, Jun 2001.

[19] Orion Lawlor. Threaded charm++ manual. http://charm.cs.uiuc.edu/ppl_manuals/html/tcharm/manual.html.

[20] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[21] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under unix. Technical Report UT-CS-94-242, 1994.

[22] B Randell. System structure for software fault tolerance. In *IEEE Transactions on Software Engineering*, pages SE1:220–232, June 1975.

[23] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[24] Georg Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 526–531. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[25] N. H. Vaidya. Staggered consistent checkpointing. In *IEEE Transactions on Parallel and Distributed Sytems*, volume 10, pages 694–702, 1999.

[26] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(5):546–554, 1995.

[27] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. Ftc-charm++: An in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. Technical Report 04-06, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2004.