DEBUGGING SUPPORT FOR CHARM++

BY

RASHMI JYOTHI

B.E., Bangalore University, 2000

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, 2003

Urbana, Illinois

Abstract

The approaches adopted so far for debugging programs written in Charm++, a data driven parallel programming language, have been the traditional logging method via *printf* statements and the "*++debug*" command-line option which runs the parallel program attaching an instance of a standard sequential debugger like *gdb* to every individual process of the parallel program. This thesis presents the implementation of a parallel debugger for Charm++ which extends the existing debugging support for Charm++ and among its functionalities includes the capability to set breakpoints, examine variables, objects and messages in queues across the parallel set-up. Also outlined is a simple record and replay mechanism for Charm++ to replay message execution in a recorded order of occurrence. This approach is extremely useful in debugging a parallel program when a bug manifests itself because of an unusual ordering of events.

To the ones closest to my heart.

Acknowledgments

As my advisor, Professor Laxmikant Kale has been a constant source of encouragement and support. His unabated enthusiasm and passion for the subject has always inspired me. I consider myself fortunate to have associated with him and thank him profusely for his guidance during the entire period I worked for him.

I am extremely grateful to Orion Lawlor and Gengbin Zheng, senior graduate students at the Parallel Programming Laboratory, for all the help, leads and suggestions they offered to me during the course of my work. My work would have been a whole lot tougher without them. I specially thank Orion for his implementation of debugging features in Charm and for the pack/unpack library, which my work is strongly based on. I also thank Eric Bohm, research programmer at the Parallel Programming Laboratory for proofreading my thesis and giving me valuable feedback.

I thank all my colleagues at the Parallel Programming Laboratory for being the amazing people they are and making my stay an invaluable learning experience and a highly memorable one. I owe a lot to my friends near and far, for all the help and support they have extended to me at various points in my life.

I could not be thankful enough for the family I have been blessed with. None of my endeavors would have been successful without them.

Table of Contents

List of Figures

Chapter 1

Introduction

Parallel programming introduces many additional challenges with respect to program correctness, robustness and reliability. The challenges faced during the design of efficient debugging tools for a parallel program include portability of such tools and examining the dynamic state information of a parallel program which is much more than a sequential program [2, 9, 10].

Traditional sequential debuggers offer state information by allowing the programmer to set breakpoints and view contents of variables and thereby examine the nature of the program's execution. These sequential debuggers are helpful in debugging the individual processes in a parallel program. However, this approach does not allow the user to examine the execution of the parallel program across the parallel machine as if it were a single program executing, which in effect it is. The other sequential debugging method often used is the traditional method of inserting output statements like `printf` in the program that output specific variables or reflect the section of the code executing. This method requires no tools or additional software and is quite reliable. Nevertheless, the programmer must decide in advance which variables to print and where to insert the output statements. Besides, incorporating new statements translates to editing and compiling the program over again. Also, scouring through large logs of output statements to gather required information can become quite tiresome. In the case of a parallel program, the events may not be written to the buffer in the order of execution due to reasons like disparate processor speeds.

The means of using a sequential debugger for debugging a parallel program is resorted to in Charm++ [4, 5, 7] in the form of the command-line run-time option "*++debug*", where each process of the parallel program running on a processor element (pe) is debugged in a separate window using a sequential debugger like *gdb* or *dbx*. The approach of logging using `CkPrintf` statements is used for debugging more often than not in Charm++. `CkPrintf` is the parallelized version of `printf`. If the command-line parameter "*+syncprint*" is passed to a Charm++ program, the CkPrintf actually blocks until the output is queued, allowing the logging to happen in causal order, at the cost of dramatically slowing down the output.

It is seen that there is a need for a parallel debugger for Charm++ that overcomes the shortcomings of employing sequential debugging techniques in conjunction with a parallel program. The primary goal of such a debugger should be to provide an integrated debugging environment which allows the programmer to examine the changing state of the parallel program as it executes. This thesis is an effort in this direction and describes the implementation of a parallel debugger for Charm++.

1.1 Motivation

The typical functionalities offered by sequential debugging schemes are cyclic interactive debugging, setting breakpoints, memory dumps and tracing. These techniques have limited applicability to the parallel programming scene because of the inherent non-deterministic and non-repeatable nature of a parallel program, the difficulty in examining the system state of the parallel program in a global sense and the fact that the debugging process could perturb the program so as to significantly change its behavior.

One approach to debugging in a parallel environment is through a collection of instances of sequential debuggers, each attached to a constituent process in the parallel program. Although this method of debugging is very useful in debugging a parallel program, programmers usually need more information to understand the behavior of their code. The state of a sequential program includes contents of all its variables and registers. A parallel program in addition contains data that is shared and in transit between processors. There is a need for a way to examine this data in whatever form the parallel programming model presents it.

Nondeterminism of a parallel program adds to the complexity of the debugging process. Certain bugs manifest themselves only due to a specific ordering of messages and may not show up in a re-run of the program due to a different ordering of messages in the re-run [15, 1]. One way to reproduce such a bug deterministically is to record the states of the parallel program and replay the execution using the recorded data.

1.2 Parallel Debugger Objectives

The objectives for the parallel debugger have been identified as follows.

- Making it possible for the programmer to freeze/unfreeze the execution of the program
- Provision to set and remove breakpoints at the entry point [11] level in a charm program.
- Providing means to view the contents of entities like array elements and messages in queues across the parallel set-up [14] during program execution.
- A flexible way of attaching specified processes of the parallel program to the sequential debugger, during the course of program execution.

  This is specially useful when the number of processor elements is high and sequential debugging is sought on specific processors. In such a situation, using the runtime "*++debug*" option is cumbersome as it becomes very difficult to keep track of the large number of *xterm* windows opened at the onset of execution for each instance of the sequential debugger attached to every process in the parallel program.

- Incorporating a record-replay mechanism into Charm, for reproducing bugs which happen once in a while depending on the order in which messages are processed.

  A *+record* option provided at runtime to a Charm program writes a trace to a file, which is later read when the *+replay* runtime option is provided to ensure messages are processed in the same order as the recorded run.

1.3 Thesis Organization

The thesis consists of 6 chapters. Chapter 2 talks about the programming model and concepts of Charm++. Chapter 3 gives an overview about the Converse Client Server (CCS) module and the PUP framework in Charm++, which are used in the implementation of the debugger. Chapter 4 elaborates the design of the newly added debugging features. The record and replay mechanism is presented in Chapter 5. Conclusion and possible future work are outlined in Chapter 6.

Chapter 2

Charm++: An Overview

Charm++ is an object oriented portable parallel programming language based on C++. In other words, it is a parallel language consisting of C++ with a few extensions. It provides a clear separation between sequential and parallel objects. The execution model of Charm++ is message-driven [7].

2.1 Charm++ Design

Portability is a key requirement for large scale development of parallel software. Charm++ programs are portable and run without change on all MIMD machines. The system supports dynamic load balancing strategies which means that dynamic creation of parallel work is allowed. Dynamic load balancing is necessary when there are irregular parallel computations and load is unevenly distributed among the processor elements [4, 5].

Charm++ programs specify parallel processes called *chares* [11]. Chares can create new chares and can send messages to each other. Message-driven execution is employed in Charm++ instead of the traditional send/receive based communication. In such a message-driven execution model all computations are initiated in response to messages being received. The system calls in Charm++ are non-blocking and therefore, asynchronous. Charm++ entities can contain private data and public methods like regular C++ objects. The significant difference is that these methods can be invoked from remote processors asynchronously.

Asynchronous method invocation implies that the caller does not wait for the method to be executed or in other words does not wait for the method to return a value. Such a method that can be invoked remotely in Charm++ is called an *entry method* or an *entry point*. Entry methods do not have a return value. An entry method is always a part of a chare - there are no global entry methods in Charm++.

2.1.1 Language Entities

Charm++ consists of the following categories of objects

- Sequential objects

  Sequential objects are ordinary sequential C++ objects. Such entities are accessible locally and are not known to the Charm++ runtime system. Charm++ does not affect the syntax or semantics of such C++ entities.

- Concurrent objects

  Chares are the concurrent objects in a Charm++ program. Syntactically, they are instances of C++ classes that are derived from a system-provided class called `Chare`. In addition to the private and public data and methods of a usual C++ object, they contain entry methods which are asynchronous methods that can be invoked remotely. Chares are different from C++ objects because they can be created asynchronously from remote processors. They are accessed using a proxy or a handle. Chares and their entry methods have to be specified in an interface file.

- Replicated objects

  These objects consist of a branch on every processor.

  – Chare Arrays

    A *Chare Array* [8] is a collection of chares and the size of the array is not constrained by the underlying parallel machine such as number of processors or nodes.

Therefore, a chare array can have any number of elements. Each array element of a chare array has a globally unique index and messages are addressed to that index. The dynamic load balancing framework which kicks off when a Charm++ program starts, treats array elements as objects that can be migrated across processors.

  - Chare Groups

    A *Chare Group* is a collection of chares with one representative on each processor. All members of the chare group share a globally unique name.

  - Chare Nodegroups

    A *Chare Nodegroup* is very similar to a Chare Group except that instead of having one group member on each processor, the nodegroup has one member on each shared memory multiprocessor node.

- Shared objects

  Charm++ does not allow global variables for keeping programs portable across a wide range of machines. *Read-only* variables provide a mechanism for sharing data among all objects. They are used to share information that is obtained only after the program begins execution and does not change after they are initialized in the dynamic scope of the *main* function. Just like global variables, they can be accessed from any chare on any processor.

- Messages

  Messages are entities that are used for communication between concurrent objects. Messages supply data arguments to the asynchronous remote method invocation. With parameter marshalling [11], the Charm++ runtime creates and handles messages completely internally. Another variation of communication objects is conditionally packed and unpacked. This variation should be used when one wants to send messages that contain pointers to the data rather than actual data to other processors.

2.1.2 Execution Model

The basic unit of parallel computation in Charm++ programs is the chare, which can be created on any available processor and the methods of which can be invoked from remote processors. A Charm++ program consists of a number of Charm++ objects distributed across the available number of processors or processor elements. Chares are created dynamically and they invoke methods on one another asynchronously.

The runtime system of Charm++ or the *Charm Kernel* maintains a pool consisting of seeds for new chares and messages for existing chares. As mentioned previously, entry methods are methods of a chare that can be remotely invoked. Entry methods take marshalled parameters or a pointer to a message object. Since chares can be created on remote processors, they require a minimum of one constructor that is an entry method. Entry methods cannot be preempted and therefore once an entry method is entered execution is guaranteed to proceed till the method is done without interruption.

While creating a chare [11], the location does not have to be specified in terms of processor number. The Charm Kernel will place the chare on a least loaded processor. Thus, Charm++ provides dynamic seed-based load balancing. Chares can potentially migrate from one processor to another. Chare-arrays, Chare-groups and Chare-nodegroups are important parallel structures provided by Charm++. An array is a collection of chares, indexed by some index type and mapped to processors according to a user-defined map group. A Charm++ program must have at least one *mainchare*. When the Charm++ program starts up, the mainchare is created on the processor 0. Execution of the program is triggered once the mainchares have been constructed. Usually, mainchare constructor initiates computation by creating arrays, other chares and groups. Readonly data is also initialized in the mainchare.

Remote chares communicate with each other via the asynchronous entry methods and this is the only mode of communication between processors. Therefore, the Charm Kernel

```
void _call_foo_func(void *msg, foo *obj)
 {
    obj->func(msg);
 }

...at startup...

int funcEpIdx=CkRegisterEp(&_call_foo_func,...);
```

Figure 2.1: Snippet of registration code generated at startup

needs to know the type of chares in the user program and the remotely invocable entry methods along with the parameters they take as input. These user-defined entities need to be registered with the Charm Kernel when the program starts up. The Kernel assigns a unique identifier to each of them. All this registration is automatically taken care of by the Charm++ interface translator.

At startup of a Charm++ program, each module registers its methods which end up in an entry method table on every processor on which the program runs. The entry method table on each processor should be consistent for a particular program. An entry point's index in this table is used across the system to refer to that particular entry point. The Charm++ interface translator reads the interface file, the *.ci* file and automatically writes this registration code to the *.def* file. For instance, suppose the programmer needs to be able to call some C++ method `func` remotely on user object of type `foo`. To make this visible to the Charm Kernel, a function pointer to a "`call-function`" corresponding to the entry method `func` is registered in the entry table. Figure 2.1 shows a snippet the registration code generated by the interface translator to achieve this.

The interface translator similarly generates definitions for *proxy* objects. A proxy object acts as a handle to a remote chare. Method invocation on a proxy object translates to remote method invocation on the chare. A Charm++ program can be terminated by the `CkExit` call. `CkExit` need not be called on all processors, it is sufficient to call it from just one processor at the end of the computation.

2.1.3 Machine Layer: A Brief Overview

Charm++ is in essence a simple, thin wrapper on Converse. Converse [3] is a framework for parallel programming that supports multi-lingual interoperability. It extracts the essential support of runtime support into a common core so that language specific code (for instance, Charm++) does not have to pay overhead for features that it does not need. Converse treats the parallel machine as a collection of nodes that communicate primarily via messages. Each node is comprised of a number of processors that share memory. In some cases, the number of processors per node may be exactly one. The processors may have multiple threads running on them which share code and data but have different stacks. Each processor runs a scheduler which is responsible for all message reception.

When a message arrives at a processor it triggers the execution of a handler function[12]. The handler function receives as an argument a pointer to the message. The message itself specifies which handler function to be called when the message arrives. The message is a contiguous sequence of bytes and has two parts - the header and the data. The header contains a handler number which specifies which handler function is to be executed when the message arrives. Converse maintains a table mapping handler numbers to function pointers. Each processor has its own copy of the mapping.

Communication primitives insert messages into the scheduler queues at remote processors, where the scheduler thread finds them and processes them. The Converse scheduler serves not only as a message receiver but also as a central allocator of CPU time. There are two kinds of messages in the system waiting to be scheduled - messages that have come from the network and those that are locally generated. The scheduler's job is to repeatedly deliver these messages to their respective handlers.

2.2 Virtualization Concept

Charm++ is a parallel programming model based on the concept of virtualization , where the programmer divides the work into a large number of chunks, and lets the system map these entities to processors. The number of parts a computation is broken into is typically independent of the number of processors N and is more often than not larger than N. It is observed that a high degree of virtualization is favorable in most cases because of the beneficial result of smaller objects on cache performance [6]. In the case of Charm++ the entities are chares. The programmer does not refer to processors in their code but programs in terms of the interaction between the virtual entities. The Runtime system (Charm Kernel) is aware of processors and maps these virtual processors to real processors. It also has the capability to change the mapping at runtime without the user program having to specify it.

Chapter 3

Frameworks in Charm++

The implementation of the parallel debugger involved incorporating required support into the Charm++ core. The implementation of debugging support in the Charm Kernel relied heavily on the pack/unpack or PUP framework available in the core. The debugger is also modelled as a client in the Converse Client-Server (CCS) Interface provided by Charm++. This built-in features of the Charm++ core are briefly touched upon in the following sections.

3.1 The PUP Framework

The pack/unpack framework or the "PUP" framework is a generic way provided by Charm++ to describe the data in an object. It is a suite of classes that enables objects in Charm++ (for example, array elements) to migrate from one processor to another. In a nutshell, the framework provides services to any operation that requires a traversal of the object state in terms of its data members. The Charm++ system uses the generic description of an object, provided by the PUP framework, to pack the particular object into a message and later unpack the message into a new object on another processor. Thus the name, pack/unpack framework. Besides being used in transporting objects intact across processors during migration, the PUP framework can also be used to serialize an object's data to disk. The framework can also be used to retrieve an object's data in some interpretable form and this functionality is used in the implementation of the debugger.

The obvious approach for a class that needs to provide for its objects to migrate would be to implement `static pack` and `unpack` methods. If an object is required to migrate to another processor, while the program is executing, the `pack` method is invoked with a pointer to the object as an argument. The pack method allocates a memory buffer large enough to hold all the object's data and then proceeds to serialize the object's data into the memory buffer.The memory buffer is then encased in 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 constructor's 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 extract data from the packed object in the message to create the new object. At the end of the `unpack` method, migration is complete.

It is seen that most of the functionality of the `pack` and `unpack` methods is similar in nature. The `pack` function copies the data to a serial buffer in a particular order, while the `unpack` method copies the data from the serial buffer in the same order as `pack`. The PUP library avoids duplication of code by requiring the programmer of a particular class to implement just a single method, called `pup`. The `pup` method takes a single parameter, which is an instance of a class `PUP::er` (such a class is also called a *pupper*). The role of the `pup` method is to perform a traversal of the object state. The actual pack, unpack, write-to-disk, convert-to-interpretable-form operations that need to be performed on the data members are executed by the pupper class.

The following classes in the PUP framework were used in implementing debugging support in charm.

- `class PUP::er` - This class is the abstract superclass of all the other classes in the framework. 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

```
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 3.1: A simple class declaration showing the `pup` method

these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.

- `class PUP::toText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It copies the data of an object to a C string, including the terminating NULL.
- `class PUP::sizerText` - This is a subclass of the `PUP::toTextUtil` class which is a subclass of the `PUP::er` class. It returns the number of characters including the terminating NULL and is used by the `PUP::toText` object to allocate space for building the C string.

The code in Figure 3.1 shows a simple class declaration that includes a `pup` method.

3.2 Converse Client-Server Interface

The Converse Client-Server (CCS) module enables Converse [3] programs to act as parallel servers, responding to requests from non-Converse programs. The CCS module is split into two parts - client and server. The server side is used by a Converse program while the client side is used by arbitrary non-Converse programs. A CCS client accesses a running Converse program by talking to a `server-host` which receives the CCS requests and relays them to the appropriate processor. The `server-host` is `charmrun` [11] for net-versions and is the first processor for all other versions.

In the case of the net-version of Charm++, a Converse program is started as a server by running the Charm++ program using the additional runtime option "*++server*". This opens the CCS server on any TCP port number. The TCP port number can be specified using the command-line option "*server-port*". A CCS client connects to a CCS server, asks a server PE to execute a pre-registered handler and receives the response data. The function `CcsConnect` takes a pointer to a `CcsServer` as an argument and connects to the given CCS server. The functions `CcsNumNodes`, `CcsNumPes`, `CcsNodeSize` implemented as part of the client interface in Charm++ returns information about the parallel machine. The function `CcsSendRequest` takes a handler ID and the destination processor number as arguments and asks the server to execute the particular handler on the specified processor. `CcsRecvResponse` receives a response to the previous request in-place. A timeout is also specified which gives the number of seconds to wait till the function returns a 0, otherwise the number of bytes received is returned.

Once a request arrives on a CCS server socket, the CCS server runtime looks up the appropriate registered handler and calls it. If no handler is found the runtime prints a diagnostic and ignores the message. If the CCS module is disabled in the core, all CCS routines become macros returning 0. The function `CcsRegisterHandler` is used to register handlers in the CCS server. A handler ID string and a function pointer are passed as parameters. A table of strings corresponding to appropriate function pointers is created. Various built-in functions are provided which can be called from within a CCS handler. The debugger behaves as a CCS client invoking appropriate handlers which makes use of some of these functions. Some of the built-in functions are as follows.

- `CcsSendReply` - This function sends the data provided as an argument back to the client as a reply. This function can only be called from a CCS handler invoked remotely.

- `CcsDelayReply` - This call is made to allow a CCS reply to be delayed until after the handler has completed.

The CCS runtime system provides several built-in CCS handlers, which are available to any Converse program. All Charm++ programs are essentially Converse programs. `ccs_getinfo` takes an empty message and responds with information about the parallel job. Similarly the handler `ccs_killport` allows a client to be notified when a parallel run exits.

Chapter 4

Parallel Debugger

Implementing the parallel debugger required incorporating the necessary support into the charm kernel and building a friendly user interface. Features that would be useful to a programmer while debugging a parallel program were identified and implemented one by one.

4.1 Freezing and Unfreezing a Program

The debugger requires support in the Charm kernel to *freeze* a Charm++ program in execution. This involves the debugger sending a ccs message to the program which triggers the handler `CpdFreeze` on the processor element on which execution was intended to be frozen. This handler sets a flag `freezeModeFlag` to 1 and the process enters the freeze mode scheduler. As long as `freezeModeFlag` is 1, the program continues to be in the freeze mode scheduler. The freeze mode scheduler queues up all the messages that arrives on the processor element into a debug queue. Only the ccs messages are processed, keeping in mind that they could be from the debugger itself and if they were not processed, it would not be possible to resume execution of the program (which is again the result of a ccs handler). Setting the `freezeModeFlag` to 0 quits the freeze mode scheduler, before which it processes all the messages queued up in the debug queue. The ccs handler `CpdUnFreeze()` performs this functionality. Figure 4.1 shows the pseudocode for the handlers which are responsible

```
CpdFreeze()
{
  set freezeModeFlag to true;
  CpdFreezeModeScheduler();
}

CpdFreezeModeScheduler()
{
    msg = next message to be processed;
    while (freezeModeFlag = true)
    {
         if (msg is a ccs message)
               Handle msg;
         else
               Push msg on debug queue;
    }
    Process all messages in debug queue;
}

CpdUnFreeze()
{
  set freezeModeFlag to false;
}
```

Figure 4.1: Pseudocode for the handlers which take care of *freeze* and *unfreeze* functionality.

for freezing and unfreezing the program.

The programmer could opt to freeze the execution of his program at the click of a button on the debugger's user interface to examine the state of the different variables in his parallel program. Besides this, the `CpdFreeze` function is called in two scenarios - at the onset of executing a Charm++ program for debugging and when a breakpoint is reached. In the first scenario, this `CpdFreeze` function is called at the end of the `_initCharm` routine. The `_initCharm` routine runs almost the entire Charm++ setup process. It sets up the various subsystems and performs all the registration procedures by calling the various `_register` routines. A command-line argument "*+cpd*" is passed in as an argument to the program being debugged, to indicate that it is in the parallel debugging mode. When in the parallel debugging mode, the program freezes at the onset. The programmer could choose to continue

```
extern "C" void _call_foo_bar(void *msg, void *obj)
{
  fooMsg *m = (fooMsg *)msg;
  foo *f = (foo *)obj;
  f->bar{m};
}
```

Figure 4.2: A simple call function for entry point `foo::bar{fooMsg *}`

execution as it is or he/she could tinker around, by setting breakpoints.

4.2 Setting Breakpoints

The Charm++ programmer would need to set breakpoints in terms of entry points as entry points to capture the true essence of control-flow in the parallel program. Entry points are purely Charm++ entities and therefore, are not visible to the sequential debugger when the "*++debug*" option is used for debugging. Thus, it is not possible to set breakpoints for entry points using the sequential debugger. Hence it becomes essential to build in the ability to set and remove break points in terms of entry points. In the Charm Runtime System, the information about entry points is stored in a data structure `_entryTable`. `_entryTable` is basically a vector of objects of `class EntryInfo`, each object representing a single entry method or constructor. `EntryInfo` stores the function pointer to the "*call-function*" corresponding to the entry point. A *call-function* is how Charm++ actually invokes an entry method on an object. Call functions take two parameters - the message to pass to the method, the object to invoke the method on. Figure 4.2 portrays a simple call-function to invoke an entry point `foo::bar{fooMsg *}`. Call functions are even used to invoke constructors on new chares.

Information about any particular entry point can be referred to across processors by their index into the `_entryTable`. This index is referred to as the entry point's index, often abbreviated as "*epIdx*". Each processor element has a copy of its own `_entryTable`. The contents of the `_entryTable` should be consistent across processors. The debugger sets a break point for a particular entry point in a Charm++ program by sending the entry point's name packed in a ccs message to the running program. This would cause the pre-registered ccs handler `CpdSetBreakPoint` to execute.

The logic applied in setting break points is simple - The `EntryInfo` object corresponding to the entry point for which a break point is to be set, is replaced with a dummy debug `EntryInfo` object whose function pointer corresponds to the call-function `_call_freeze_on_break_point`. At the onset, this call-function saves the entry point index of the break point and the message and object with which it was invoked. Then, it freezes program execution by calling `CpdFreeze` function. The replaced `EntryInfo` information is stored in a hash table which is indexed by the entry point index. When the program continues from the break point (that is, when the ccs handler `CpdContinueFromBreakPoint` is triggered) the appropriate entry method is retrieved from the hash table using the saved entry point index. The object on which the retrieved entry point is invoked and the message that is used as a parameter are also retrieved from the previously saved information.

The pseudocode for the handlers to set and continue from a break point is shown in Figure 4.3. System-defined entry points are differentiated from user-defined entry points by the option "*intrinsic*" passed to the translator (which creates the *.def* files). If the "*intrinsic*" option is used during translation, the variable `inCharm` in an `EntryInfo` object is set to true indicating that the particular entry point is system-defined.

4.3 Introspection API

The other significant part of the parallel debugging support in Charm is the introspection API or the *CpdLists* interface (*implemented by Orion Lawlor*). The fundamental idea adopted here is to allow the debugger to retrieve and examine the entities in the parallel set-up using the pup framework. The *CpdLists* interface is used to register a list of items with pup routines and pup the items out in a readable format (in this particular implementation,

```
breakPointEntryInfo = hash table of replaced entry points
                      indexed by epIdx;

lastMessage = Pointer to a message;

lastObject =  Pointer to an object;

lastBreakPointIndex = stores epIdx of last break point reached;

_call_freeze_on_break_point(msg, obj)
{
  lastMessage = msg;
  lastObject = obj;
  lastBreakPointIndex = retrieve epIdx from msg
  freeze the program;
}

CpdSetBreakPoint(entryPointName)
{
  num = number of entries in _entryTable
  for (i = 1 to n)
  {
     entry = _entryTable[i];
     if (entry->name = entryPointName)
     {
        Insert entry into breakPointEntryInfo;
        _entryTable[i] = new entry info with function
                         pointer "_call_freeze_on_break_point";
        quit for loop;
     }
  }
}

CpdContinueFromBreakPoint()
{
   unfreeze the program;
   entry = retrieve entry point info from breakPointEntryInfo with
           index lastBreakPointIndex;
   Invoke entry on lastObject with parameter lastMessage;
}

CpdUnFreeze()
  set freezeModeFlag to false
```

Figure 4.3: Pseudocode to set a break point and continue from it.

```
_entryTable
_chareTable
_msgTable
_readonlyTable
_readonlyMsgs
_mainTable
_groupIDTable
CmiLocalQueue
CmiSchedQueue
```

Figure 4.4: Data structures in Charm++ that can be examined using *CpdLists*.

a C-style string is used). The list of items that can be examined are entries in the data structures listed in Figure 4.4.

`_entryTable`, `_chareTable`, `_msgTable`, `_readonlyTable`, `_readonlyMsgs`, `_mainTable` are linear lists consisting of registered entities. These lists are indexed by an index which is identical across processors. Each of the entries in each of these lists represents an entry method, a chare/group element/array element, a type of message, a readonly global variable, a readonly message (once was a way to get truly variable sized messages, obsolete now) or a mainchare's constructor respectively. `_groupIDTable` is a linear list of indices to access pointers to *group* objects (could be Array Managers, Nodegroups or Charegroups) from the `groupTable`. `CmiLocalQueue` is a data structure maintained at the machine layer and is a FIFO queue containing all messages from the local processor. `CsdSchedQueue` is a priority queue used to store messages from other processors.

The main component in the *CpdLists* interface is the abstract super class `CpdListAccessor`. An instantiation of class `CpdListAccessor` or a typical accessor object keeps track of one particular list, from the list of entities previously discussed. The length of the list , the path or string to identify the accessor object and a pointer to a pup function are stored as part of the object. All the accessor objects are stored in a hash table where the key is a string which is the path of the object. For instance, the accessor object for the `_chareTable` is retrieved from the hash table using the key "*charm**chare*". The registration of the accessor objects

```
cpdListTable = hash table storing accessor objects

// during Charm++ registration
// This creates an entry in the hash table cpdListTable for
// fooList
CpdListRegister(new CpdListAccessor("charm/foo", fooList.length(),
                pupFoo));
.
.

CpdList_ccs_list_items_txt (msg)
{
  path = retrieve from message;
  accessor_object = retrieve from cpdListTable using path;
  Pup out data using pup function of accessor_object;
  //If path = "charm/foo", pup function used is pupFoo
}
```

Figure 4.5: Working of *CpdLists*.

happens at the beginning of the program execution in the `_initCharm` routine. When the entities in a particular list have to be examined, a ccs handler `CpdList_ccs_list_items_txt` is invoked which retrieves the correct accessor object from the hash table (using the path passed in to the handler) and uses the pup function of the accessor object to pup out the entities as a C string to the debugger, which is displayed to the programmer appropriately. Figure 4.5 outlines the working of `CpdLists` and shows the registration of a dummy list `fooList` which is accessed using the string "*charm\foo*".

A separate ccs handler `CpdExamineArrayElement` is written to retrieve the contents of an array element on a processor. An accessor object is pre-registered for `_groupTable` and a particular array location manager could be accessed on each processor to retrieve the data of all array elements on a processor at one shot. However, this method is not used because the data retrieved can be quite overwhelming like in the case where each array element stores a huge array of floating point numbers and there exists hundreds of array elements on every processor. Therefore, the more optimal approach is to retrieve an array element's data one at a time, as and when asked for. If the user has implemented the function

```
CpdExamineArrayElement (msg)
{
  retrieve group id and array index from msg;
  retrieve appropriate array manager corresponding to group id;
  array_element = got by iterating through array manager and using
                  retrieved array index to locate the desired
                  element;
  array_element->pupCpdData();
}
```

Figure 4.6: ccs handler to examine contents of an array element.

`pupCpdData(PUP::er &)`, this function is used to pup out data to the debugger for the user to examine. The user can incorporate comments in the `pupCpdData` function to identify variables and make the information displayed by the debugger more understandable. The default behavior of `pupCpdData` is to use the pup function implemented for the array. The working of the handler `CpdExamineArrayElement` is illustrated in Figure 4.6.

4.4 Attaching Sequential Debugger

Another functionality provided by the debugger is the ability to attach specified number of individual processes in the parallel program to instances of a sequential debugger during the execution of the program. This would allow the programmer to selectively debug on specific processors during the course of execution. The current implementation of this functionality is for the *net-linux* version of Charm++. For instance, if the parallel program is being run on 10 processors, the programmer can choose to open *gdb* instances on processors 2, 3 and 4, go about debugging, close the windows and then later on in the program execution, perform sequential debugging using *gdb* on the process on processor 7. This is achieved by forking off a child process on the desired processor which executes a script to bring up an instance of *gdb* which attaches itself to the process id of the parent process which is the individual process of the parallel program on that processor.

4.5 User Interface and Instrumentation

The user interface for the Parallel debugger is modelled as a CCS client and is implemented in *Java*. The debugger performs the various actions like setting/removing break points, freezing/unfreezing the execution, examining entities in the parallel program, attaching specific processes to instances of a sequential debugger by sending ccs messages to the Charm Runtime system. These messages trigger the invocation of appropriate pre-registered handlers, which perform the required actions.

The Charm++ programmer starts the debugger Java client running the following command

$>$ java ParDebug [[-file $< charmprogramname >$] [[-param "$< charmprogramparameters >$"][-pes $< numberofpes >$]]]

The program to be debugged and the parameters passed to the program including the number of processor elements the program should run on are passed as command-line arguments to the debugger client. Alternatively, they can be set via a menu on the debugger GUI. This is shown in Figure 4.7.

Once the debugger's GUI loads the programmer triggers the program execution by clicking the *Start* button. The program starts off displaying the user and system entry points as a list of check boxes, freezing at the onset. The user could choose to set breakpoints by clicking on the corresponding entry points and kick off execution by clicking the *Continue* Button. Figure 4.8 shows a snapshot of the debugger when a breakpoint is reached. The program freezes when a breakpoint is reached.

Clicking the *Freeze* button during the execution of the program freezes execution, while *Continue* button resumes execution. *Quit* button can be used to suspend execution at any point of time. Entities (for instance, array elements) and their contents on any processor

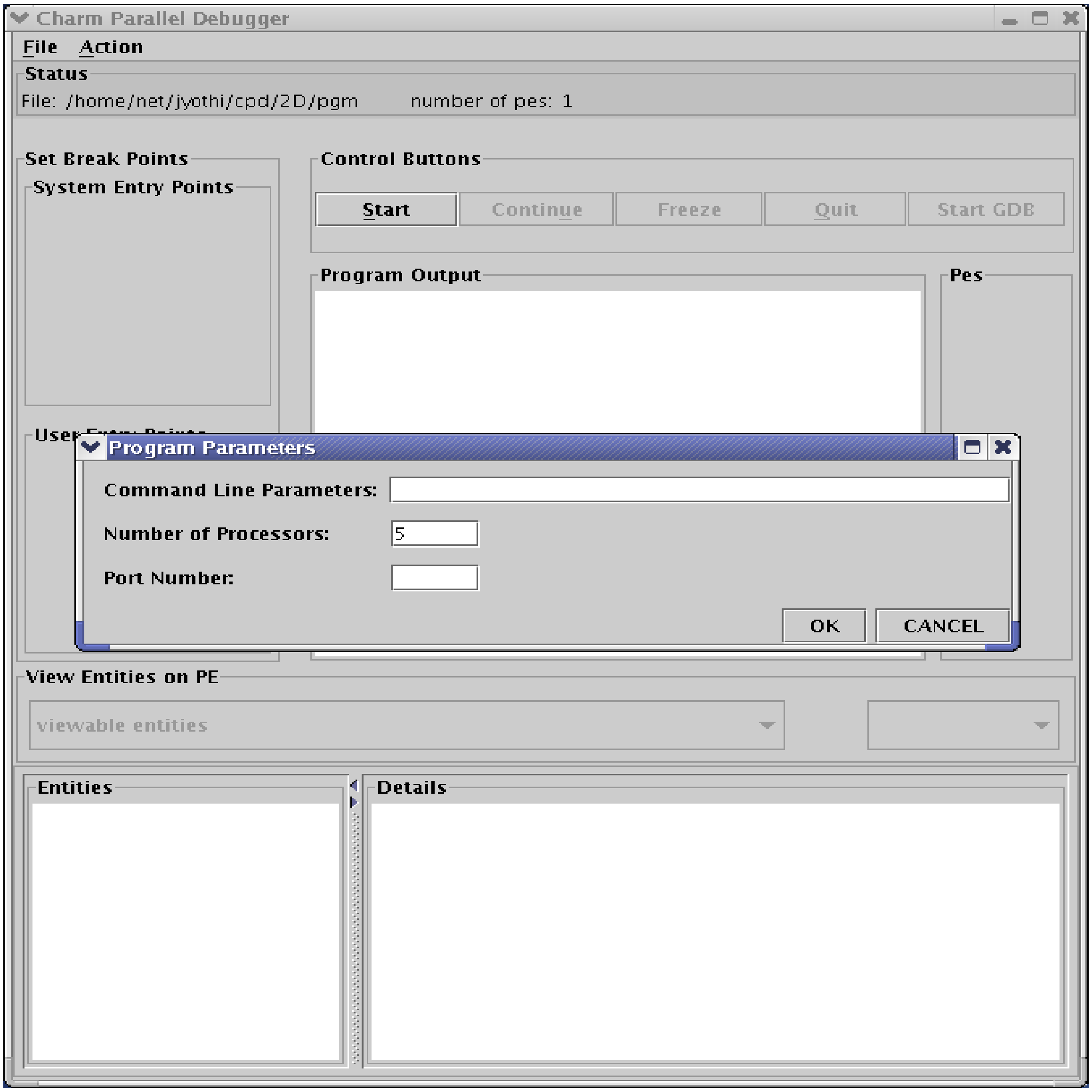

Figure 4.7: Using the menu to set parameters for the Charm++ program

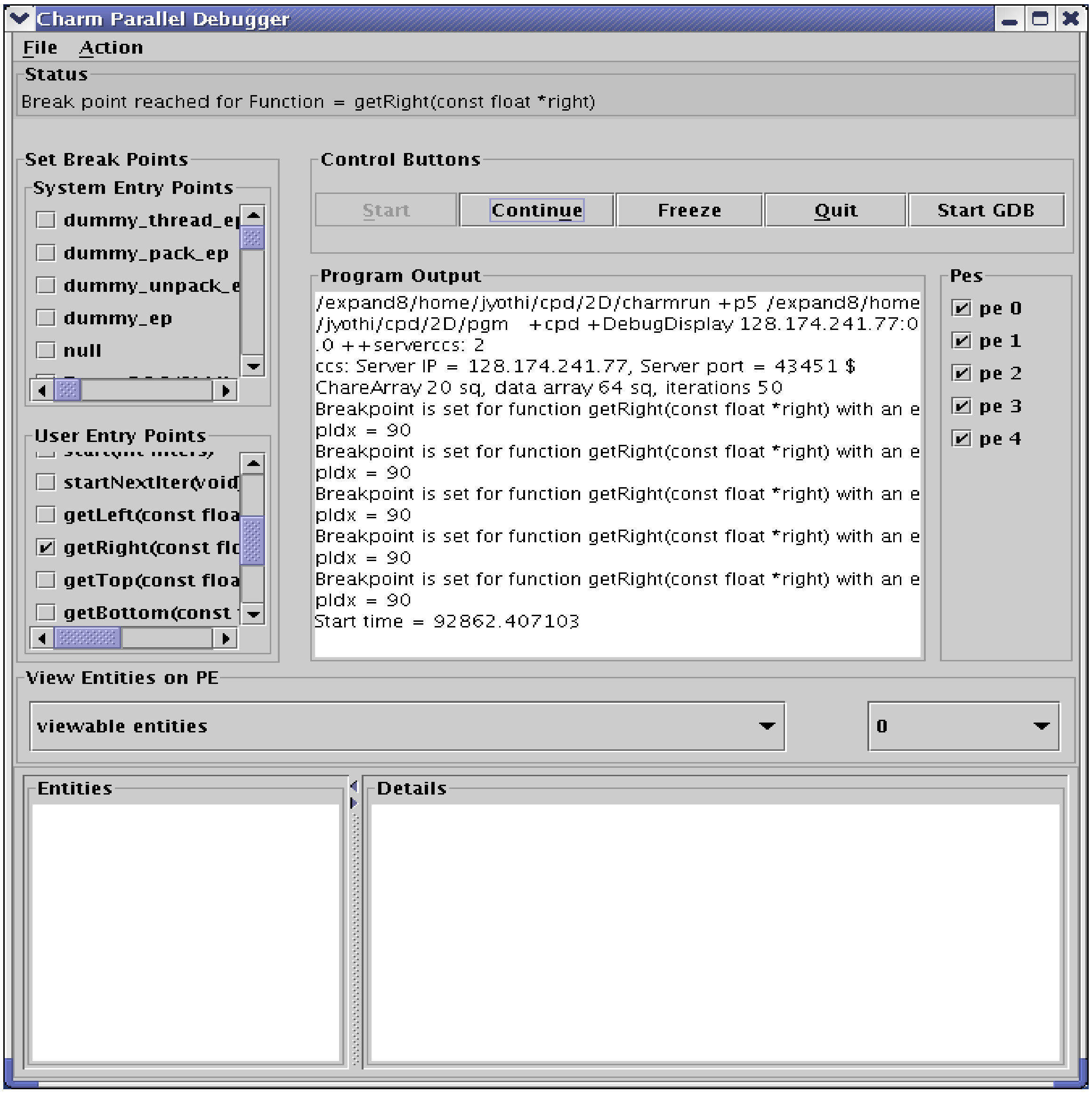

Figure 4.8: Parallel debugger when a break point is reached

can be viewed at any point in time during execution as illustrated in Figure 4.9.

Specific individual processes of the Charm++ program can be attached to instances of *gdb* as shown in Figure 4.10. Like mentioned previously, the programmer can implement the function `pupCpdData(PUP::er &)` for an array element and thereby control the information displayed by the debugger by choosing the data to be displayed and by inserting appropriate comments. An example is illustrated in Figure 4.11.

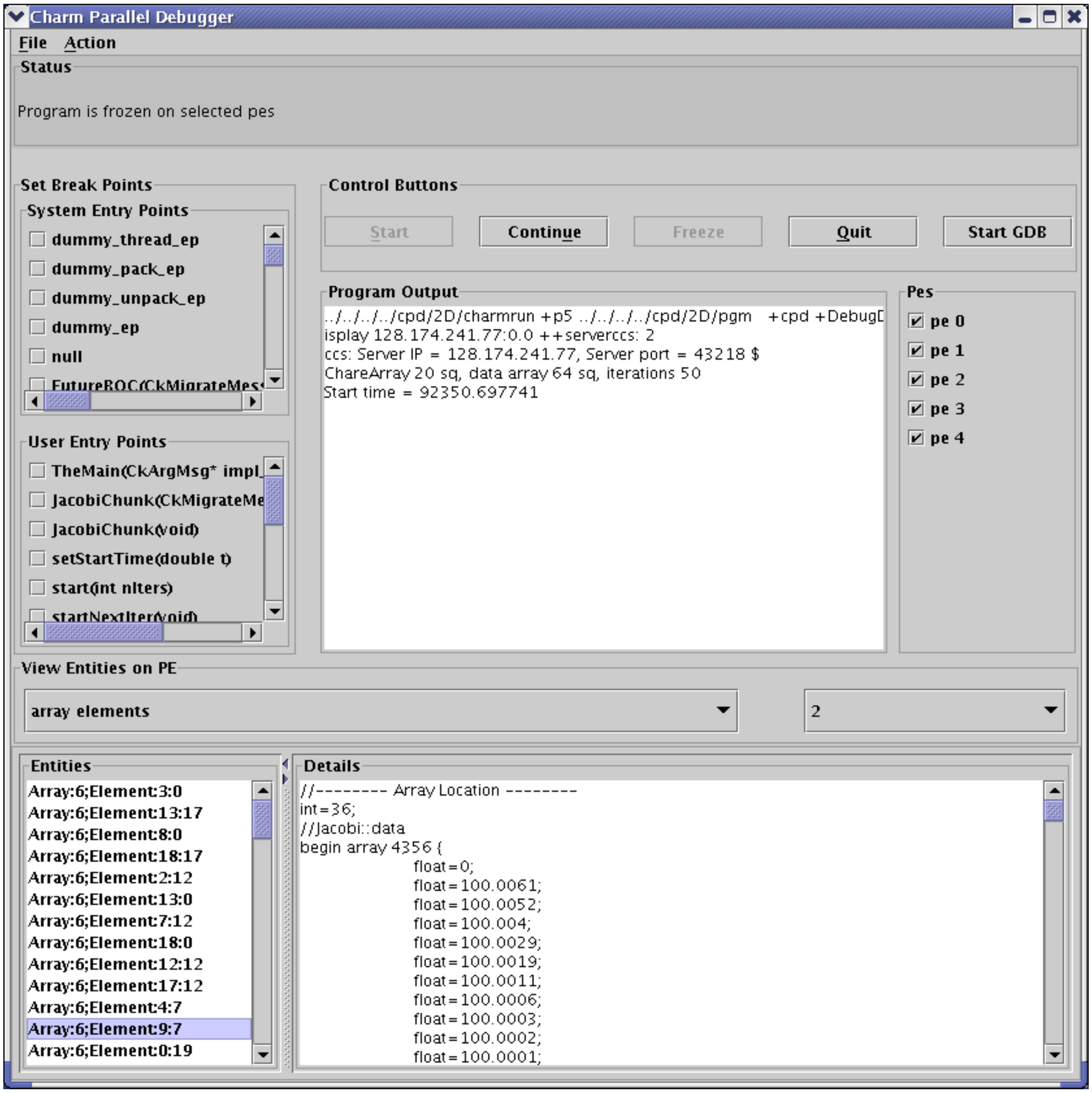

Figure 4.9: Freezing program execution and viewing the contents of an array element using the Parallel Debugger

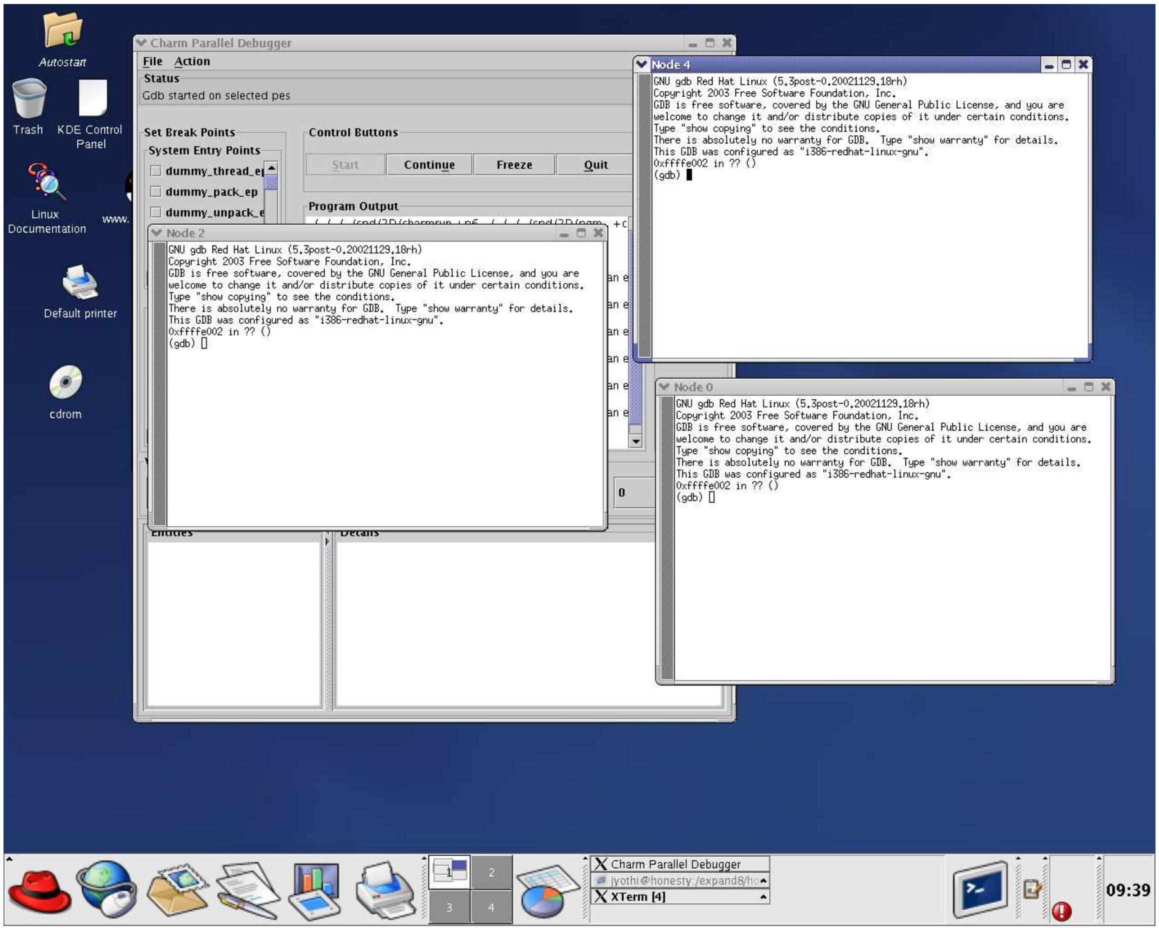

Figure 4.10: Parallel debugger showing instances of *gdb* open for the selected processor elements

```
//MyArray is a chare array where each array element has a member
//variable data, which is an integer array

void MyArray::pupCpdData(PUP::er &p) {
  p.comment("contents of integer array: data");
  p|data;
}
```

Figure 4.11: A sample user implementation of `pupCpdData`

Chapter 5

Record and Replay

Record and Replay is a mechanism used to detect bugs that happen only once in a while depending on the order in which messages are processed. The program in consideration is first run in a record mode which produces a trace. When the program is run in replay mode it uses a previous trace got from a record run to ensure that messages are processed in the same order as the recorded run. The idea is to make use of a message-sequence number and a theorem says that the serial numbers will be the same if the messages are processed in the same order [15].

5.1 Related Work

Instant replay [1] was introduced as a mechanism to debug the asynchronous behavior of a parallel program. In a parallel program's execution a bug can manifest itself because of an unusual ordering of events. The bug may not recur if the experiment is repeated under the control of a debugger, because the debugger may alter the original ordering of events. In such cases it would be helpful to be able to deterministically reproduce the bug. Instant replay is not dependent on the particular form of interprocess communication used. Replay is provided for an entire program rather than individual processes in isolation. It avoids global synchronization of events through the use of a fully distributed protocol. There is no centralized bottleneck and no need for synchronized clocks or a globally consistent logical time.

Instant replay models all process interactions in a parallel program as operations on shared data. This is not restrictive since all communication and synchronization primitives can be reduced to operations on shared data. Message passing can be modelled as communication through a shared port, mailbox or memory segment. Instant replay requires that the set of operations on each shared object have a valid serialization. A set of operations has a valid serialization if the result of each individual operation is the same as it would be if the operations had all been executed in some sequential order.

The idea in [1] is to use a set of process history tapes to record partial order of accesses to objects that characterizes an execution. During the monitoring phase, a process history tape is used to record the version number of each shared object accessed by a process; it is modified only by the process. Upon creation, each shared object is assigned a version number 0. Also upon creation, each process is assigned a history tape that is initially blank. During each read or write operation on a shared object by a process, information about the object is recorded on the process's history tape.

Record and Replay makes debugging easier because it enables cyclic debugging. Cyclic debugging is possible with record and replay because order of execution is repeatable. Repeatable execution also makes top-down interactive debugging possible.

5.2 Implementation in Charm++

The record and replay mechanism allows a user to reproduce a program's execution. The key idea in replay is to identify atomic events and record their order of occurrence in the execution. The execution can be replayed by re-executing the atomic events in their recorded order of occurrence. A replay cannot take into account spontaneous events like events which occur periodically with no other causal events.

The first execution run or the record run is used to collect minimum trace data about entry points. Since entry points are atomic events in a Charm program this information is sufficient for replay. Subsequently, the program is replayed at which time extensive trace data is gathered. This trace is used to replay the program execution by re-executing each entry-point on each processor as recorded on the trace. The only information needed to replay a Charm++ program is the order of processing of events on each processor (no information is needed about creation, enqueue and dequeue events) An event can be uniquely identified by a tuple consisting the following pair (i) message-sequence-id (ii) processor-id.

The trace data necessary for replay is an ordered set of such tuples where the ordering is imposed by the order in which events occurred on that processor. Extensive tracing is not required in this scheme. There are some unique issues for replay in the context of Charm because it provides high-level support for dynamic load balancing, quiescence detection and information sharing. Many of the load balancing strategies in Charm have a spontaneous component. The strategy periodically checks the sizes of the queues on the local processor. A replay load balancing strategy implements the known load redistribution. The behavior of the old balancing strategy is therefore not replayed only its effect is. Since minimal tracing is used by the replay mechanism the amount of perturbation due to tracing is reduced. The replay mechanism is proposed as a debugging support to replay asynchronous message arrival orders.

A new trace module is implemented in Charm for the record replay functionality. To include this required tracing into a Charm++ program it is required to link the program with the link option *-tracemode recordreplay*. For every charm message handled a sequence number is assigned to a variable `event` in the envelope. By default, when an envelope is created its member variable `event` is set to zero. Every time an entry method is invoked, depending on where in the sequence of events the invocation happened (the trace module keeps track of the sequence number) the `event` variable in the envelope is set appropriately. When the program is run in record mode (achieved by running the program with a "*+record*" command line option) , a tuple <source pe, envelope size, sequence number> is written to a trace file for every message processed on each of the processor element. There exists such a trace file for every processor element. When the program is rerun using the *"+replay"* command-line option, these trace files are read to achieve the same message sequence. Before each message is processed on a processor element, a check is made to see if the event number of its envelope matches the expected sequence number in the last read tuple from the trace file corresponding to that processor element. This particular implementation is not guaranteed to work with all of the load balancing strategies.

5.3 Testing

To test the record replay implementation, a simple array hello program was used. All this Charm++ program does is create a chare array of size 100 and call the entry method *SayHi* in broadcast mode on the array. In the *SayHi* method, the array element prints a "*Hi*" message along with its index. After the message is printed, the array element reports back to the main chare, by calling the entry method *done* of the main chare. Depending on the order in which the array elements report back to the main chare, the program proceeds or does not proceed successfully to completion. For a certain order of reporting back, a "divide by zero" operation is performed, which results in a floating point exception. This program was run for a varying number of processor elements in the record mode using the runtime option "*+record*" several times till the exception occurred. The probability of the exception occurring in a consecutive rerun is low, therefore making it difficult to reproduce the error deterministically. But, when the program is run using the recorded trace produced when the exception occurred, using the *+replay* runtime option, the exception reproduces itself every time.

Chapter 6

Conclusions and Future Work

The parallel debugger will prove to be a valuable tool to a Charm++ programmer. It allows one to inspect the state of the parallel program during execution. A programmer can keep track of the control flow in the parallel program by setting break points at entry points. The data in the array elements on each processor and the messages in the queues can be retrieved during the running of the program. In this way, via the debugger, the programmer is provided a means of examining the dynamic state of the program. The implemented tool also provides the programmer a way of going about sequential debugging on selected processors on the fly. The record and replay mechanism allows the programmer to deterministically reproduce a program's behavior.

The functionality of the parallel debugger can be extended such that it could be used in conjunction with the existing performance analysis tool for Charm++, **Projections** [13]. There is scope for enhancement of the debugging support to make use of the huge amounts of trace data produced by **Projections**. It would also be very useful to incorporate methods to access network statistics into the existing debugging support. These are just a few of the many possible extensions. The topic of debugging in Charm++ is a green area and there is lots of scope for future work.

References

[1] Thomas J. Blanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[2] J. Cunha, J. Lourenco, and T. Antao. A debugging engine for parallel and distributed environment. In *Proceedings of 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 111–118, Miskolc, Hungary, 1996.

[3] 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.

[4] L. V. Kalé, B. Ramkumar, A. B. Sinha, and A. Gursoy. The CHARM Parallel Programming Language and System: Part I – Description of Language Features. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

[5] L. V. Kalé, B. Ramkumar, A. B. Sinha, and V. A. Saletore. The CHARM Parallel Programming Language and System: Part II – The Runtime system. *IEEE Transactions on Parallel and Distributed Systems*, 1994.

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

[7] 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.

[8] O. Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. In *Proceedings of International Symposium on Computing in Object-oriented Parallel Environments*, Stanford, CA, Jun 2001.

[9] John May and Francine Berman. Panorama: A portable, extensible parallel debugger. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, San Diego, California, 1993.

[10] John May and Francine Berman. Designing a parallel debugger for portability. In *Proceedings of the Eighth International Parallel Processing Symposium*, pages 909–915, 1994.

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

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

[13] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *Projections Manual*, April 1999.

[14] Parthasarathy Ramachandran and L. V. Kalé. Mulitlingual debugging support for data-driven and thread-based parallel languages. Technical Report 99-04, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, August 1999. To appear in the Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC '99).

[15] Amitabh Sinha. *Performance Analysis of Object-based and Message-driven programs*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1994.