© 2010 FILIPPO GIOACHIN

DEBUGGING LARGE SCALE APPLICATIONS WITH VIRTUALIZATION

BY

FILIPPO GIOACHIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor William D. Gropp
Professor Ralph E. Johnson
Luiz DeRose, Ph.D., Cray Inc.

Abstract

Debugging is a fundamental part of software development, and one of the largest in terms of time spent. When developing parallel applications, debugging becomes much harder due to a whole new set of problems not present in sequential applications. One famously difficult example is a race condition. Moreover, sometimes a problem does not manifest itself when executing an application using few processors, only to appear when a larger number of processors is used. In this scenario, it is important to develop techniques to assist both the debugger and the programmer to handle large scale applications. One problem consists in the capacity of the programmer to directly control the execution of all the allocated processors, even if the debugger is capable of handling them. Another problem concerns the feasibility of occupying a large machine for the time necessary to discover the cause of a problem—typically many hours.

In this thesis, we explore a new approach based on a tight integration between the debugger and the application's underlying parallel runtime system. The debugger is responsible for interacting with the user and receiving commands from him; the parallel runtime system is responsible for managing the application, and performing the operations requested by the user through the debugger interface. This integration facilitates the scaling of the debugging techniques to very large machines, and helps the user to focus on the processors where a problem manifests. Furthermore, the parallel runtime system is in a unique position to enable powerful techniques that can help reduce the need for large parallel machines when debugging a large-scale application.

Acknowledgments

I would like to thank my advisor, Professor Kalé, for his guidance during these years, and for his constant encouragement. I would also like to extend my thanks to the rest of my committee for their constructive feedback.

The members of the Parallel Programming Laboratory, present and past, made the years in graduate school a cheerful experience, and I cannot thank them enough for this. In particular, a special thank to Celso Mendes for his numerous proofreading of my writings, Gengbin and Ram for collecting performance results for this thesis, Dave for help with Adobe Illustrator, and Sayantan for the countless evenings at Moonstruck.

During these years in Illinois I have been fortunate to attend a university with a large catholic community. I am thankful for this blessing which helped me grown in faith. In particular, the Koinonia and Grad Rosary groups. Thank you to all their members, present and past, for their friendship and prayers.

Last in this list, but first in my heart, a special thank to my parents, for their support and understanding when I left home to pursue this doctorate, and to Weng Lin for the final push to complete it.

Grants: The work presented in this thesis has been supported by various grants throughout the years: NSF OCI-0725070, NSF 0720827, NASA NNX08AD19G, and NSF ITR-0205611. Several experimental results were obtained using large supercomputing machines, Kraken at NICS and Abe at NCSA, made available through the TeraGrid allocation TG-ASC050039N.

Table of Contents

List of Tables . . . . **vi**

List of Figures . . . . **vii**

1 Introduction **1**

2 Background **4**
2.1 Charm++ 4
2.2 Converse Client-Server Model 5
2.3 CharmDebug 6
2.4 BigSim Emulator 9

3 Handling Large Scale Applications **13**
3.1 Tools Scalability 13
3.2 Unsupervised Execution 16
3.3 The Charm++/Python Interface 17
3.4 Usage within Parallel Debugging 22
3.5 Performance 25
3.6 Related Work 28
3.7 Future Work 29

4 Virtualized Debugging **31**
4.1 Related Work 32
4.2 Debugging Charm++ Applications on BigSim 32
4.3 Debugging MPI Applications on BigSim 35
4.4 Debugging Overhead in the Virtualized Environment 37
4.5 Case Study 40
4.6 Future Work 43

5 Isolating Objects **46**
5.1 The Charm++ Memory Subsystem 47
5.2 The CharmDebug Memory Library 48
5.3 Detecting Cross-Object Memory Modifications 50
5.4 Detecting Buffer Overflow 54
5.5 Performance Aspects 55

5.6 Related Work 58
5.7 Future Work 59

6 Processor Extraction 61
6.1 Related Work 62
6.2 The Three-step Procedure 63
6.3 Ordering Messages 66
6.4 Robustness and Accuracy 67
6.5 Processor Extraction 69
6.6 Further Reducing the Need for Large Machines 70
6.7 Performance 72
6.8 Case Study 80
6.9 Future Work 81

7 Provisional Message Delivery 84
7.1 Provisionally Delivering Messages 85
7.2 Exploring Solutions 87
7.3 Implementation 89
7.4 Performance Evaluation 95
7.5 Case Study 97
7.6 Related Work 98
7.7 Future Work 100

8 Conclusions 101

References 103

Appendix A: Debugging API 111
A.1 Queries to a Charm++ Application 111
A.2 Extending Query Set 113
A.3 Notification Events 113

Appendix B: Python/Charm++ Interaction API 115

Curriculum Vitæ 116

List of Tables

3.1 Client request processing time results in milliseconds with varying number of processors. The Python script runs inside an interpreter connected to a chare group. 25
3.2 Time to execute the script with varying number of calls to the high-level interface. 26
3.3 Time to execute the script with varying number of elements over which to iterate. 26
3.4 Execution time of a 5-point 2D Jacobi application on a matrix with dimension *size*X*size* decomposed into *#chares* chares on 32 processors. 26

4.1 Time taken by the CharmDebug debugger to perform typical operations, using MPI Jacobi3D application with 1024 emulated processors on varying number of physical processors. 40

6.1 Amount of data stored to disk using different recording schemes for ChaNGa. All data in megabytes. 77
6.2 NAMD execution time in seconds with different recording schemes in step one, running on 1024 processors of NCSA BluePrint. The last row is the total time with file I/O. 78

7.1 Performance of single forked process during various provisional delivery operations with relative standard deviations. Measurements in microseconds. . . 96
7.2 Analytical comparison of the two provisional delivery methods for multiple subsequent deliveries. n is the total number of messages provisionally delivered, k the number of messages being undelivered/committed; c, d, m the time for creation of a process, destruction of a process, and delivery of a message, respectively. 97

List of Figures

2.1 Perceived decomposition of a parallel application into migratable chares. Comparison between user and system view. 4
2.2 Example of a charm interface (ci) file. Declaration of a chare array of type `MyArray` of dimension one, with its constructor and entry methods. 5
2.3 Diagram of CharmDebug's system. 7
2.4 Main CharmDebug window with program's output, entry methods list, processor sets, and listing of all the messages enqueued on a given processor. . . . 8
2.5 Screenshot of CharmDebug's introspection mechanism. The information contained in a chare element of type "TreePiece" is unrolled for inspection. Pointers can be dereferenced for further details. 10
2.6 BigSim Emulator Architecture. 12

3.1 Performance of two different global operations on a parallel application: collection of processor status and setting of a breakpoint. In comparison, time take to perform the former operation in the original framework. 14
3.2 Time taken for the debugger to attach to a running parallel application, compared with the launcher's startup time. 15
3.3 Execution flow of a Python code insertion. 18
3.4 Definition of a chare array using the Python interface. 19
3.5 Python code using only the low-level interface, without the iterate mode. . . . 20
3.6 Definition of a chare array using the high-level Python interface. 21
3.7 Python code when using iterate mode. 22
3.8 Screenshot of CharmDebug. 23
3.9 Screenshot of CharmDebug displaying installed Python scripts. 24
3.10 Introspection code to check range of an array. 25
3.11 Client request processing time results in milliseconds with varying amount of computation (different color) and number of processors (X axis). The Python script runs inside an interpreter connected to a single chare. Dotted lines have a new interpreter created every request; solid lines have the same interpreter reused over multiple requests. 27

4.1 BigSim Charm++ Software Stack. 33
4.2 Diagram of CCS scheme under BigSim Emulation. 34
4.3 AMPI model of virtualization of MPI processes using Charm++. 36

4.4 Screenshot of GDB attached to a specific MPI rank, and displaying its stack trace. . . . 37
4.5 Jacobi3D execution time on 1024 emulated processors using varying number of physical processors. The last bar is the actual runtime on 1024 processors. 38
4.6 Jacobi3D execution time on 1M (1,048,576) emulated processors using varying number of physical processors. . . . 39
4.7 NAMD execution time on 1024 emulated processors using varying number of physical processors. The last bar is the actual runtime on 1024 processors. . 41
4.8 NAMD memory usage per process on 1024 physical processors vs. NAMD on emulation mode using from 8 to 1024 physical processors. . . . 42
4.9 Screenshots of CharmDebug parameter window. . . . 43
4.10 Screenshot of ChaNGa debugged on 4,096 virtual processors using 32 real processors. . . . 44
4.11 Inspection of a Jacobi object in the system. . . . 45

5.1 Application linking stage. The application modules are linked together with one Charm++ memory implementation to produce the executable. . . . 47
5.2 Layout of the extra space allocated by the CharmDebug memory library on 64 bit machines. In shaded color the memory allocated for CharmDebug purposes, in white the memory for the user. . . . 48
5.3 Memory view of the allocated memory for a simple hello world program. The color correspondence is: yellow-chare, pink-message, blue-user, red-system. . 50
5.4 Dimmed memory view. In brighter colors are shown the regions of memory allocated by a specific chare, in darker colors all the others. . . . 51
5.5 Normal execution flow of a Charm++ application. . . . 52
5.6 Modified execution flow of a Charm++ application for cross-chare memory corruption detection. . . . 52
5.7 Execution flow of a Charm++ application when *mprotect* is used to detect cross-chare memory corruption. . . . 54
5.8 Overhead to set and reset the memory protection for all the allocated memory using the different protection mechanisms. . . . 56
5.9 Overhead to set and reset the memory protection for 32 MB of memory allocated in 4,000 blocks using the different protection mechanisms. Comparison between the different CharmDebug memory libraries. . . . 57
5.10 Cost to allocate and deallocate a memory block in Charm++ when using the two CharmDebug memory libraries based on `malloc` and `mmap`. Averaged over 10,000 allocations. . . . 58

6.1 Flowchart of the three-step algorithm. . . . 64
6.2 Example of data structure padded by the compiler. . . . 68
6.3 Recording overhead per message using the three schemes. Computed using kNeighbor test (NCSA Abe cluster). . . . 72

6.4 Comparison of kNeighbor total execution time with and without recording schemes (the total time includes file I/O at the end of execution). 74
6.5 Total replay time in step two for kNeighbor (NCSA BluePrint 256 processors). 75
6.6 Comparison of kNeighbor total execution time in step two when recording varying number of processors in full detailed mode. (NCSA BluePrint 256 processors). 76
6.7 Recording overhead for ChaNGa application using the three schemes (on NCSA BluePrint cluster). The tests were performed with optimized code, except for the first bar in black. 77
6.8 Overhead during replay in steps two and three for ChaNGa application (on NCSA BluePrint cluster). 78
6.9 NAMD execution time in replay mode on 1024 emulated processors using varying numbers of physical processors, recording 16 emulated processors. The last bar is the actual runtime in the non-emulated replay mode on 1024 processors. (on NCSA BluePrint). 79
6.10 Screen outputs from replaying two different processors under the emulator. . 80
6.11 3-step procedure used for debugging ChaNGa. 81
6.12 CharmDebug window to set the record-replay parameters. This view shows an execution of the third step of the procedure. 82
6.13 CharmDebug's main view during the debugging of an application on the third step of the procedure. Only one processor is available. 82

7.1 Options available for different system status and message selected. 85
7.2 Screenshot of the message queue. Some messages have been delivered provisionally (in purple on top), while others are still in the regular queue. 86
7.3 Description of the behavior of a processor when provisionally delivering messages. 89
7.4 Control flow of a processor in normal mode. When a CCS request arrives, the processor handles it and replies to the sender. 90
7.5 Control flow of a processor when in provisional mode. 91
7.6 Timeline of the execution on a processor when provisionally delivering messages: several messages delivered provisionally, and a rollback. 93
7.7 Parallel algorithm for prefix computation. 98
7.8 Screenshot of CharmDebug while debugging the parallel prefix application. Multiple messages are enqueued from different steps. 99

1 Introduction

Astrophysicists seeking to understand how the universe evolved to the form we know, or biologists seeking knowledge to defeat viruses that threaten our lives, require enormous computational power. Parallel machines with thousands of processors are the only form of computational power sufficient to address these issues with significant accuracy.

Application developers have to constantly face the problem of errors, or bugs, introduced in their applications while writing them. In fact, the development of an application is often divided into two phases: writing the code to perform a certain operation, and correcting the errors that have been introduced in the first phase. On sequential programs, developers spend more time eradicating bugs than writing new code[1]. For parallel programs, debugging time is even greater.

These errors can be of various natures, and discovering them can be time consuming. Examples of errors in simple programs range from simple type mistakes, generally easier to detect and solve, to problems in the implementation of the desired algorithm (i.e the application produces incorrect results), which tend to be harder to identify.

In parallel applications, these bugs become harder to discover. For example, a memory corruption may not only manifest itself later in the execution, but also on another processor. Moreover, other bugs are added to those present in sequential programs. These new bugs typically come from problems in the coordination among the various processors involved in the task. For example, in distributed memory systems, the transmitted data or the order in which this data is processed can be erroneous. In shared memory systems the synchronization on commonly shared data structures can be problematic. In general, bugs in parallel applications are very hard to find. One reason is the non-deterministic behavior of many parallel applications due to timing differences between executions, thus resulting in intermittent errors that appear only once in a while.

Moreover, bugs in parallel applications are sometimes completely hidden when executing the application on a small number of processors, only to appear when a larger number of processors is used. One possible reason is races between messages coming from different processors: on more processors it is more likely that an ordering of the messages revealing the error will occur. Running the application repeatedly on small configurations may eventually yield a message ordering showing the problem. Unfortunately, this is not always true: message latencies in the underlying hardware layout may change the application's behavior, and hide completely the problem on the smaller configuration. For example, on larger systems, the network configuration—typically torus or fat-tree in modern parallel computers—may route different messages through different intermediate nodes. If one link is congested, messages can arrive in an unexpected order.

Various tools exist to help programmers discover bugs in applications. Many handle only sequential programs, while some are capable of handling parallel applications. Among the most widely used parallel debuggers are TotalView [2] from TotalView Technologies, DDT [3] from Allinea, and Eclipse [4] from the Eclipse Foundation. All these tools have the capability to handle applications written in C/C++ and Fortran, and parallelized using MPI [5, 6, 7] or OpenMP [8, 9] paradigms. They incorporate source-level debugging, allowing the programmer to step through the source code. They also allow memory debugging by identifying memory leaks, corruptions due to out-of-boundary writes of allocated blocks, and other common mistakes. As production tools, they support various modern platforms and integrate with their batch schedulers.

Some of these tools have recently proven their scalability to two hundred thousand processors running an MPI application. Even when allocating these many processors, the interface they provide to the user has not changed: the user visualize his program and steps through the individual instructions. In essence he is still in charge of monitoring the entire application composed of many thousands of processors. Moreover, this stepping procedure will only work well for application where all the processors proceed in an almost lockstep fashion. If each processor is allowed to branch and execute different parts of the code, then this method becomes impractical.

Another restriction that these tools have is that when debugging an application on thousands of processors, the entire set of processors must be allocated and available at all time during debugging. During a typical debugging session, most of the time is spent with the application waiting for the programmer to decide on the next action. While for sequential programs the cost associated with the usage of the processor is negligible (as the program is typically run on the programmer's workstation), for parallel programs this is usually not the case. Most computing centers require jobs to be submitted to a batch scheduler which will later start them using a set of processors exclusively allocated to the job. Allocating a large number of processors can be impractical, very expensive, or both. It is impracticable because the job may be scheduled to run after a long delay (up to many days) unless a special reservation is prearranged—difficult for debugging sessions which might span over several days with repetitive short runs. It is expensive due to the cost associated with the large number of computing hours used.

The aim of this thesis is to develop scalable techniques to improve the proficiency of debugging message-passing parallel applications. In particular, we target the debugging of parallel applications when problems happen at large scales with thousands, or possibly millions, of processors. Our approach consists in both enabling the user to debug his application using a large number of processors as well as reducing the need for very large machines during the whole debugging process.

When debugging an application on thousand of processors, the developer can hardly control the progress of all the processors manually. The debugger has to provide some utility to let the application run without explicit control by the user, or unsupervised, and halt automatically upon error detection. The error condition can be a variety of events. It can be something automatic, like the abnormal termination of a process or the detection

of a memory corruption by the system, or a condition set by the user, like a traditional breakpoint or a failed correctness check. An infrastructure for the user to insert correctness check dynamically is illustrated in Chapter 3, while a tool for the automatic detection of memory corruption will be presented later as part of Chapter 5.

Since debugging using very large machines can be cost prohibitive or very inefficient for most programmers, a large portion of this thesis is devoted to reducing the need for large machines during the majority of the debugging time. Chapter 4 discusses how an application can be virtualized, and debugged under an emulation environment using only a fraction of the processors requested by the user. In this scenario, the problem of shared resources arises. A discussion on how we propose to solve it is presented in Chapter 5. When a problem can be isolated on a subset of processors, these can be extracted from the whole application and executed under a controlled environment. How to use record-replay techniques in a non-intrusive way to extract processors from a large application is the topic of Chapter 6. Finally, Chapter 7 illustrates how the user can benefit from the capability of testing the effects produced by the delivery of messages in a specific order, with the possibility to rollback the delivery of such messages. Each chapter describes the most related work to the topic it describes, and possible future extension.

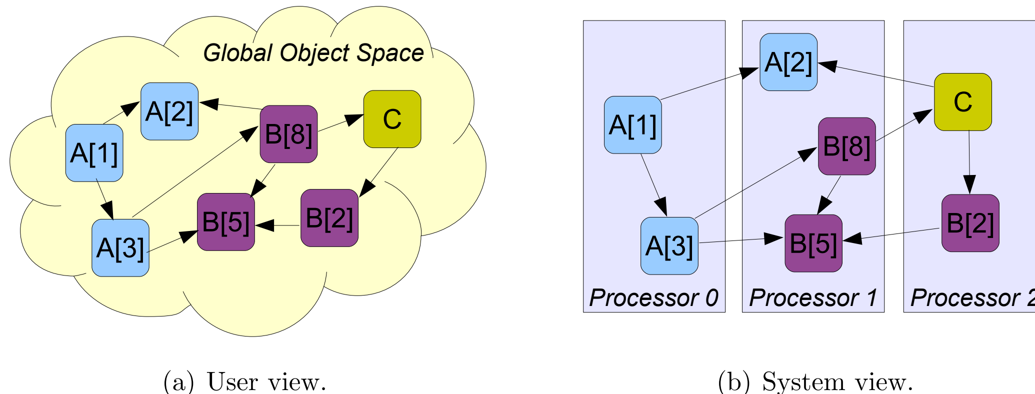
This thesis demonstrates the techniques described above in the context of the CHARM++ parallel runtime system [10, 11, 12], and CHARMDEBUG [13, 14], the parallel debugging tool tailored to CHARM++ applications. Most results are therefore taken from applications natively written in CHARM++. Additionally, some results will be shown using the popular MPI programming model to demonstrate the broader applicability of the proposed techniques. The specific implementation of the MPI standard we used is AMPI [15]. As a side effect, this thesis also provides an integrated environment containing several valuable debugging feature to developers of CHARM++ applications.

2 Background

The techniques presented in this thesis have been deployed in the context of the CHARM++ parallel programming model and the CHARMDEBUG debugger. In addition, some techniques leverage previous work on performance prediction, and in particular the BigSim framework. In this chapter, we shall review the main concepts and infrastructures that will be used throughout this thesis.

2.1 Charm++

CHARM++ [11] is a popular runtime system for developing parallel applications. Several applications have been developed based on the CHARM++ runtime system. Among these are NAMD [16], a molecular dynamics code and winner of the Gordon Bell award in 2002, OpenAtom [17] for Car-Parrinello ab-initio molecular dynamics and ChaNGa [18] for cosmological simulations.



(a) User view.

(b) System view.

Figure 2.1: Perceived decomposition of a parallel application into migratable chares. Comparison between user and system view.

The primary concept of CHARM++ is object virtualization [19]. In CHARM++, the user divides the computation into small objects, called *chares*. These chares live inside a "global object space", as depicted in Figure 2.1(a). The runtime system assigns these chares to the available processors, as illustrated in Figure 2.1(b). By measuring the computational time utilized by each chare, and by dynamically changing the assignment of chares to processors, the CHARM++ runtime system can automatically load balance the application [20]. The

CHARM++ infrastructure also offers other automated performance optimization techniques, such as communication optimization [21].

```
module MyCharmModule {
    array [1D] MyArray {
        entry MyArray();
        entry void MyMethod();
        entry void MyMethod2(int);
    }
}
```

Figure 2.2: Example of a charm interface (ci) file. Declaration of a chare array of type `MyArray` of dimension one, with its constructor and entry methods.

Chares communicate with each other via asynchronous messages. Messages trigger method calls on the destination chare. These methods are called *entry methods*. The computation performed by an entry method upon receipt of a message depends on the information carried by the message and the internal state of the chare receiving that message. Chares performing the same operation can be grouped into collections of chares, and all chares in a collection share the same *chare type*. These collections are indexable, and are also referred to as *chare arrays*. One special type of collection, where exactly one chare in the collection is present on each processor, is called a *chare group*. Since the state of a chare is encapsulated by the instantiation of a C++ class that embodies it, all chares are considered independent entities, irrespective of their type. The declarations of the chare types available in an application, as well as their entry methods, is specified in a *charm interface* file, or simply *ci* file. Figure 2.2 shows an example of a simple ci file.

As an example, consider a parallel program for the simulation of galaxy formation. This program will likely have multiple modules to compute the various forces present in the universe: gravitational, hydrodynamic, magnetodynamic, etc. For good performance, these modules may run simultaneously, thus allowing the communication of one module to be overlapped with useful computation of another module. By "simultaneously", we mean that any given processor can participate in the computation of all the different forces, interleaving their execution over time. In the context of CHARM++, the program will consist of several different collections of chares, one for each force computation performed during the simulation, such as SPH, gravity, etc. Each collection will consist of many chares, possibly thousands or even millions, and each chare will perform a specific force computation on a small portion of the simulation space. The coordination among them will be described by the messages they exchange.

2.2 Converse Client-Server Model

There are several situations where an end-user, or even an application developer, can benefit from interacting with a live parallel application. For example, visualizing the application's

behavior during its operation or steering the computation dynamically. An application developer may want to debug an application or visualize its performance.

Converse Client-Server (CCS)[22] is a communication protocol that allows parallel applications to receive requests from remote clients. This protocol is part of Charm++'s underlying system specifications and is therefore available to any Charm++ application. Note that "application" does not mean only the user written code, but also the Charm++ runtime system and its modules that run as part of the application itself. In this scenario, if a system module decides to use CCS, the user code does not require any change, unless it wants to explicitly take advantage of the feature.

In Charm++, CCS obeys to the normal Charm++ semantics. Upon a request made by a CCS client, a message is generated inside the application. Computation by the application is triggered by the delivery of this message. As such, CCS requests are serviced asynchronously with respect to the rest of the application which can proceed unaffected. When an application, or Charm++ module, desires to use the CCS protocol, it must register one or more handlers, each with an associated tag. This ensures that requests sent by clients can be correctly matched and delivered to the intended handler. Registration is performed by calling a function in the CCS framework. Moreover, at startup, a flag must be passed to the application to ensure that the runtime system opens a socket and listens for incoming connections. The connection parameters are printed to standard output by the Charm++ RTS. Remote clients can send requests to the parallel application using this information. After receiving a CCS request message, the application can perform any kind of operation, including complicated parallel broadcasts and reductions. Finally, a reply can be returned to the client via the CCS protocol.

2.3 CharmDebug

CharmDebug [14] is a graphical debugger designed for Charm++ applications. It consists of two parts: a GUI with which a programmer interacts, and a plugin inside the Charm++ runtime itself. The GUI is the main instrument that a programmer will see when debugging his or her application. It is written in Java, and is therefore portable to all operating systems. A typical debugging session is shown in Figure 2.3. The user will start the CharmDebug GUI on his own workstation. He can then choose to start a new application to debug, or attach to a running application manually, using the appropriate commands available in the GUI.

By default, every Charm++ application is integrated with debugging support in the form of a CharmDebug plugin. This plugin is responsible for collecting information from the running application, and communicating with the CharmDebug GUI. When a program starts, this plugin registers inspection functions that the CharmDebug GUI will send requests to. This initialization happens during Charm++'s startup without user intervention. Therefore, any program is predisposed for analysis with CharmDebug. Additional debugging modules linked into the application can extend the set of requests handled by the

CharmDebug plugin.[1]

In contrast with other debugging tools, with this plugin integrated in the application itself, no external tool is necessary on every compute node. Thanks to the coupling between these two components of CharmDebug, the user can visualize several kinds of information regarding his application. Such information includes, but is not limited to, the Charm++ objects present on any processor and the state of any such objects, the messages queued in the system, and the memory distribution on any processor.

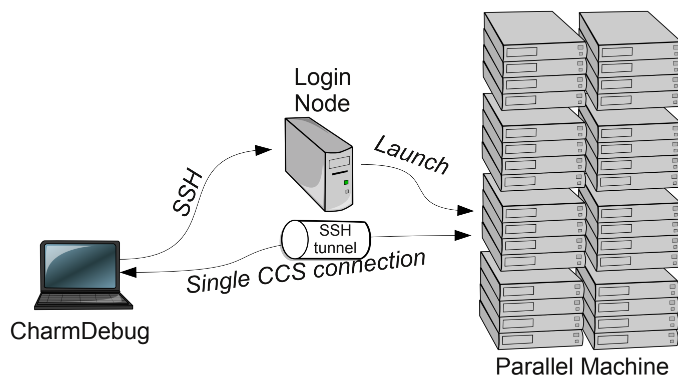


Figure 2.3: Diagram of CharmDebug's system.

In Charm++, every parallel application is integrated with debugging support in the form of a CharmDebug plugin. When a program starts, this plugin registers inspection functions that the CharmDebug GUI will send requests to. This initialization happens by default during Charm++'s startup without user intervention. Therefore, any program is predisposed for analysis with CharmDebug.

The communication between the CharmDebug GUI and the CharmDebug plugin happens through the CCS high-level communication protocol. This leverages the message-driven scheduler running on each processor in Charm++: in addition to dealing with application messages, the scheduler also naturally handles messages meant for debugging handlers. Note that since only one single connection is needed between the debugger and the application under examination, CharmDebug avoids the scalability bottleneck of having the debugger connect directly to each process of the parallel application. Although lacking direct connection to each processor, the user can request the debugger to open a GDB [23] session for any particular processor. This gives the user flexibility to descend to a lower level and perform operations that are currently not directly supported by CharmDebug.

A potential problem for such model is that the communication with the remote application requires messages to be delivered to the CharmDebug's plugin. While the application is executing normally, all messages are delivered by the Charm++ runtime system itself, together with the application's own messages. On the other hand, when the application is

[1]For the list of commands that the CharmDebug plugin handles, and a description of how this list can be extended, please refer to Appendix A.

paused (or frozen) for inspection, the main runtime scheduler responsible for message delivery is halted. This poses the problem of allowing specific messages to still be delivered even while the application is frozen. Another problem is raised when the application is not only frozen, but also under inspection by the user through a sequential debugger attached to a specific processor. The first problem has been solved by modifying the runtime system to selectively allow certain messages to be delivered even while the application is in frozen state. The latter problem is currently unsolved, and a tighter integration between sequential debugger and CharmDebug, beyond the scope of this thesis, is envisioned to solve it.

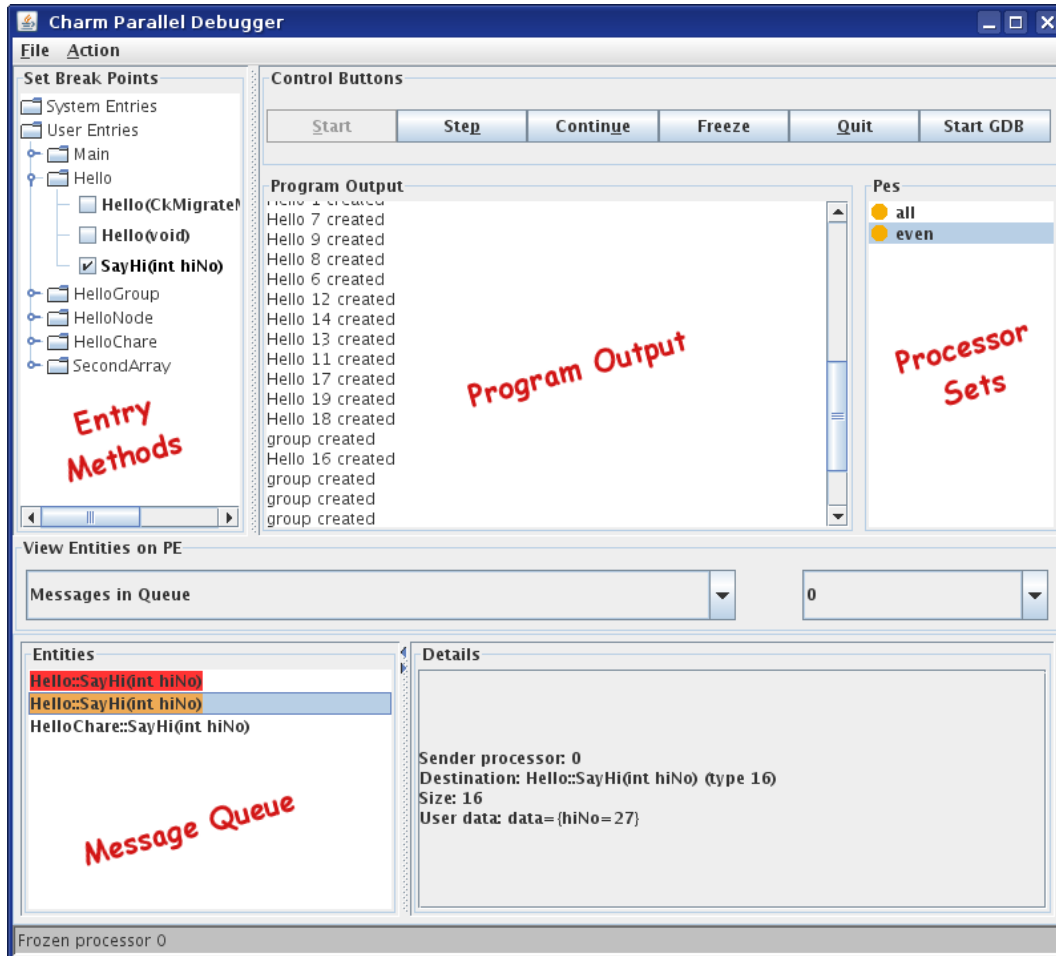


Figure 2.4: Main CharmDebug window with program's output, entry methods list, processor sets, and listing of all the messages enqueued on a given processor.

Figure 2.4 shows the main view available to the user. As in most debuggers, this view contains a mechanism (on the right) to select certain sets of processors to focus on. In the example, the default "all" set is available, and the user created another set called "even" which is currently selected. Since in Charm++ the main computational unit is an entry method, breakpoints can be set at the entry methods' boundaries. These are listed on the

left side. They are grouped by the containing chare type, and separated between internal to the Charm++ system and user-defined. On the bottom, several entities can be visualized. These include the chares and messages in the system, read-only variables and messages, and the properties of each chare/message type. The user can select one entity (on the left) and a processor (on the right) for which to visualize information. When highlighting one entity, its content and/or description is expanded in the "Details" view.

Entities of a Charm++ application, being either chares on a processor or messages in the queue, can be inspected through CharmDebug in two ways. The first method relies on the PUP (Pack/UnPack) framework. According to the Charm++ manual, "The PUP framework is a generic way to describe the data in an object and to use that description for any task requiring serialization. The Charm++ system can use this description to pack the object into a message, and unpack the message into a new object on another processor." During the serialization process, CharmDebug automatically records the type and name of each variable that it serializes. The message created will be interpreted by the client interface later. This approach allows independence from the underlying memory layout or specifics of the architecture. A drawback is that it requires the user to write PUP routines for every type that might be inspected. While many structures might already have a PUP routine, such as those automatically generated by the Charm++ translator, others may be lacking them. Moreover, the already existing PUP routines are usually optimized to reduce the packed version of an object, for example by not including some variables that are known to be not alive at migration time, thus omitting information that would be useful during debugging.

The second method of inspecting application entities is through the "Inspector" framework which was created as part of this thesis. This framework builds a description of the application's object types inside CharmDebug. The raw memory is then loaded directly from the application on demand, and CharmDebug interprets it according to the declared type of the memory. This allows CharmDebug to be completely independent of the application, and enables operations such as memory casts, where the memory can be reinterpreted as different types. While this method is more generic, and can be used to read any data from the application, there are cases where the content of the raw memory is still difficult to interpret. For example, messages containing marshaled parameters store the information as a sequence of bytes, without context. For these, a PUP routine automatically generated by the Charm++ translator is better suited to interpret the data. Thus, the two methods are complementary. Figure 2.5 shows a screenshot from CharmDebug while inspecting a chare element of ChaNGa, a cosmological simulator written in Charm++, using the Inspector framework.

2.4 BigSim Emulator

BigSim [24, 25] is a simulation framework that provides fast and accurate performance evaluation of current and future large parallel systems using much smaller machines, while supporting different levels of fidelity. It targets petascale systems composed of hundreds of

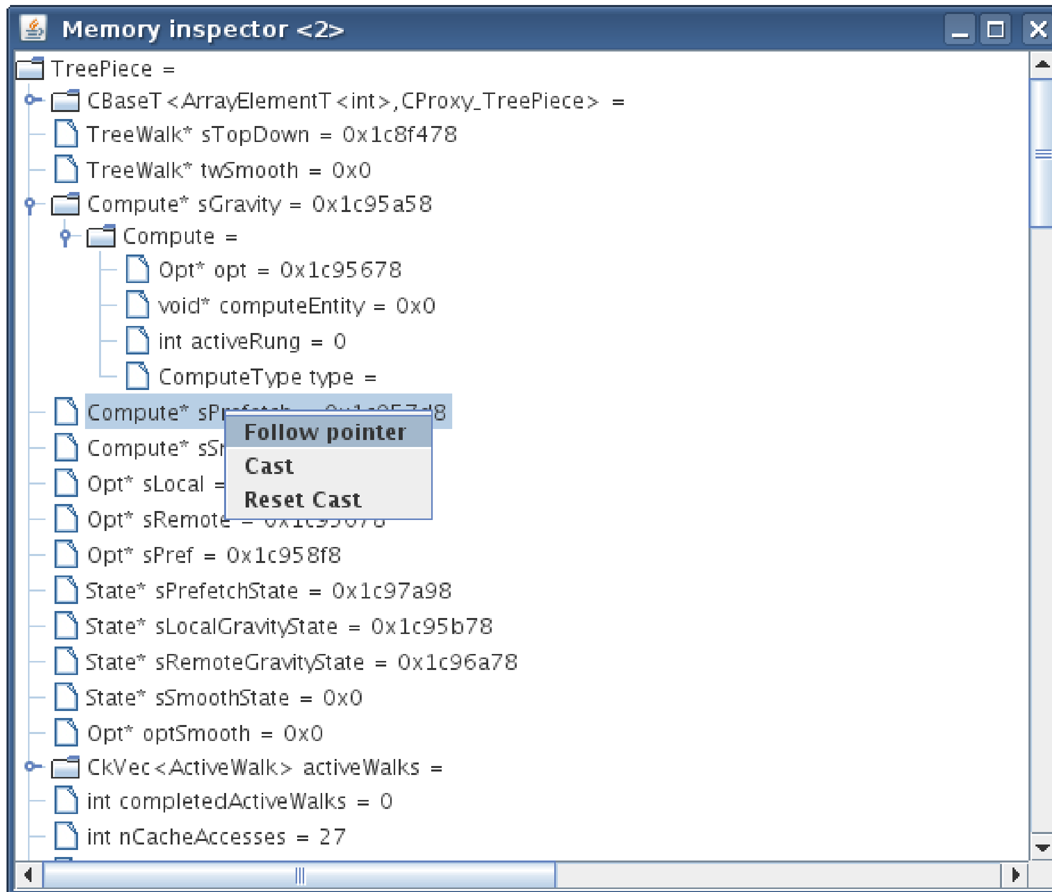


Figure 2.5: Screenshot of CharmDebug's introspection mechanism. The information contained in a chare element of type "TreePiece" is unrolled for inspection. Pointers can be dereferenced for further details.

thousands of multi-core nodes. The main purpose of the simulation framework is to help application developer better understand the application's behavior (including its performance) by actually running the application in a virtualized target machine environment defined by a user.

BigSim consists of two components. The first component is a parallel *emulator* that provides a virtualized execution environment for parallel applications. This emulator generates a set of event logs during execution. The second component is a post-mortem trace-driven parallel *simulator* that predicts parallel performance using the event logs as input, and supports multiple resolutions for prediction of sequential and network performance. For example, the simulator can (optionally) predict communication performance accurately by simulating packets of each message flowing through the switches in detail, using a parallel discrete event simulation technique. Since the simulator only considers the trace logs and does not re-execute the application at the code level, it is not suitable for debugging purpose. However, the BigSim Emulator, which supports emulation of a very large application

using only a fraction of the target machine, is useful for debugging. In the remainder of this section, we shall focus our attention on the emulator component.

Since multiple target processors are emulated on one physical processor, the memory usage on a given physical processor may increase dramatically. It may thus become impossible to fit the whole application into the physical memory available. Interestingly, many real world scientific and engineering applications, such as molecular dynamics simulation, do not require a large amount of memory. For example, in one experiment, researchers were able to emulate NAMD [26] running on a 262,144-core Blue Waters machine [27] using just 512 nodes of the Ranger cluster, a Sun Constellation Linux Cluster at the Texas Advanced Computing Center (TACC).

For applications with large memory footprint, the physical amount of memory available per processor indeed poses a constraint. However, even in this scenario, we can still emulate these applications by using an efficient out-of-core technique [28, 29] optimized for the BigSim Emulator. During an out-of-core emulation, the emulator can move the memory associated to a virtual processor between the physical memory and a bulk storage, such as local hard drive. When a virtual processor that resides on the bulk storage is needed during the emulation, the emulator fetches the entire data of that virtual processor into the main memory before scheduling it. This process is similar to the swapping performed by operating systems. However, emulator runtime tends to schedule the swapping more efficiently due to its knowledge of when and what data will be needed by looking ahead in the message queues.

Clearly, out-of-core execution, even with optimization, incurs a much higher overhead than the pure in-memory execution, mainly due to the constraint imposed by disk I/O bandwidth. For example, a slowdown of about 18 times in terms of the total execution time of a Jacobi application was observed in [28].

2.4.1 BigSim Emulation Implementation Details

BigSim Emulator started as an emulator for the Blue Gene/Cyclops machine. Later, it evolved into a general purpose emulator for a large variety of parallel machines, several at the petascale level, including Blue Waters [27]. It is written using CONVERSE [30], a portable and efficient runtime system that supports a variety of communication sub-systems such as infiniBand, Myrinet, and IBM Blue Gene DCMF, among others. CONVERSE also provides support for user-level threads via a common machine independent interface.

Figure 2.6 illustrates the architecture of the BigSim Emulator. In the figure, the box represents one physical processor, many of which are used in a parallel emulation. In this example, the physical processor is emulating two nodes of the target machine. In general, however, this number will be much larger. Each target node, or virtual node from here on, represents a multi-core processing unit with shared memory. Based on their functionality, these cores are divided into communication and worker processors. Both are emulated as CONVERSE user-level threads. Communication processors are in charge of polling the network for incoming messages, and delivering them to the correct worker processors. This is done by storing each message into the message queue associated to the destination worker processor of that message. Similarly, worker processors continually dequeue messages from

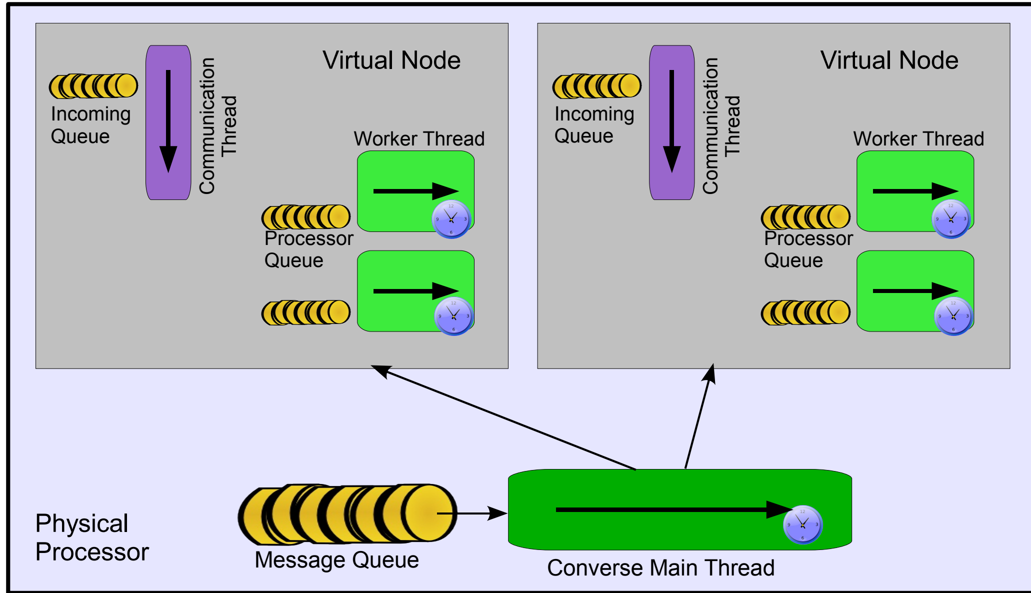


Figure 2.6: BigSim Emulator Architecture.

their own message queues, and handle these messages appropriately. This generally implies calling a handler function specified by the message itself. Certain messages that are not assigned to a specific worker processor may also be handled directly by a communication processor. Note that the network polled by the communication processors is not the physical network, but the virtual network established by the emulator itself. On each physical processor, in addition to the virtual nodes, there is also a CONVERSE scheduler. The main function of this scheduler is to poll messages from the CONVERSE queue, and dispatch them to the recipient emulated virtual node. The CONVERSE queue contains all the messages arriving from target nodes emulated by other physical processors, as well as the messages sent from one virtual node to another virtual node in the same physical processor.

3 Handling Large Scale Applications

When debugging a parallel application that exhibits problems only on very large machines, in addition to the usual problems the programmer faces while debugging his application, there is also the problem of the amount of data that needs to be handled. This problem affects both the debugger system infrastructure, as well as the final user of the debugger. The debugger's infrastructure has to be capable of handling large applications efficiently, without increasing the latency for common operations such that the usability would be hindered. The user must not be overwhelmed by the amount of information that he has to consider. Both of these aspects are equally important to allow the debugging of a parallel application on a large machine, and will be the main topic of this chapter. [1]

3.1 Tools Scalability

Several tools exist to help a programmer to debug his parallel applications. Among the most widely used there is TotalView [2] from TotalView Technologies, and DDT [3] from Allinea. Historically, both these debuggers have maintained a direct connection to each processor in the application. This hinders usability when using thousands of processors due to the large response time for even simple operations. Recently, in concurrency with this thesis, Allinea has developed a new infrastructure based on tree connectivity to improve the collection of data from a large scale parallel application [32]. Other tools like STAT (Stack Trace Analysis Tool) [33], based on MRNet [34], have shown excellent scalability to very large machines. Unfortunately, STAT is not a full debugger, and it can only focus the programmer toward a set of processors to look at. Moreover, the range of errors detected is limited.

All the tools described use external programs to control the processes embodying the parallel application. This implies that the debugger has to understand the specific implementation of the application's underlying runtime system and communication library. For example, to provide the processor rank or pending messages of an MPI application, the debugger has to know how the specific MPI implementation stores such information. Moreover, the debugger has to implement its own communication infrastructure to scale to large machines.

In our approach, we leverage the runtime system underlying the parallel application to extract the information in an implementation independent way. This implies, in particular,

[1] Part of the work in this chapter is reprinted, with permission, from the Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS 2009): "Dynamic High-Level Scripting in Parallel Applications", by Filippo Gioachin and Laxmikant V. Kalé ©2009 IEEE. [31]

that we can utilize the same network infrastructure that the application is using to scale the collection of the data from the parallel application. By doing so, the debugging techniques scale together with the scalability of the application's communication infrastructure.

While it is clear that CHARM++ provides a flexible runtime system, this may be less intuitive for other programming models. For example, in MPI applications, MPI is considered a library in charge only of the communication between processors. Nevertheless, if one considers a call to an MPI function as a transition between the user code to the runtime system, MPI can be considered a runtime system too. In general, when the application calls an MPI function, the specific MPI implementation can perform any operation it desired, and may delay the return of the control to the application. This same analogy can be applied to any parallel programming paradigm.

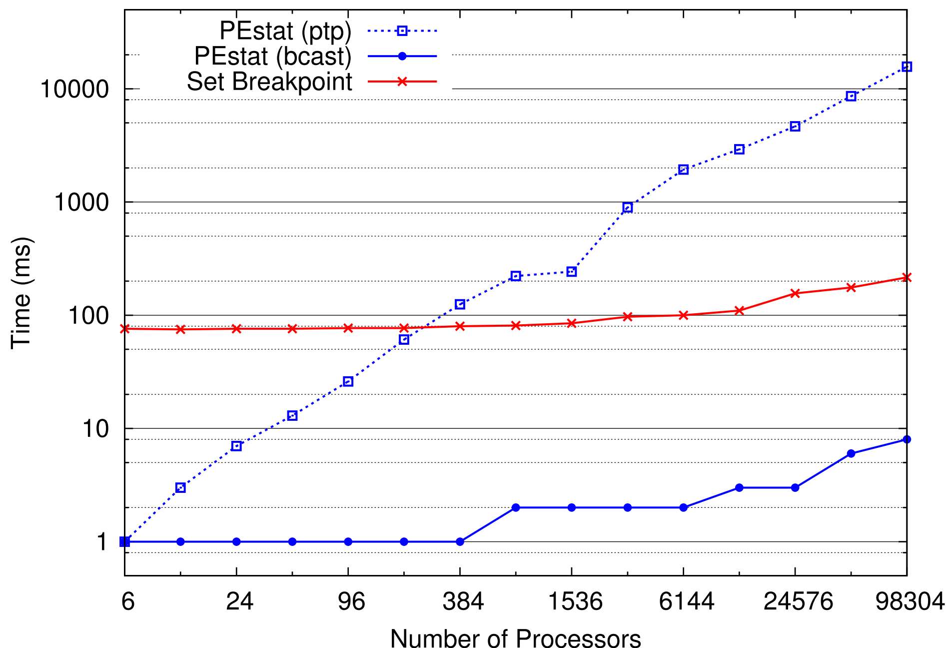


Figure 3.1: Performance of two different global operations on a parallel application: collection of processor status and setting of a breakpoint. In comparison, time take to perform the former operation in the original framework.

Figure 3.1 shows two examples of operations that the user may perform. All measurements are taken on the client side, therefore including any possible network latency. The experiments were conducted using Kraken, a Cray/XT5 machine at the National Institute for Computational Sciences at the University of Tennessee, and a simple CHARM++ application compiled using MPI as CHARM++'s underlying communication layer. The blue line at the bottom of the figure plots the time taken to collect some information from every processor, in this case the status in which the processor is (running, stopped, crashed, etc). In the red line (horizontal in the middle of the figure) the user is setting a breakpoint on all processors. In this case the time is higher since the sequential time on each processor to set a breakpoint is higher.

It is important to note that using a single connection to the parallel application is a necessary condition for high scalability, but not a sufficient one. To explain this, the dotted blue line in Figure 3.1 shows the time taken by the status collection operations, using the original implementation of the CharmDebug framework. In this case, the single connection was used very poorly by querying each processor individually, one after another. The line shows a clear linear trend. The main problem was the lack of a multicast-reduction mechanism for general CCS requests: a request could only be delivered to a single processor. Thus, the naive implementation was to use the CCS connection to loop over all processors. This was solved in this thesis by generalizing the CCS framework to natively support broadcast and multicast requests, thus obtaining the better performance.

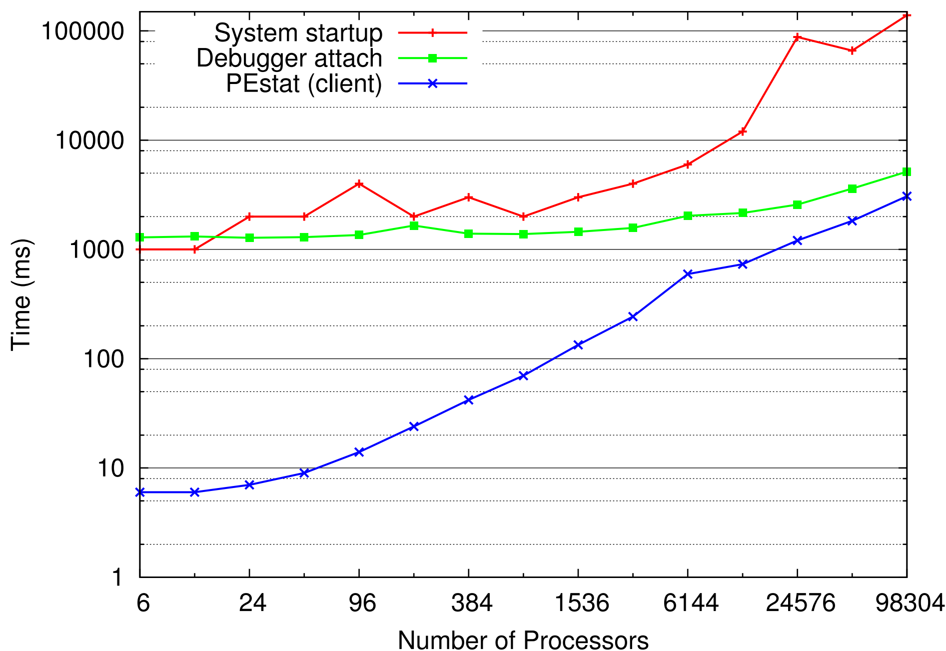


Figure 3.2: Time taken for the debugger to attach to a running parallel application, compared with the launcher's startup time.

During the experimentation phase, we also measure the time it takes for the debugger to attach to the running application. This is plotted on the green curve (in the middle) of Figure 3.2. As it can be seen, it scales perfectly until a few thousand processor, after which it starts showing a slight degrade in performance. This is due to the sequential work performed by the client to instantiate a data structure containing the status collected from all the processors (shown on the blue line at the bottom). Note that this information is minimal (a few bytes per processor), but on very large configuration it can alone account for a non-negligible amount of time. This suggests that to continue scaling to even larger machines, either the client itself is parallelized to some extent, or the client cannot be allowed to store any per-processor information. This second approach might add latency even for simple refresh operations if the require communication with the application. Another consideration worth noting is the time taken to launch the application on the parallel machine, shown

on the red line on top of the figure. Even for a fast job launcher like ALPS, starting an MPI application on one hundred thousand processors takes more than two minutes. In comparison, the five seconds taken by the debugger to attach are still negligible.

3.2 Unsupervised Execution

Even if the debugger program can handle a parallel application running on many thousands of processors, the programmer that is using the debugger may not. He can be easily overwhelmed by the amount of information presented to him. In current debuggers supporting MPI parallel application, the user still follows the traditional stepping programming model: he follows the execution on the various processors by looking at the program execution line by line. While this might still work for MPI application to some extent, for other programming paradigms this is not applicable. For example, for CHARM++ applications, there is no clear common control flow among processors: each of them executes different entry methods asynchronously. In this case a lockstep approach would be completely infeasible.

Instead, the way we see the programmer executing his application is with less direct control, and some help from the application's runtime system to control the application and notify the user when something interesting happens. We call this mode of operation "unsupervised" since it does not require the direct control of the application from the user. A similar approach is taken by the Abnormal Termination Processing (ATP) available on Cray systems [35]. In this case, there is no user monitoring the application, and if the application aborts, the operating system will gather some information for some post-mortem analysis. This information does not cover the entire application, but only of some processors of interest. This differs from our approach since in our case the user is available when the application raises an exception, and he can conduct live analysis on the still running application.

The type of events that the user can be notified of can be divided in two categories.[2] On one side there are operations that the runtime system can perform automatically on the application. For example, if the program triggers a fatal exception, the runtime system can maintain the faulty processor running for immediate inspection. As part of this thesis, we developed a simple mechanism to detect signals triggered by the application, report them to the user, and freeze the faulty processor for further inspection. Another type of automatic check is memory corruption detection among virtual objects in the system. This will be described in detail in Chapter 5.

The other category of notifications regards events that the user explicitly inserted into the running application. A typical example is the setting of a breakpoint, when it is triggered, the user is notified of the event and can follow the execution in detail. Another condition that is notified to the user is the failure of an assertion in the code. Assertion are commonly inserted into the application during compilation, and they describe what the user believes should not happen by looking at his code. While interactively debugging his application, however, the user may realize that some other condition he was not expecting should be

[2] Please refer to Appendix A for the list of notification events currently available in CHARMDEBUG.

checked, and reported upon failure. The typical solution is to stop the application, modify the source code, recompile, and rerun it. The recompilation process can take a significant amount of time, especially for large applications. This reiterated overhead can significantly reduce user productivity. In the case of parallel applications, there is additional overhead in the re-submission of the modified program to the scheduler queue, where it might stay for a long time before being scheduled again for execution. Moreover, sometimes the source code might not be available, making it impossible to add the desired functionality.

In the remainder of this chapter, we shall present a solution developed inside the CHARM++ parallel runtime system. This solution provides the user with a scripting interface to dynamically insert Python scripts to perform different kinds of correctness checks. In particular, this scripts can be executed either just once, or periodically upon a condition. Since Python is a high-level scripting language, it can contain control flow statements, allowing great expressiveness. Moreover, Python code is typically more compact than C or Java code, and it is well established that programs written in scripting languages are easier to write than programs written in declarative languages [36, 37]. In particular, since this script is dynamically inserted into the running application, this can lower the probability of a bug in the checking code itself.

3.3 The Charm++/Python Interface

Other than the debugging scenario presented in the previous section, there are many situations where some new functionality might be desired at runtime. While analyzing scientific data, intermediate results can steer the user towards new and unexpected hypotheses. Unforeseen procedures might be needed to prove or disprove these hypotheses. During a long-running simulation, the user might want to steer the simulation by modifying some parameter or internal data structure. For example, he might want to inject new molecules while studying the behavior of an enzyme. In all these situations, it would be convenient to simply write the function needed, upload it to the running application, and use it immediately.

In this section, we present the generic interface we developed to allow the dynamic insertion of an arbitrary Python code into a running CHARM++ application. With this interface, a user can write Python code that will be uploaded to the running parallel application. Here, the Python code will be executed, and it will be able to interact with the main CHARM++ code. How the code can interact with the main application is decided by the application developer, but the interface allows enough flexibility to cover a wide variety of uses, if not all. Since CHARM++ is implemented in C++, we utilized the Python/C API [38] to make the two languages interact and exchange data.

3.3.1 The Interface

We used the CCS protocol as the basic communication mechanism between remote clients and the parallel application, also referred to as server. Upon CCS, we implemented our

interface to facilitate the programmer's task to augment the parallel application to interact with uploaded Python code, and to create clients capable of generating Python requests.

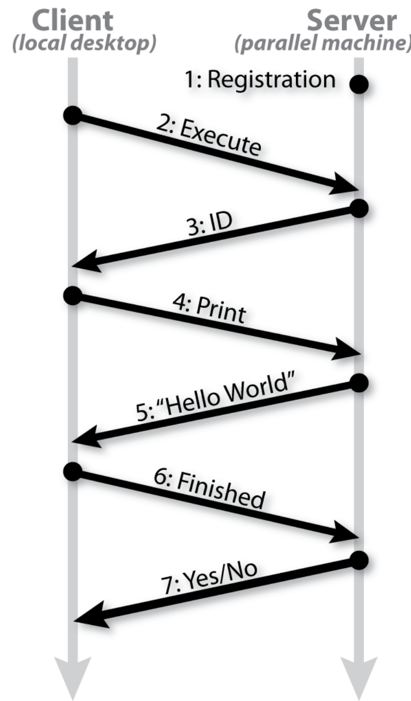


Figure 3.3: Execution flow of a Python code insertion.

A typical control flow for inserting a Python code is illustrated in Figure 3.3. At the beginning of the execution, as with any other application using the CCS protocol, the server registers a string identifying Python requests (step 1 in the Figure). Subsequently, a remote client can send an *Execute* request containing Python code. The server will, at this point, encapsulate the code into a message and schedule it together with the other messages present in the system. Upon delivery of the message, the server will create a new Python interpreter using the Python/C API, initialize it with some basic information, and execute the user code inside this interpreter (step 2). An ID representing the interpreter will be sent back to the client for later usage (step 3). The client can then probe the server with *Print* request to see if the code running on the server printed anything (step 4 and 5), possibly multiple times. Depending on the client setup, the server can finally be queried for completion of the uploaded code with a *Finished* request (step 6 and 7). In the default configuration, the server destroys the Python interpreter at the end of the Python code. This destruction can be overridden by the client, as we shall see. Notice, the application does not need to be stopped, and can continue with normal execution. The request is delivered to the application as a message, and is scheduled as any other message present in the application. Therefore, insertion points are between message deliveries. In some scenario, the user might also want to submit more than one request simultaneously.

Each of the three CCS messages—Execute, Print, Finished—is encapsulated by a C++

and Java object. Client programs written in these two languages can use the provided classes to communicate with the parallel application. These classes contain a series of flags that the user can set to modify the behavior of the server. For example, a Request can return an ID for the interpreter as soon as it is created, or wait on the server until the entire Python code has run and return a reply only at that point. In this second case, an Execute request also provides the functionality of a Finished request. The client can be implemented in multiple ways. It can maintain a CCS connection for each Python request and use them to retrieve the prints from different requests concurrently, or maintain a single CCS connection to the server, and periodically probe for prints from the various active Python requests.

As described, the server returns an ID for the interpreter used to serve a particular Execute request. This ID can be used by the client in multiple ways. First, it is needed to retrieve the finish status, or any print generated by the Python code. Second, interpreters can be set to be persistent. In this case, the parallel application will not discard the interpreter at the end of the execution of the Python code. The client can then use this ID to issue a new Execute request on the existing interpreter. In this way, the server will internally maintain the environment set by a previous request, and build upon it. This can be used to upload Python routines and modules, and subsequent requests can contain code using these modules. As we will see in the results, the reuse of an interpreter also has performance benefits.

In the context of Charm++, all computation is performed inside entry methods, and within the scope of the chare to whom the message was delivered. Dynamically uploaded Python code is not an exception. When the code is uploaded and executed, it runs inside the scope of a chare, determined during step 1 of Figure 3.3. Notice that during registration either a single chare or a collections of chares can be registered; in the latter case, the same script runs independently in each chare of the collection. In the following section, we will describe three ways in which the Python script can interact with the Charm++ application: (1) *low-level*, to allow the Python script to perform simple queries on the hosting chare; (2) *high-level*, to allow the Python script to perform more complicated parallel operations on the entire application; and (3) *iterative*, to apply a Python method to a set of objects provided by the hosting chare.

```
module MyPython {
    array [1D] [python] MyArray {
        entry MyArray();
    }
}
```

Figure 3.4: Definition of a chare array using the Python interface.

3.3.2 Cross Communication

Figure 3.4 shows the *ci* file for the server definition of a Charm++ array that can receive Python requests. As can be seen by comparing it with Figure 2.2, the only addition to a

normal definition of a Charm++ array is the keyword “[python]” in the definition of the chare array MyArray. The other necessary change to the user code is the registration of a string for CCS requests to be identified as Python requests for the chare array MyArray. The registration is a simple function call, made by processor zero, into a registration routine with the string as parameter. These two simple modifications are sufficient to have Python code execute inside interpreters bound to the chare array MyArray. Naturally, if the Python script could not interact with the chare object itself, it would not be very useful. There are three interfaces to allow interaction between the code running in the Python interpreter and the chare object linked to it.

The simplest way for the Python script to interact with the parallel application is through the *ck* module. This Python module is imported into the interpreter before the user script is allowed to run (by executing a default code), and allows Python to query some properties of the Charm++ environment. These are defined by the system, and include some standard properties, like the processor and the node it is running on, and some specific data about the chare running the interpreter, like the index of the chare inside the collection.[3] In addition, there are two other methods, namely *read* and *write*, through which the Python script can read and write variables with the same access privileges as the containing chare has. The user-defined C++ class (MyArray) inherits these virtual methods from a system-defined C++ class. MyArray can redefine them overriding the default empty behavior.

The *read* method accepts as input parameter a single object, representing where the data should be read from. This object can be a tuple or a list, thus allowing multiple values to be passed in. An example of usage is illustrated in Figure 3.5 (which will later be described in more detail). Here, we pass a tuple of two values, a string and an integer, as input parameter, and return a single integer as output. To handle input and output from/to Python, the programmer can use the standard Python/C API as well as an extra API provided by our interface (not described here). The *write* method accepts two parameters, one representing where the data should be written to, the other what data to write. It is up to the programmer to define the *read* and *write* methods to correctly interpret the parameters passed as input. A mismatch between definition and usage of these methods generates an exception. If these methods do not give access to some portion of the data, the Python code will not have access to it. This allows some control on what Python can access. Similarly, some data can be made read-only by having it accessible through the *read* methods but not the *write* method.

```
size = ck.read((”numparticles”, 0))
for i in range(0, size):
    vel = ck.read((”velocity”, i))
    mass = ck.read((”mass”, i))
    mass = mass * 2
    if (vel > 1): ck.write((”mass”, i), mass)
```

Figure 3.5: Python code using only the low-level interface, without the iterate mode.

[3]For a complete list of available functions, please refer to Appendix B.

Nevertheless, the information gathered through the *ck* module is limited to the scope of the processor and the chare that executes the script. If the script requires information generated from a combined operation on all the chares in a collection, say the maximum value of a variable, then another Python module called *charm* can be used. This Python module is constructed to contain all the methods in the *ci* file declared as *python*. The example in Figure 3.6 shows the method `run` declared as such.

```
module MyPython {
    array [1D] [python] MyArray {
        entry MyArray();
        entry [python] void run();
    }
}
```

Figure 3.6: Definition of a chare array using the high-level Python interface.

There are two differences between the methods of the *ck* module and those of the *charm* module. The first is that the "python" keyword can be used in conjunction with as many entry methods as needed, thus augmenting at will the set of functions available through the *charm* module. Each of these functions can accept any number of input parameters (the input parameters of the Python calls are always passed by the Python/C API into the C function as a single tuple object), and provide different functionality. The second is that methods of the *charm* module are run inside a user-level thread. This allows the method to issue parallel operations and suspend itself while waiting for the results. On the other hand, creating a user-level thread has a small but not insignificant cost[39], making the high-level interface slightly more expensive. Notice that parallel operations are initiated by the C++ functions defined in the user class (MyArray), and not by the Python script itself, which can always communicate only with its enclosing chare.

Finally, while the above modules allow the Python script to access the underlying chare and parallel application, there are situations when this is not the best approach. Sometimes a small operation needs to be applied to large sets of homogeneous structures in the parallel application. An example is shown in Figure 3.5. Here, we want to double the mass of all particles with high velocity, but the user may want to apply many different operations to such particles. One way to solve this problem is by utilizing the previously described interface. The user can write a loop over the desired particles, and by using the low-level or high-level routines, access all the needed information. Each call to `ck.read` and `ck.write` in the script invokes the *read* and *write* methods, respectively, of the user-defined class MyPython. These methods will retrieve/store the information from/to the appropriate locations. Most likely, these locations will be some variable declared inside the MyPython class itself. The other way would be to have the CHARM++ application iterate over the available particles, and call a simple update method with each particle as input. This is the third method of interaction between the Python script and the CHARM++ application. The user defines two functions in the chare to provide a *begin* and *next* iterator over the particles. CHARM++ uses these

two functions to iterate over all the particles, and the user-provided Python method will be applied to each particle. The Python code for this iterative mode is shown in Figure 3.7. This approach can significantly reduce the complexity of the code that the user has to write (in our example from six lines of code to two), and therefore reduce the probability of making a mistake.

3.3.3 Error Handling

```
def increase(p):
    if (p.velocity > 1): p.mass = p.mass * 2
```

Figure 3.7: Python code when using iterate mode.

There are two possible situations in which an error is raised while running an uploaded fragment of code: (1) the uploaded code has an error, (2) the interface code written inside the parallel application is buggy.

In the second case, there is not much that can be done to prevent the application from terminating. While it is possible to capture most signals and errors, the application will likely be in an inconsistent state. If this happens, the application should be corrected.

The case where the uploaded script is erroneous should be tolerated to the extent possible. In our implementation, if the Python code raises an error, this error will be captured and reported to the client as a regular printed string. The user will be able to see the problem, and possibly correct the script and upload a new request with a corrected code. In the implementation presented in this paper, if the erroneous script modified the state of the application before raising the exception, these changes could not be undone automatically. Therefore, it is up to the user to consider what has executed, and take appropriate actions.

Sometimes, recovery may not be easy after the script has left the application in an inconsistent state. In these cases, we still want the possibility to recover from a failure as much as possible. One possibility to expand the coverage of automatically recoverable errors is to use a checkpoint-restart approach. For example, the provisional delivery methodology that will be presented in Chapter 7 can be used. With this improvement, the state of the application can be saved before executing the Python script, and restored upon failure of the script.

Even with the current limitations, we believe that our technique is still valuable. Moreover, an appropriate definition of the atomicity of operations that modify the application (e.g low-level vs. high-level) can help the final user to better recover from mistakes.

3.4 Usage within Parallel Debugging

We built upon the existing CharmDebug system. We used our interface to provide an introspection platform to the user. The user can upload Python code to run only once, after every message processed by the program, or selectively only after a subset of messages. This

code can perform checks on the status of the system and identify problems at an early stage. The script is bound to a chare collection selected by the user, and has access to any variable accessible to that chare.

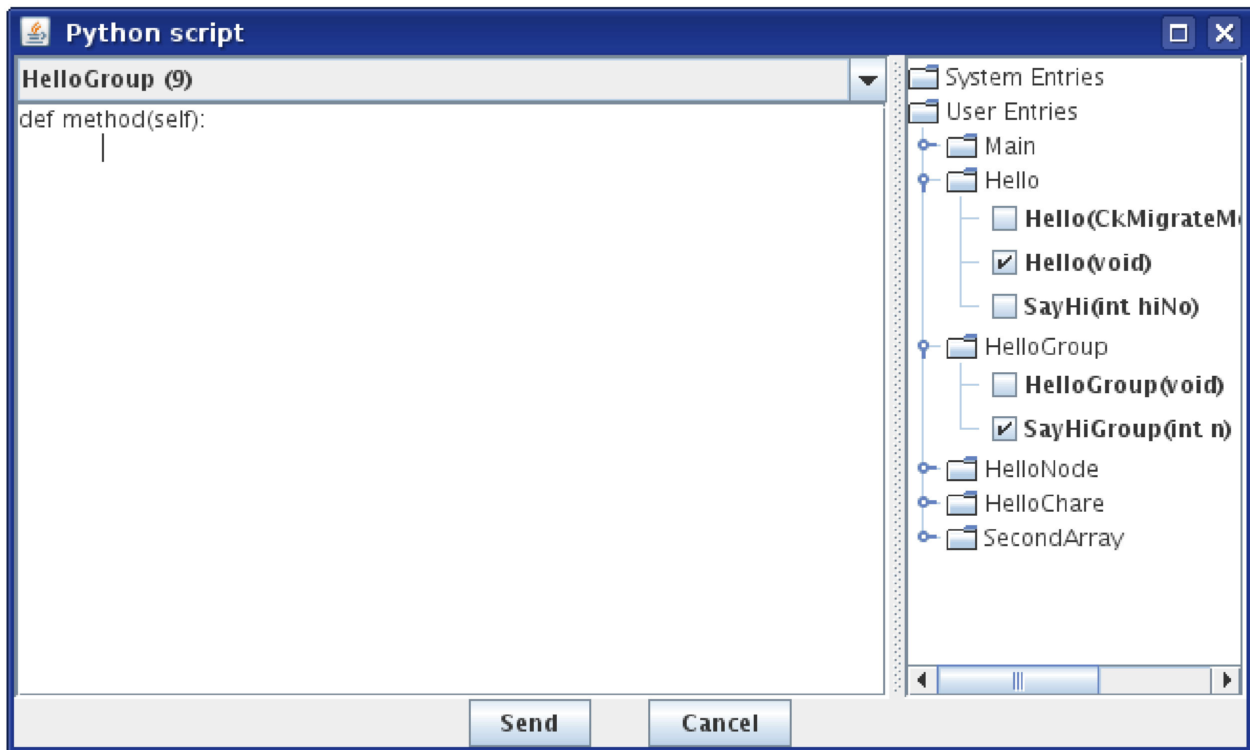


Figure 3.8: Screenshot of CharmDebug.

Figure 3.8 shows a screenshot taken from CharmDebug. On the right side the user can select the entry methods after which the introspection code should be run, or if None is selected, the code will run only once. On the left side, the user can choose the instantiated chare collection that will be hosting the script (at the top), and enter the actual Python code below. Once the code is sent, the CharmDebug plugin inside the parallel application will receive the code, and either execute it immediately and only once, or install it for repeated use. If installed, the code will then automatically be triggered when the specified entry methods are called. The user can also visualize the list of Python scripts installed. An example of this view is shown in Figure 3.9. If the Python script returns any value other than "None", the parallel application will be suspended, and the user will be notified. He can then use the other views of CharmDebug to inspect the application state in more detail.

As we have seen in Section 3.3.1, to have a Python script delivered to a particular chare collection, the *ci* file requires that collection be defined as "python". Nevertheless, we did not want to require the programmer to declare every chare collection as "python".

To solve these problems, we used a chare group as target for the Python script. This chare group is called *CpdPythonGroup* and is part of the CharmDebug plugin module.

Python script

Entry Point	Where	Associated Chare	Code

ghostsFromLeft(int width, const dou...	BEFORE	CkArray (8)	def method(self):
			len = charm.getStatic(int, block_height);
			array = charm.getValue(self, Jacobi, temperature);
			for i in range(0, len):
			value=charm.getArray(array,double,i);
			if (value>10 of value 0):
			return i;

ghostsFromBottom(int width, const ...	BEFORE	CkArray (8)	def method(self):
			len = charm.getStatic(int, block_height);
			array = charm.getValue(self, Jacobi, temperature);
			for i in range(0, len):
			value=charm.getArray(array,double,i);
			if (value>10 of value 0):
			return i;

Figure 3.9: Screenshot of CharmDebug displaying installed Python scripts.

CpdPythonGroup uses the *iterative* method, as described in Section 3.3.1, to iterate over all the chares in the chare collection selected by the user from the dialog box in Figure 3.8. For each chare in the collection, the user-specified Python code is executed in conjunction with that chare. This Python code can access a variable inside that chare by using three helper functions. These functions are exported by CpdPythonGroup through the high-level interface into the *charm* module. They are: *getArray*, to browse through arrays, *getValue*, to return a specific field of a data structure, *getMessage*, to return the message being delivered, and *getCast*, to perform dynamic casts between objects. All three functions return either opaque objects or simple type objects, such as int or float. Opaque objects can represent any complex data structure in the application, similar to a void pointer in C++. Simple type objects represent primitive data types in C++. To start browsing, the Python code receives an opaque object representing the chare on which it is running as input.

Another major challenge was the fact that while we wanted to have full introspection capability, where the user would be able to read and modify all variables, C++ does not support reflection. This means that at runtime, the application alone cannot identify its data layout. Therefore, the opaque object alone is not enough for the helper functions to provide the desired functionality. Our solution was to require the user to specify the type of every object when calling any of the helper functions. The CharmDebug graphical tool modifies every call to the helper functions, and adds the additional information needed at runtime to browse through the data structures. For example, in a call to getValue, CharmDebug adds to the parameters (1) the type of the resulting value and (2) the offset of the requested field from the beginning of the requested class type. This information is available to CharmDebug since it internally constructs a representation of the class hierarchy of the running application. This representation is needed by CharmDebug for other purposes, therefore its creation is not an overhead.

Figure 3.10 shows an example of code that can be issued through CharmDebug to perform introspection checks on the running application. Here, we perform a simple check on an array of doubles, to check if their values are between some bounds. Initially, we load the size of the array into the integer value "length" and the opaque value representing the

```
def check(self):
    length = charm.getValue(self, MyArray, len)
    arr = charm.getValue(self, MyArray, data)
    for i in range(0, length):
        value = charm.getArray(arr, double, i)
        if (value > 10 or value < -10):
            print "Error: value ", i, " = ", value
            return i
```

Figure 3.10: Introspection code to check range of an array.

C++ array "data" into the value "arr". Then we loop through all the values in the array, retrieve the ith element, and check if it is within range. If not, we return a value to stop the parallel application.

3.5 Performance

The time the user has to wait between sending the code and receiving a response is important for the success of an interactive system. Response time is a known problem in existing tools. The benchmarks show that the time is very short (milliseconds). While evaluating the performance of our implementation, we focused on the overhead we incur. We did not consider the time spent to satisfy the user request (e.g the time spent in the loop to check the array correctness in Figure 3.10), since this can take as much as needed, and is not part of our interface. We also did not consider memory overhead.

We created two benchmarks with the same behavior as the two case studies described[4]. We ran our benchmarks on the NCSA Linux Cluster Abe, which consists of dual socket quad core Intel 64 2.33 GHz nodes interconnected with infiniBand OFED 1.2, through the batch scheduling queue.[5] We used the net-linux ibverbs build of CHARM++ v6.0.1 (publicly available), compiled with gcc 3.4.6 and optimization -O3. The Python interpreter available was v2.5.2. We used the default CHARM++ timers which, for this platform, is gettimeofday.

	Execution time in ms							
#procs	1	2	4	8	16	32	64	128
no reuse	41	69	222	474	503	1904	2905	1844
with reuse	0.7	0.8	0.9	0.9	1.3	1.2	1.9	1.9

Table 3.1: Client request processing time results in milliseconds with varying number of processors. The Python script runs inside an interpreter connected to a chare group.

[4]The benchmarks are available as part of the CHARM++ distribution under the directory tests/charm++/python

[5]Data in this section was collected with help from Dr. Sayantan Chakravorty.

	Execution time in ms									
#calls	0	1	2	4	8	16	32	64	128	1000
time	0.14	0.16	0.17	0.19	0.22	0.31	0.46	0.77	1.48	10.18

Table 3.2: Time to execute the script with varying number of calls to the high-level interface.

The first benchmark creates a single chare where the Python requests are processed. The Python code contains a call to a high-level function. This C++ function broadcasts to all processors, which perform a certain amount of computation in parallel. The computation is defined by a simple loop with timer. At the end, a return value is reduced from all processors, and returned to the Python client, which prints it. The amount of computation performed by each processor in parallel is specified as an input parameter. We measured the time on the client, from when the Execute request is sent, until the ID of the interpreter used is returned back to the client. We made the Execute request wait for the completion of the Python code on the server. The total request time thus consists of (1) round-trip time of the message between the client and the server (within the same cluster); (2) creation of a user-level thread inside Charm++; (3) creation of a new Python interpreter by the server (optional); and (4) execution of the Python script itself. For each execution, we sent 30 requests to the same server.

	Execution time in ms							
#elements	1	4	16	25	100	400	2500	10000
total	10.04	39.95	160.5	248.3	1017	4010	25158	100926
per element	10.04	9.99	10.03	9.93	10.20	10.03	10.06	10.09

Table 3.3: Time to execute the script with varying number of elements over which to iterate.

	Execution time in ms							
size	8000					1000		
#chares	64	256	1024	4096	25600	100	400	10000
original	186	156	163	315	1144	9.8	67	297
with Python	188	155	159	312	1151	9.9	64	289

Table 3.4: Execution time of a 5-point 2D Jacobi application on a matrix with dimension *size*X*size* decomposed into *#chares* chares on 32 processors.

The results are shown in Figure 3.11 with varying number of processors and amount of computation performed by each processor. Dotted lines represent each request allocating a new Python interpreter on the server, while solid lines represent the same interpreter reused. The difference between corresponding lines show that the creation of a new Python interpreter (point 3) takes between forty to fifty milliseconds. This number is independent of the number of processors as expected, since the interpreter is created only on processor zero by a single chare. From the solid lines, by subtracting the amount of computation

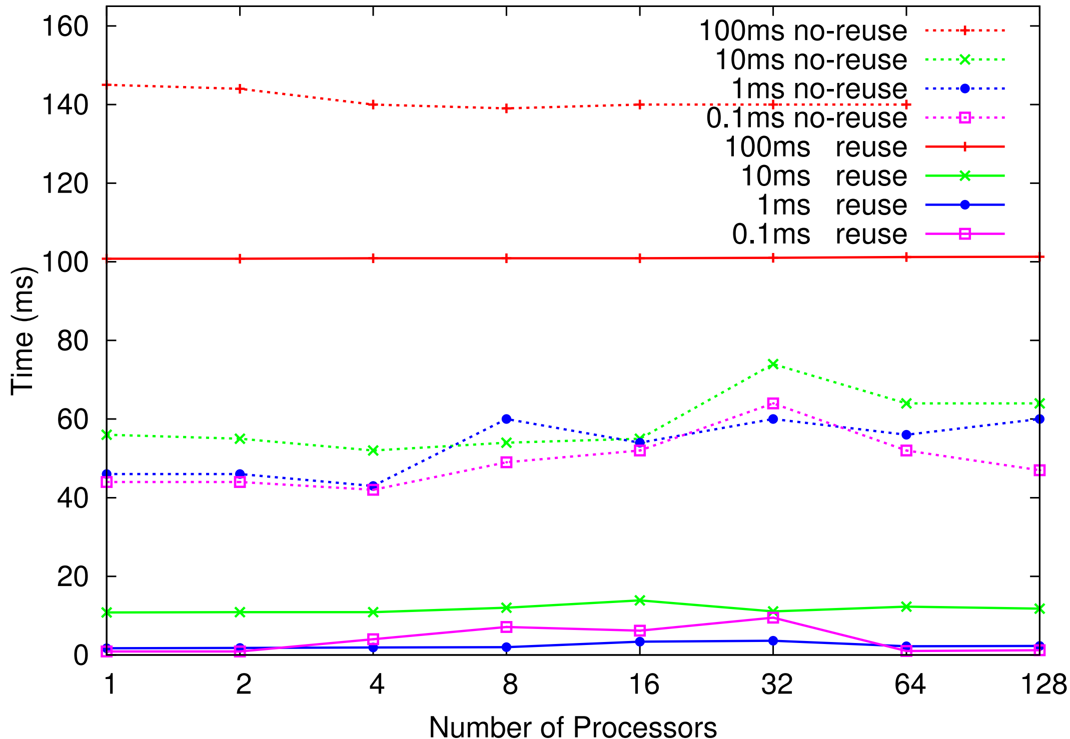


Figure 3.11: Client request processing time results in milliseconds with varying amount of computation (different color) and number of processors (X axis). The Python script runs inside an interpreter connected to a single chare. Dotted lines have a new interpreter created every request; solid lines have the same interpreter reused over multiple requests.

performed by the Python script which is known, we obtain the overhead of creating a new Python interpreter and a user-level thread. This is in the range of one to two milliseconds.

In the second benchmark, we used the CpdPythonGroup group, and sent a Python request to it. This request did not perform any work. Again we ran this test with varying number of processors, both creating new Python interpreters every request, and reusing the old one. Table 3.1 shows the results. As in the previous test, by reusing the same Python interpreter, we suffer only a few milliseconds of overhead. On the other hand, the overhead of creating new Python interpreters at every request grows to about two seconds for more than 32 processors. We do not understand this behavior completely, and we are still investigating it.

In all situations, having the client reuse the same Python interpreter for multiple requests reduces the overhead of the interface to below two milliseconds. This overhead can be tolerated both in a scenario of a user interactively writing code to upload, and in the scenario of a batch process uploading requests. Moreover, the performance results show that our implementation scales well up to at least 128 processors. This proves that our technique of uploading a high-level scripting language such as Python into a running parallel application is not only desirable, but also practicable.

Furthermore, we analyzed in greater detail the time spent to execute the Python script. We first tested the time taken to make a call from Python to Charm++. We used the second benchmark, and increased the amount of work performed by the Python script by adding calls to the charm.getValue method. We ran the benchmark on a single processor to avoid pollution from the parallel environment. We collected and averaged ten requests, excluding the first one. Since the Python script runs on each processor independently from the others, the results reflect on the multiprocessor case as well. Table 3.2 shows the time taken to execute the script with varying number of calls to the high-level interface. By linearly interpolating the results, we can see that one function call accounts for about 10μs. This value is independent of our implementation and depends on the Python/C library.

Secondly, we tested the overhead of the iterative interface to apply the same Python operation to multiple input elements (see Figure 3.7). We again used the second benchmark. Table 3.3 shows the results with varying number of elements over which iterating. It can be seen that the time scales linearly with the number of elements, therefore the overhead of repeatedly calling Python for each element is virtually zero.

Finally, we experimented with a real application to see the impact of repeatedly running Python scripts to check for application bugs. We used a 5-point 2D Jacobi application. Through CharmDebug, we installed a lightweight version of the code in Figure 3.10 stripped of the time consuming loop (since we are interested in the overhead only). This checking code ran after every message exchanged by the application (roughly four times the number of chares). We ran this on a 4-node Linux cluster, each node composed of dual socket quad core Intel Xeon 2.0 GHz, against the "original" program (which does not even contain the [python] keyword). Table 3.4 shows the performance results with varying amount of computation, determined by the matrix size, and granularity, determined by the number of chares. The overhead to link the Python interface and run the checking code is negligible in all scenarios, even in the extreme ones with thousands of chares.

3.6 Related Work

In the realm of parallel debugging, there are several tools that a programmer can use to understand why his program is misbehaving and correct the problem. Widely used commercial products are TotalView [2] from TotalView Technologies, and DDT [3] from Allinea. At least one of these tools is generally available in the majority of parallel supercomputers. Within the Open Source community, a tool worth mentioning is Eclipse [4]. Several Eclipse plugins have been developed to address parallel computing, in particular the Parallel Tools Platform (PTP) [40]. All these debuggers target applications written both in C/C++ and Fortran languages, and using MPI and/or OpenMP as programming models. None of them supports the Charm++ programming model natively. They all could manage Charm++ programs if Charm++ were built with MPI as its underlying communication layer. In this case, though, users would be exposed to the Charm++ implementation, rather than their own program.

Other tools performing dynamic insertion of code into a running application include

DynInst[41]. While an application is running, they allow an external program, called *mutator*, to attach to the running application, and modify its code image. After the image has been modified, the application will continue running the new code. This approach allows great flexibility in how the code is modified. Nevertheless, DynInst is not meant to be used directly by the user to write the new code, but through other tools that will simplify the modification process, which is otherwise tedious and potentially error prone. More recently, Dyner[42] has provided a TCL interface to the DynInst library to allow any modification to the user code in a simpler way. While this approach allows any modification at the source level, it does not provide the right level of abstraction for some kind of applications, like data analysis. Our aim is to allow the user to easily write a snippet of code to perform the desired operation while the application is running, and having it run immediately. In our approach the application developer retains the faculty to provide operations at the desired level of abstraction, and deny others that should not be used directly. This is an advantage for closed-source codes, where the user otherwise has to step down to the assembly level.

Other tools[43, 44] are used to patch non-stop applications to update them from one version to the following. These programs, like DynInst, provide low-level patching mechanisms, which again are not suitable for some kind of applications. Moreover, the patching mechanism is only for expert programmers, as the uploaded code is supposed to have passed all correctness tests.

GDB[23] provides the capability to inspect variables when the program is suspended at a breakpoint, as well as suspend execution when a condition is satisfied. A breakpoint in GDB can be set at any instruction line in the source code. While this is a powerful tool for debugging, if the condition is complicated, this approach might not be practical. For parallel distributed applications written in MPI, TotalView[2] can provide similar functionality. Again, if the checking code to run is complicated, writing it correctly might be challenging. It is agreed that scripting languages such as Python, Lua or Ruby are easier to use than programming languages like C/C++ or Fortran. In our approach, we focused on the usage of scripting languages to simplify the on-the-fly writing of checking code.

On the topic of introspection within application written in C++, many tools have been built[45, 46, 47, 48, 49]. The main scope of these works is to provide the program itself access to its data types. In our approach, we used the information already collected by CharmDebug to provide this capability. Nevertheless, these other approaches are also viable implementations, and might be considered in future work.

3.7 Future Work

In addition to the interface illustrated in this chapter to allow a user to update Python scripts into a running parallel operation to perform correctness checks, interfaces for other programming languages could be provided. For example, a similar approach could be taken to dynamically insert a C/C++ piece of code to perform the same operation. The user could be unfamiliar with the Python language and prefer something he is more familiar with. One aspect to consider would be the operation that the inserted checking code should be allowed

to perform. While for Python scripts the interaction with the main code can be controlled to some extent via the programmed interface, with C/C++ code this would be more difficult.

As described towards the beginning of the chapter, as applications need to be debugged on larger and larger machines, even simple operations can be prohibitive on a sequential client if they entail a computation proportional to the number of processors allocated. A possible direction is to maintain only collective information on the client, in addition to what is being displayed. Instead, any additional information would reside on the parallel machine, and be gathered back to the client only upon request. This solution would highly scalable in the long term since it would allow to continue scaling the debugger together with the application itself, without additional resource allocation to the debugging infrastructure itself. However, there may be some information that can be required often, and having to always gather it from the remote application can decrease the responsiveness of the debugger. An example of such information is the status of each processor in the application. For this and other basic information, the client could store and process them using more than a single processor. As workstations and even laptops are moving towards multicore architectures, this approach is more feasible.

4 Virtualized Debugging

Debugging a parallel application requires numerous iterative steps. Initially, the application is tested on simple benchmarks on a few processors. This can already capture many errors due to the communication exchanges between the processes. Later, during production runs, the application will be deployed with larger input datasets, and on much bigger configurations. Oftentimes, in this new scenario, the application will not behave as expected, and will terminate abnormally. When this happens, the programmer is left to hunt the problem at the scale where it manifests, with possibly thousands of processors involved. If lucky, he may be able to recreate the problem on a smaller scale and debug it on a local cluster, but this is not always possible.

One example of a bug that may not be reproduced on a smaller scale is when the bug is located in an algorithm, and this algorithm depends on how the input data is partitioned among the processors. Reducing the problem size might be a solution to scale down the problem, but the inherent physics of the problem may not allow that. Another example is when the physics simulation output is incorrect. In this case, the problem can derive from rare conditions that only big datasets expose. Again, the problem size may not be reduced since otherwise the bug disappears. In all these examples, the only alternative left to the programmer is to use the whole machine, and debug with the full problem size on possibly thousands of processors.

Interactive sessions on large parallel machines are usually restricted to small allocations. For large allocations, batch scheduling is often required. To debug the application, the programmer will have to launch the job through the scheduler and be in front of the terminal when the job starts. Unless a specific allocation slot is pre-requested, this can happen at unpredictable, inconvenient times. Furthermore, the nature of debugging is such that it may require multiple executions of the code to track the bug, and to try different potential solutions. This exacerbates the problem and leads to highly inefficient debugging experience.

Moreover, debugging sessions on a large number of processors are likely to consume a lot of allocation time on supercomputers, and significantly waste precious computation time. During an interactive debugging session, the programmer usually lets the program execute for some time and then pauses it to inspect its data structures, then iteratively advances it step-by-step, while monitoring some data of interest. Therefore, processors are idle most of the time waiting for the user to make a decision on what to do next, which is a very inefficient use of supercomputers.

The innovative approach we describe in this research is to enable programmers to perform the interactive debugging of their applications at full scale on a simulated target machine using much smaller clusters [50]. We do this by making each processor in the application

a *virtual processor*, and mapping multiple virtual processors to a single physical processor. This reduces the processor count needed for debugging. This mapping is transparent to the application, and only the underlying runtime system needs to be aware of the virtualization layer. A parallel debugger connected to the running application presents to the programmer the vision of the application running on thousands of processors, while hiding the fact that maybe only a few dozen were actually used.

Our idea transcends the programming model used for the virtualization and how the debugging infrastructure is implemented. The only important component is a parallel runtime system adapted to support virtualization of processing entities. To prove the feasibility of this approach, we implemented it within the CHARM++ runtime system, using the BigSim emulation environment and the CHARMDEBUG debugger. Thus, applications written in CHARM++ will be the main target for our debugging examples. MPI applications are supported via AMPI [15], a virtualized implementation of the MPI standard.

4.1 Related Work

As mentioned in Section 3.6, several tools exist to debug parallel programs on large machines. However, they all require to allocate the whole set of processors used for debugging. If the users desires to perform his debugging using one hundred thousand processors, then a big machine has to be used and occupied for long periods of time for the debugging to happen. This alone can hinder the capability to debug a large scale application.

Virtualization for High Performance Computing has been claimed to be important [51]. Nevertheless, no tool known to the author provides a debugging environment tailored to thousands of processors or more, while utilizing only the few processors that a local cluster can provide. A few techniques have been developed in contexts other than High Performance Computing leveraging the concept of virtualization. These target the debugging of embedded systems [52], distributed systems [53], or entire operating systems using time-travel techniques [54, 55]. All of them target virtual machines (such as Xen [56] or IBM Hypervisor [57]) where the entire operating system is virtualized. Using virtual machines may pose problems for a normal user as the installation and configuration of such virtual environments require administration privileges, and most supercomputers do not provide them by default. Our technique, as we shall see, resides instead entirely in the user space, and does not suffer from this limitation.

4.2 Debugging Charm++ Applications on BigSim

In order to combine the BigSim emulation system described in Section 2.4 with the CHARMDEBUG debugging framework, several new problems had to be solved. Most arose from the fact that CHARMDEBUG needs to deal with the virtualized CHARM++ and other virtualized layers in the emulation environment.

Normally, CHARM++ is implemented directly on top of CONVERSE, which is respon-

sible for low-level machine-dependent capabilities such as messaging, user-level threads, in addition to message-driven scheduling. This is shown on the left branch of Figure 4.1. When CHARM++ is re-targeted to the BigSim Emulator, there are multiple target CHARM++ virtual processors running on one physical processor, as explained in the previous section. Therefore, all layers underneath CHARM++ must be virtualized. This new software stack is shown in the same Figure 4.1, on the right branch. Specifically, the virtualized CONVERSE layer becomes BigSim CONVERSE, which is the CONVERSE system implemented using the BigSim Emulator as communication infrastructure. This is equivalent to treating the BigSim Emulator as a communication sub-system.

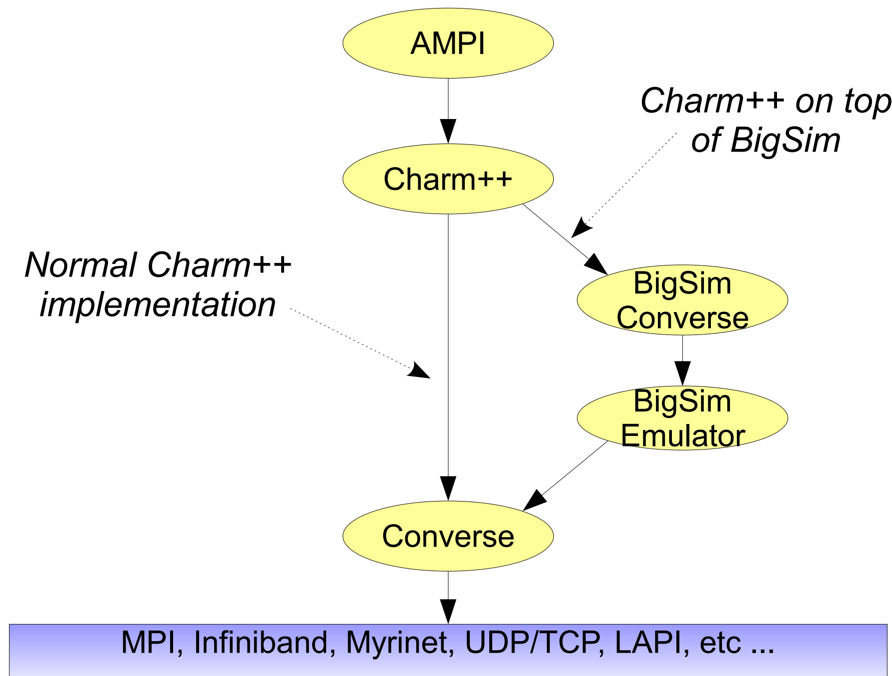


Figure 4.1: BigSim Charm++ Software Stack.

4.2.1 Communicating with Virtual Processors

One problem we had to overcome was the integration of the CCS framework into BigSim. CCS connects CHARMDEBUG and a running application considering each operating system process as an individual CHARM++ processor. However, in the BigSim Emulation environment, CCS is unaware of the emulated virtual processors because it is implemented directly on CONVERSE. Therefore, it needs to be adapted to the emulation system so that the CHARMDEBUG client can connect to the emulated virtual processors. To achieve this, we created a middle layer for CCS (virtualized CCS) so that messages can reach the destination virtual processor. The target of a CCS message becomes now the rank in the virtual processor space. Figure 4.2 depicts the new control flow.

When a CCS request message is sent from CharmDebug to a virtual processor, the message first reaches the CCS host (1). From here, it is routed to the real processor where the destination virtual processor resides (2). The processor level scheduler in Converse will pick up the request message, but not execute the message immediately. Instead, it enqueues the message to the corresponding virtual node, and activates it (3). The scheduler on the virtual node will serve the CCS request by invoking the function associated with the request message (4), and return a response message. Notice that the response does not need intervention from Converse since the virtual processor has direct access to the data structures stored in the common address space. Multicast and broadcast requests are treated in the virtualized environment. While this can add some overhead to the execution of a CCS request, it greatly simplifies the system and the code reuse between the emulated and non-emulated mode.

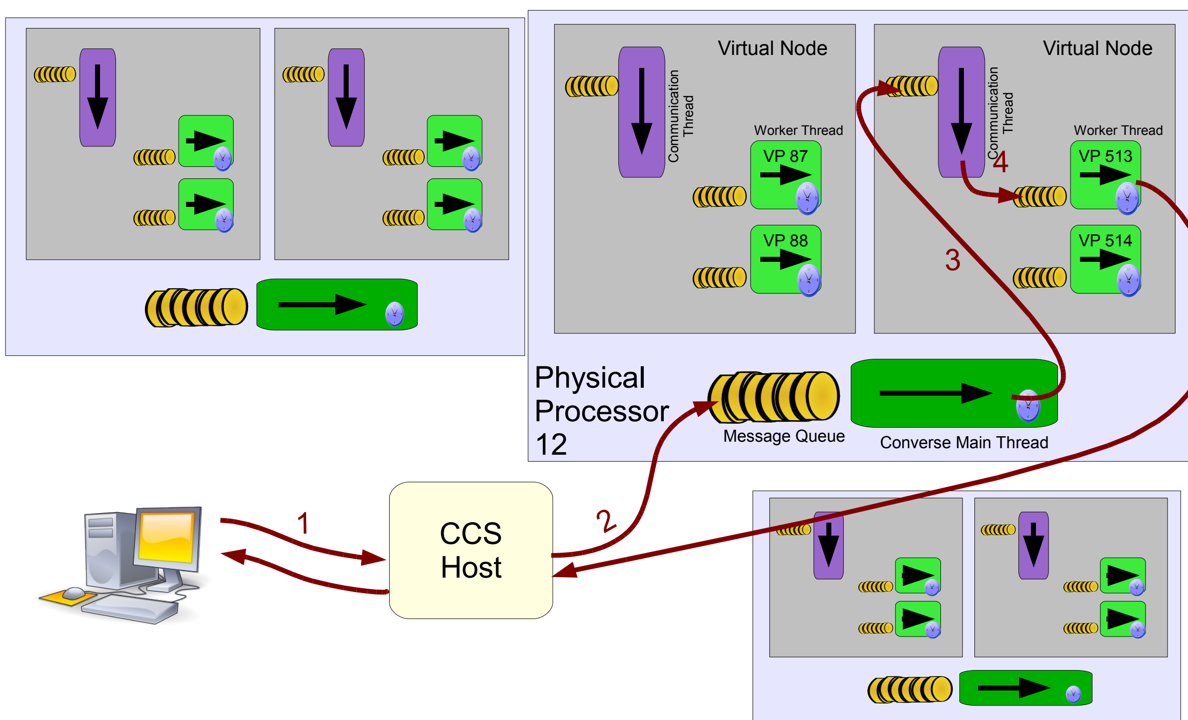


Figure 4.2: Diagram of CCS scheme under BigSim Emulation.

Some CCS request messages are not bound to any specific virtual processor. For example, CharmDebug may send CCS requests to physical processors to query processor-wide information such as those related to the system architecture or the memory system. However, since all virtual processors on the same physical processor have access to the processor information including the whole memory, any of these virtual processors can, in fact, serve the CCS requests. Therefore, our approach is to have CharmDebug client always send such CCS requests to a virtual processor on a physical processor. This approach greatly simplifies the design and implementation of the CCS protocol, since we eliminate the need

of having to specify if the request needs to be treated at the physical processor level, or at the virtual processor level.

4.2.2 Suspending Virtual Processors

Another challenge was to figure out how to suspend the execution of a single virtual processor. Notice that while a processor is suspended, we still want to deliver messages to it. For example, requests from the debugger should be honored regardless of the processor's state. At the same time, we do not want other virtual processors emulated inside the same physical processor to be affected. In the non-virtualized environment, the technique we use to suspend a processor is to enter a special scheduler when the processor needs to be suspended. In this mode, regular messages are placed into a queue, and buffered in FIFO order until the processor is allowed to handle them. This scheduler is also in charge of driving the network, and receiving incoming messages. In this way, commands from the debugger can still be executed. In the virtualized environment, the scheduler that drives the network and forwards messages to the virtual processes is a separate entity from the scheduler inside each virtual processor. In this case, it is not possible to have each virtual processor driving the network, which will be too chaotic.

We modified our scheme to move the buffering of messages inside each individual virtual processor. When a worker processor needs to suspend due to an explicit debugger "freeze" command or due to a breakpoint, it calls its own scheduler recursively. Since this scheduler is stateless, such a recursive scheme is feasible. This new scheduler then starts the buffering of messages. When the processor is released by the debugger, and is allowed to continue its normal execution, we terminate the internal scheduler, and return control to the outer one. Buffered messages are guaranteed to be executed in the same order as they were received while we exit from the internal scheduler. Meanwhile, the main CONVERSE scheduler remains the only one that drives the network and receives messages. Moreover, the CONVERSE scheduler is always active, and never enters a buffering mode.

With the techniques described, we can now debug applications in the virtualized environment as if they were running on a real machine. We shall see an example of using CHARMDEBUG on a real application in Section 4.5.

4.3 Debugging MPI Applications on BigSim

Debugging a large scale MPI application on a smaller machine requires running multiple MPI "processes" on one processor. This can be done using existing MPI implementations, if allowed by the operating system. However, this is often infeasible for various reasons. First, operating systems often impose hard limits on the total number of processes allowed by a user on one processor, making it challenging to debug a very large scale application. Secondly, processes are heavy-weight in terms of creation and context switching. Finally, there are very few MPI implementations that support out-of-core execution, which is needed for running applications with large memory footprints.

To overcome these challenges, we adopted the same idea of processor virtualization used in CHARM++: each MPI rank is now a virtual processor implemented as a *light-weight* CONVERSE user-level thread. This leads to Adaptive MPI, or AMPI [15], an implementation of the MPI standard on top of CHARM++. As illustrated in Figure 4.3, each physical processor can host a number of MPI virtual processors (or AMPI threads). These AMPI threads communicate via the underlying CHARM++ and CONVERSE layers. This implementation also takes advantage of CHARM++'s out-of-core execution capability. Since AMPI is a multithreaded implementation of the MPI standard, global variables in MPI applications may be an issue. AMPI provides a few solutions to automatically handle global variables [58] to ensure that an MPI application compiled against AMPI libraries runs correctly.

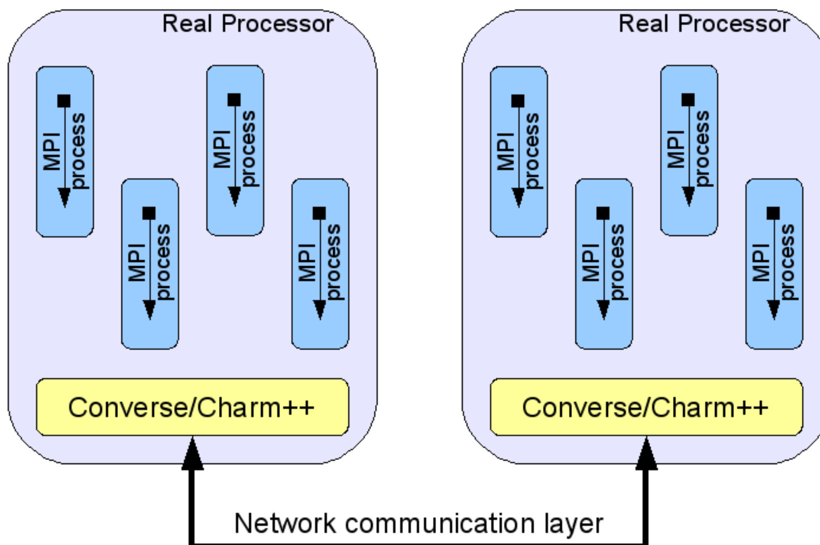


Figure 4.3: AMPI model of virtualization of MPI processes using CHARM++.

Debugging MPI applications can now use any arbitrary number of physical processors. For example, when debugging Rocstar [59], a rocket simulation program in MPI developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois, a developer was faced with an error in mesh motion that only appeared when a particular problem was partitioned for 480 processors. Therefore, he needed to run the application on a large cluster at a supercomputer center to find and fix the bug. However, the turn-around time for a 480 processor batch job was fairly long since the batch queue was quite busy at that time, which made the debugging process painfully slow. Using AMPI, the developer was able to debug the program interactively, using 480 virtual processors distributed over 32 physical processors of a *local* cluster, where he could easily make as many runs as he wanted to resolve the bug.

Since AMPI is implemented on top of CHARM++, the basic techniques for debugging as described in Section 4.2 work on AMPI programs automatically. In addition, if the user

```
Node 4
(gdb) c
Continuing.

Breakpoint 2, AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60,
    displs=0x95c360, sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1,
    recvtype=1, root=7, comm=9000000) at ampi.C:4215
4215        AMPIAPI("AMPI_Scatterv");
(gdb) bt
#0  AMPI_Scatterv (sendbuf=0x959680, sendcounts=0x94bf60, displs=0x95c360,
    sendtype=1, recvbuf=0x3fbd400ffe50, recvcount=1, recvtype=1, root=7,
    comm=9000000) at ampi.C:4215
#1  0x00000000004a5c04 in MPI_Tester::test (this=0x3fbd400ffec0) at test.C:191
#2  0x00000000004a6064 in AMPI_Main_cpp (argc=1, argv=0x8e6340) at test.C:279
#3  0x00000000004c5945 in AMPI_Fallback_Main (argc=1, argv=0x8e6340)
    at ampi.C:401
#4  0x00000000004d5dfa in MPI_threadstart_t::start (this=0x3fbd400fff70)
    at ampi.C:552
#5  0x00000000004c5619 in AMPI_threadstart (data=0x916838) at ampi.C:569
#6  0x00000000004e41c2 in startTCharmThread (msg=0x916820) at tcharm.C:138
#7  0x00000000004f09ce in CthStartThread (fn1=0, fn2=5128596, arg1=0,
    arg2=9529376) at threads.c:1579
#8  0x00007f7a47330c20 in ?? () from /lib/libc.so.6
#9  0x0000000000000000 in ?? ()
(gdb)
```

Figure 4.4: Screenshot of GDB attached to a specific MPI rank, and displaying its stack trace.

desires to perform more in-depth analysis on a specific MPI rank, he can choose to start a GDB sequential debugger attached to the processor hosting that rank, and focus on the desired rank. This GDB process is shown in Figure 4.4 for a simple test program. In this example, the user has set a breakpoint on MPI_Scatterv function, and when the breakpoint was hit, he printed the stack trace.

4.4 Debugging Overhead in the Virtualized Environment

In this section, we study the debugging overhead using a synthetic Jacobi benchmark and a real application, NAMD, running on the modified BigSim emulator with CharmDebug support.[1]

Our test environment is Blue Print, a Blue Waters interim system at National Center for Supercomputing Applications (NCSA). It is an IBM Power 5+ system. There are 107 compute nodes actually available for running a job, and each node has 16 cores (i.e. 1712 cores total).

We first tested a Jacobi3D program written in Charm++ on 1024 virtual processors on a varying number of physical processors with CharmDebug enabled, and measured the

[1]Data in this section was collected with help from Dr. Gengbin Zheng.

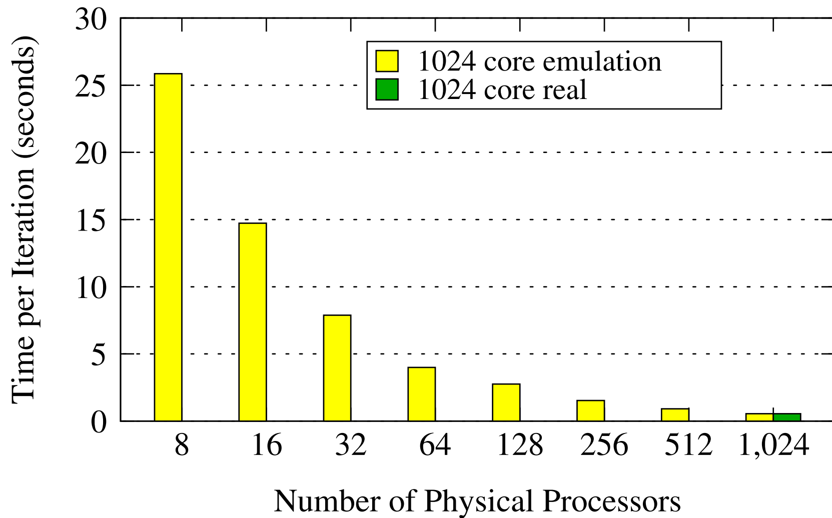


Figure 4.5: Jacobi3D execution time on 1024 emulated processors using varying number of physical processors. The last bar is the actual runtime on 1024 processors.

execution time per step. Figure 4.5 shows the results of the execution time with varying number of physical processors, from 8 to 1024. The last bar in the figure is the actual execution time of the same code on the 1024 processors with normal CHARM++. We can see that by using exactly same number of processors, Jacobi under BigSim emulator runs as fast as the actual execution in normal CHARM++, showing almost no overhead of the virtualization in BigSim. When we use fewer physical processors to run the same Jacobi emulation on 1024 virtual processors, the total execution time increases as expected. However, the increase in the execution time is a little less than the time proportional to the loss of processors. For example, when using 1024 physical processors, the execution time is 0.25s, while it takes only 23.96s when using only 8 physical processors. That is about 92 times slower (using 128 times fewer processors). This is largely due to the fact that most communication becomes in-node communication when using fewer processors.

As a stress test, we ran the same Jacobi3D program on one million (1,048,576) emulated processors, while trying to use as fewer number of physical processors as possible. Figure 4.6 shows the execution time when running on 400, 800, and 1712 physical processors. These experiments show that it is feasible to debug an application in a virtualized environment for very large number of target processors using a much smaller machine.

To test how much time typical operations take from the debugger point of view, we used a similar Jacobi3D program, this time written in MPI. Table 4.1 reports timings for starting the MPI application, loading the list of messages queued on a given processor, and perform a step operation (deliver a single message) on all virtual processors. The latter two operations perform in an almost identical amount of time in all scenarios, including the case when the application is run in the non-virtualized environment.

We also studied the BigSim overhead on a real application. NAMD [16, 26] is a scalable

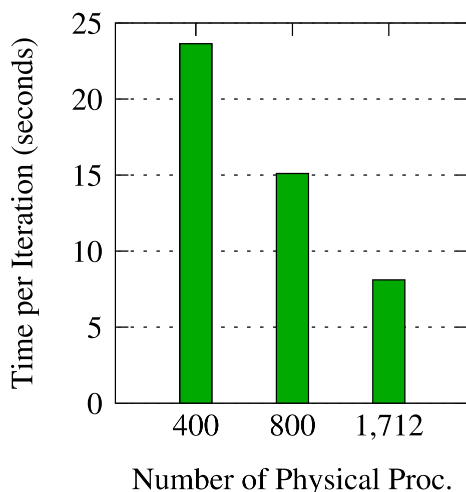


Figure 4.6: Jacobi3D execution time on 1M (1,048,576) emulated processors using varying number of physical processors.

parallel application for Molecular Dynamics simulations written using the Charm++ programming model. It is used for the simulation of biomolecules to understand their structure. In these experiments, we ran NAMD on 1024 emulated processors with Apolipoprotein-A1 (ApoA1) benchmark for 100 timesteps. We measured the total execution time of each run (including startup and I/O) using a varying number of physical processors, from 8 to 1024. This is illustrated in Figure 4.7. Same as for Jacobi, we ran NAMD also in non-emulated mode using 1024 physical processors. The total execution time is shown in the last bar of the figure. We can see that NAMD running on the BigSim Emulator is only marginally slower (by 6%) compared to the normal execution on 1024 physical processors, showing little overhead of the emulator. On 512 processors, however, NAMD running in the emulation mode is even slightly faster than the actual run on 1024 processors. This is due to savings in the NAMD initial computation phases: faster global synchronization on fewer nodes.

Overall, this demonstrates that in terms of the time cost, debugging in a virtualized environment using much smaller number of processors is possible. Although it takes a longer time (19 times slower from 1024 to 8 processors) to run the application, debugging on a much smaller machine under a realistic scenario is not only easily accessible and convenient, but also simpler for setting up debugging sessions.

We further studied the memory overhead under the virtualized environment. Using the same NAMD benchmark on 1024 virtual processors, we gathered memory usage information for each processor. Figure 4.8 shows the peak memory usages across all physical processors. Again, the last bar is with the non-emulated Charm++. Note that in emulation mode, the total memory usage is the sum of the application's memory usage across all emulated processors, plus the memory used by the emulator itself. It can be seen that there is no difference in memory usage between the emulation mode and non-emulation mode when using 1024 physical processors. When the number of processors decreases to 512, or even

	Startup (seconds)	Load a message queue (ms)	Single step, all pe (ms)
8	11.60	398	131
16	11.63	399	99
32	13.34	399	213
64	13.12	400	66
128	15.86	400	41
256	14.41	399	118
512	16.45	399	67
1024	17.71	379	118
original	17.85	379	114

Table 4.1: Time taken by the CharmDebug debugger to perform typical operations, using MPI Jacobi3D application with 1024 emulated processors on varying number of physical processors.

256, the memory usage remains about the same. This is because NAMD has some constant memory consumption that dominates the memory usage (for example, read-only global data such as molecule database, which is replicated on each node), and the emulator itself tends to use less memory when the number of processors decreases. However, when the number of physical processors keeps reducing, each physical processor hosts a much larger number of emulated virtual processors whose memory usage starts to dominate, therefore the total memory usage increases significantly. Nevertheless, when the number of physical processors is down to 8, the peak memory usage reaches about 1GB, which is still very feasible on machines nowadays. Note that this is an increase of only about 7 fold compared to the 1024 processor case, due to the sharing of the global read-only data at the process level.

In summary, we have demonstrated that debugging under virtualized environment incurs reasonably low overhead, considering the overhead proportional to the loss of processors. This makes it feasible to debug applications running on a large machine using only a portion of it.

4.5 Case Study

To demonstrate the capabilities of our technique, we used a few examples of complex applications, and debugged them in the virtualized environment. It is not the purpose of this section to describe actual bugs that were found with this technique, but rather illustrate how the user has available all the tools that he has in a normal scenario. With those tools, the user can search for the bug as he seems fit. Some applications have been described in Section 4.4 while considering the overhead our technique imposes to the application under debugging. In this section, we use another real world application as an example.

ChaNGa [18] is a production code for the simulation of cosmological evolution, currently in its second release. It is capable of computing standard gravitational and hydrodynamic

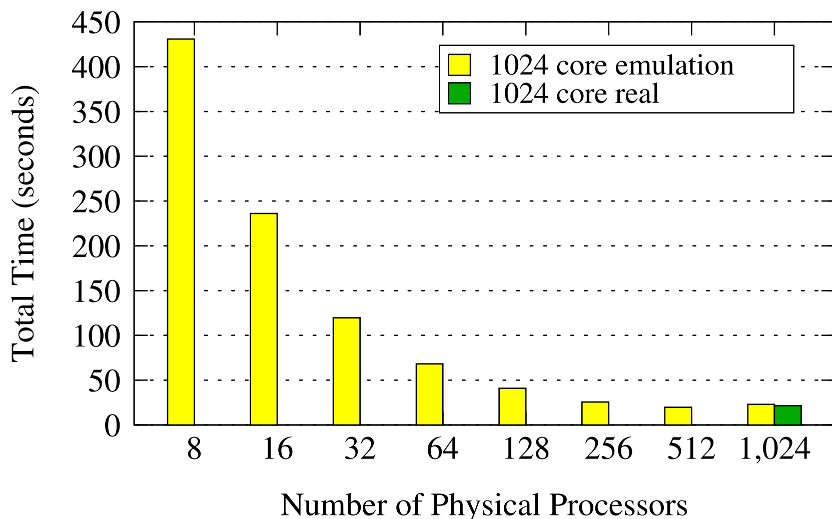


Figure 4.7: NAMD execution time on 1024 emulated processors using varying number of physical processors. The last bar is the actual runtime on 1024 processors.

forces using Barnes-Hut and SPH approaches respectively. This application is natively written in CHARM++, and it uses most of the language abstractions provided by the runtime system. While most of the computation is performed by CHARM++ *array elements*, which are not bound to the number of processors involved in the simulation, the application also uses CHARM++ *groups* and *nodegroups* for performance reasons. The groups have the characteristic of having one entity per processor, thus modifying the application behavior when scaling to larger number of processors. The complexity of this application is one reason why we chose it over other examples.

After the user has built the CHARM++ runtime system with support for BigSim emulation and compiled the CHANGA program over the virtualized CHARM++, he can start CHARMDEBUG's GUI. Figure 4.9(a) shows the dialogue box for the application parameters. In here, the user will indicate the location of his executable, the arguments, and the number of processors he wants to run on. The only difference from a standard non-virtualized execution is the presence of a checkbox to enable the virtualization. In general, the user will input the number of desired processors in the "Number of Processors" textfield and confirm. In this case, "Number of Processors" refers to the number of physical processors CHARMDEBUG will allocate on the machine. The number of processors the user wants to debug on has to be specified in the field named "Number of Virtual Processors". These fields are highlighted in the Figure. At this point the user can confirm the parameters, and start the execution of the program from CHARMDEBUG's main view.

If the machine to be used for debugging requires jobs to be submitted through a batch scheduler (or if the user desires to start the application himself), only the fields regarding executable location and CCS host/port connection need to be specified. These are highlighted in Figure 4.9(b). When the attach command is issued from the main view, the CHARMDEBUG plugin will automatically detect the number of processors in the simulation,

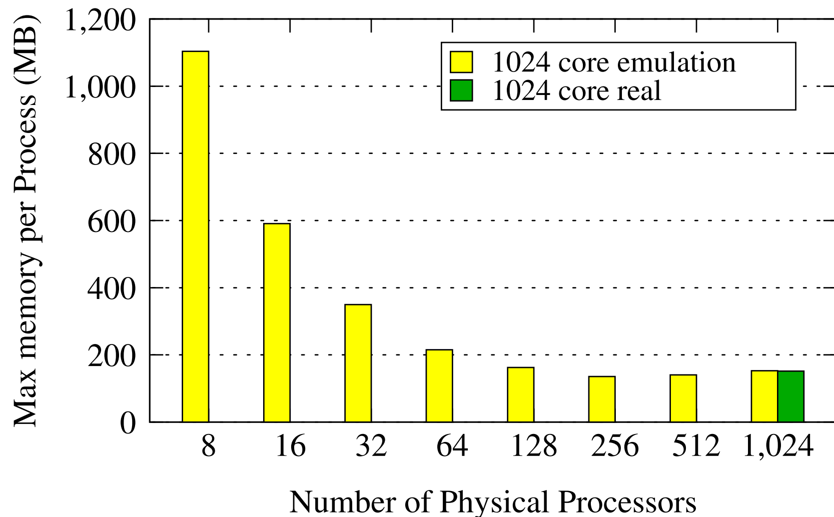


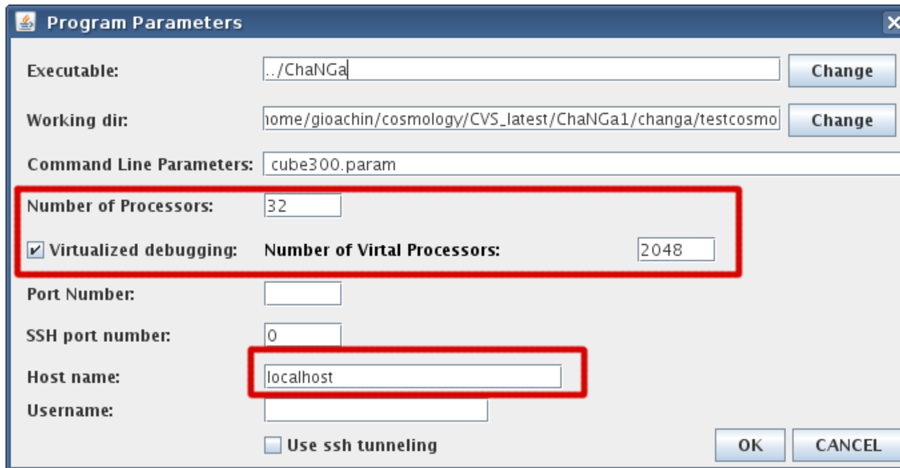
Figure 4.8: NAMD memory usage per process on 1024 physical processors vs. NAMD on emulation mode using from 8 to 1024 physical processors.

and if the execution is happening in the virtualized environment.

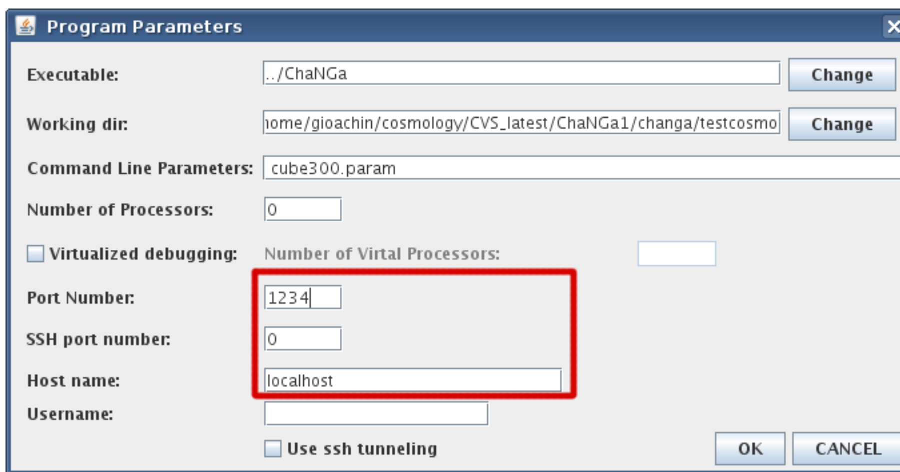
Once the program has been started, and CharmDebug has connected to it, the user can perform his desired debugging steps, oblivious of the fact that the system in using fewer resources internally. Figure 4.10 shows the ChaNGa application loaded onto four thousand virtual processors. Underneath, we allocated only 32 processors from four local dual quad-core machines. In the bottom left part of the view, we can see all the messages that are enqueued in the selected processor (processor 3,487 in the Figure). Some messages have a breakpoint set (7th message, in orange), and one has actually hit the breakpoint (1st message, in red). In the same message list, we can see that some messages have as destination "TreePiece" (a Charm++ array element), while others have as destination "CkCacheManager", one of the groups mentioned earlier. One such message is further expanded in the bottom right portion of the view (10th message).

Finally, Figure 4.11 illustrates how normal operations, such as inspection of an object in the system, is still available to the user. In this case, we are inspecting the "TreePiece" element number 5,213 present on (virtual) processor 2,606. Again, this operation interacts with the runtime system and reports the same information to the user as if the application were actually running on the whole four thousand processors. As expected, given the architectural design, the response time was of the order of tens to hundreds milliseconds, mostly depending on network latency and GUI overhead.

When joining multiple processes inside the same address space, the behavior of the system might be altered. First of all, one virtual processor could corrupt the memory belonging to another one. A solution to this problem is the topic of the next chapter. Another problem regards the kind of bugs that can be detected. In particular, race conditions may become difficult to treat: by reducing the amount of physical processors available, the communication



(a) Launching scenario



(b) Attach scenario

Figure 4.9: Screenshots of CharmDebug parameter window.

latency might change such that a race condition will not appear anymore. Two solutions are possible. One is to use record-replay techniques to force the execution of a particular message ordering. This is available in the virtualized environment, and will be described in detail in Section 6.6. The other possibility is to force the delivery of messages in the virtualized environment in a different order. The foundations to this approach will be presented in Chapter 7.

4.6 Future Work

In this chapter, we presented an innovative technique to address the issue of debugging applications on very large number of processors without consuming a large amount of resources. This is possible by having the runtime system emulate the existence of processors allocated

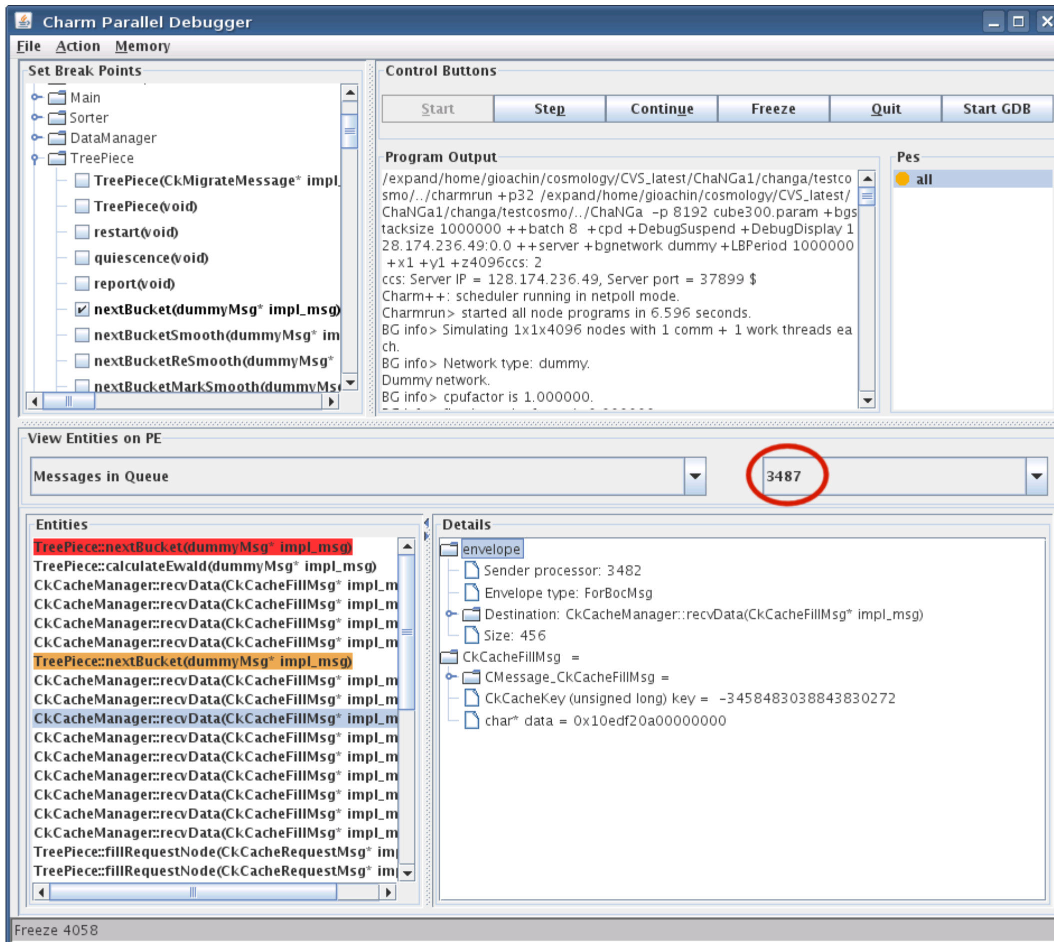


Figure 4.10: Screenshot of ChaNGa debugged on 4,096 virtual processors using 32 real processors.

to the user inside the user space. This technique is already powerful in its current state. Nevertheless, further improvements are possible to make its use more seamless and more widely used.

With the co-existence of multiple virtual processors inside the single address space of a physical processor, some memory operations have been disabled. For example, searching for memory leaks requires the debugger to disambiguate which virtual processor allocated the memory. One approach would be to use the same memory tagging mechanism described in the next chapter, and cluster memory allocation by virtual processor. The tools inside the runtime system that perform operation directly on allocated memory would then need to be modified to consider only a portion of the address space.

Another future work regards MPI. As we described in Section 4.3, currently CharmDebug focuses primarily on applications written in Charm++. While it can debug MPI applications using the AMPI implementation of the MPI standard, we realize that for a programmer debugging his application there may be unnecessary overhead. For the future, we are consid-

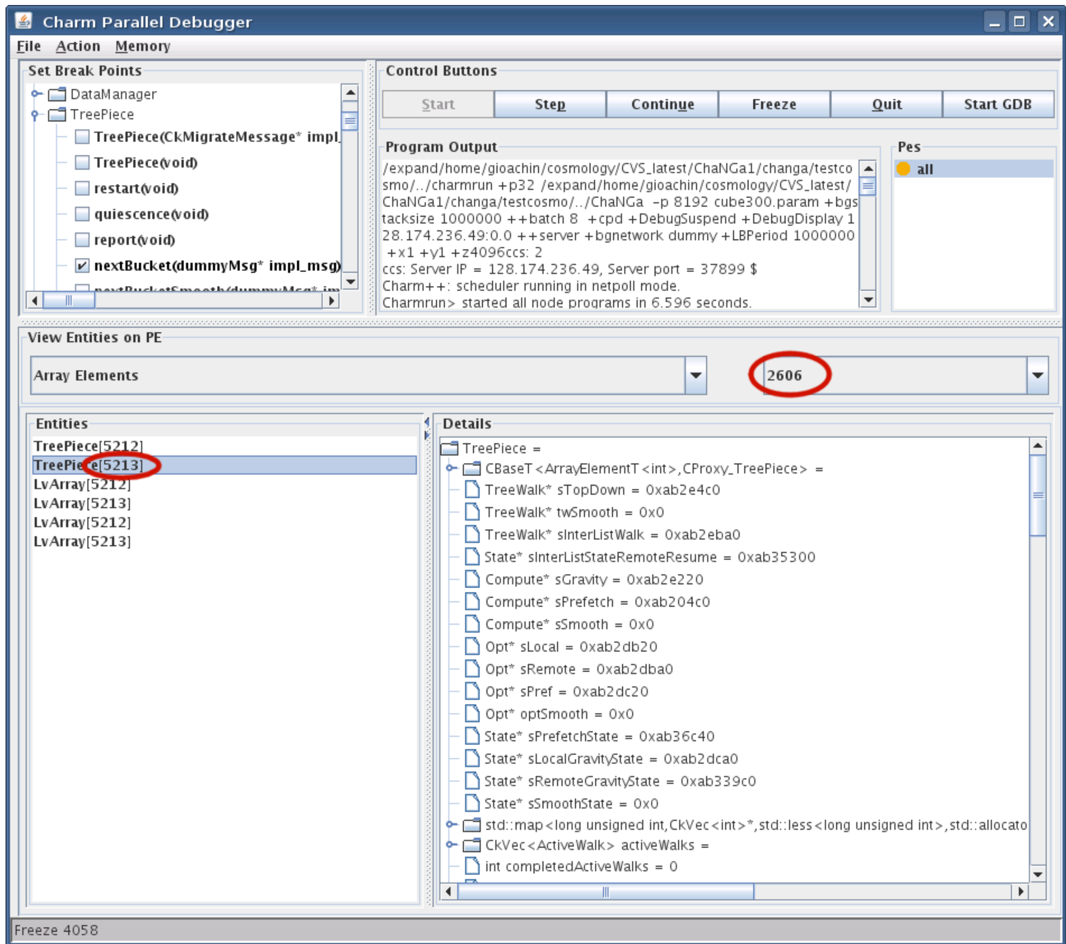


Figure 4.11: Inspection of a Jacobi object in the system.

ering possible extensions to provide a more natural debugging also for MPI programs. There are two directions worth considering. One is the integration of MPI-related techniques directly inside CharmDebug, making it natively display MPI-related information. The other is to provide a wrapper API to the Charm++ runtime system so that other debuggers that already support MPI will be able to access the information desired.

5 Isolating Objects

When allocating multiple processors inside a single address space, as explained in the previous chapter, these virtual processors will share the same address space. In particular, they could end up sharing the same global variable if the user was not careful enough to avoid them. When accessing the global variable, the different processors will inherently share the same data.[1] In particular, if this variable is a pointer, they will access and modify the same data structures. Note that global variables in MPI applications are allowed, and AMPI treats them using several mechanisms [58], but in CHARM++ applications, the user should not use global variable, and should instead place all the data inside the pertinent chare.

This problem is more general than the case of multiple virtual processors sharing the same address space. For example, when an application is decomposed into independent modules that share the same executable, it is possible to have conflicts in how the memory is used by the different modules. In a correct decomposition, each module stores the state necessary to perform its task in memory, and uses some specific area to exchange information with other modules. Nevertheless, since the virtual address space is common, all the modules have access to the entire address space. In particular, one module can modify the state of another module, accidentally or intentionally. While this cannot be prevented during a normal execution, it breaks the abstraction that modules are independent, and makes a faulty module difficult to identify.

Consider the parallel program for the simulation of galaxy formation, described in Section 2.1, composed of multiple separate modules to compute the various forces present in the universe. All these modules will need to update the memory storing the final forces acting on the simulated portion of space. This memory is therefore used to exchange information. In addition, each module will also have its private data to be used during the computation phase. This private data should be modified exclusively by the owner module. If, for example, the gravity module accesses and modifies the data stored for the SPH computation, we want to notify the user that the gravity module is misbehaving, and might be faulty.

Both virtual processors discussed in the previous chapter, and the modules just described, can be considered "objects" inside the system. Throughout this chapter, the term object will be used to indicate either of them. By creating a tagging system where each memory allocated block is marked with an identifier of the object that uses it, it becomes possible to intercept modifications of such memory by other objects. The user can be notified of these misuses and can determine if they are valid, such as in the case of the final acceleration in

[1] Part of the work in this chapter is reprinted, with permission, from "Memory Tagging in Charm++", in the Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08), ©ACM, 2008. http://doi.acm.org/10.1145/1390841.1390845 [60]

the previous example, or if they are not, and therefore identify the faulty entity. In this chapter, we shall build upon the existing Charm++ framework to provide a mechanism to detect cross-object memory corruption.

5.1 The Charm++ Memory Subsystem

In Charm++, the memory subsystem, i.e the implementation of the "malloc", "free" and other memory related functions, is included in a shared library. This allows Charm++ to implement multiple versions of the memory library, and enables the user to choose which one to use at link-time, see Figure 5.1. The default version, gnu.o in figure, does not have any debugging support and is meant for production usage. This version is based on the glibc memory allocator. Another memory library, os.o, does not implement the memory functions, and lets the user link to the default one provided by the operating system. All the others are based on the glibc standard library, as the default implementation, but they re-implement the memory functions (malloc, free, realloc, etc) and use the glibc ones internally.

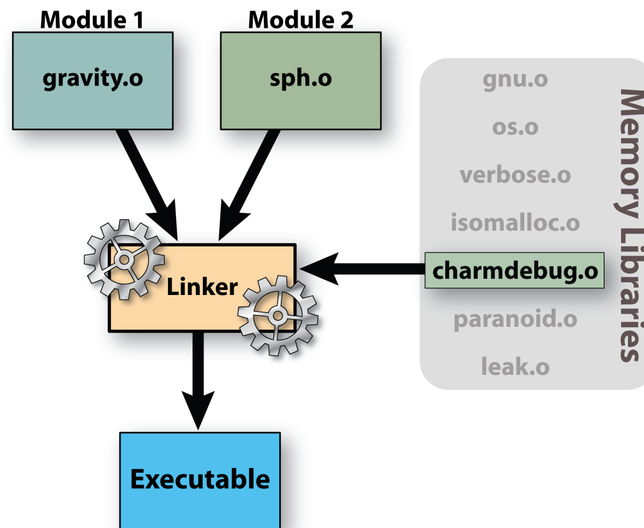


Figure 5.1: Application linking stage. The application modules are linked together with one Charm++ memory implementation to produce the executable.

To allow multiple memory libraries with different capabilities to be based on the same underlying glibc memory allocator while avoiding the problem of re-implementing the entire allocator, the glibc routines have been renamed prepending them with a "mm_" prefix. Any memory library wrapping around glibc allocator will define the functions malloc, free, etc, and internally use the "mm_" versions. For example, the default malloc simply calls mm_malloc, while another malloc implementation can decide to allocate extra size for internal usage of the library itself, or to fill the newly allocated memory with certain patterns.

Some of the existing memory implementations inside CHARM++ are named in Figure 5.1. *Paranoid* provides buffer overflow detection by allocating extra space at both sides of the user allocation and filling it with a predefined pattern. At deallocation, these extra spaces are checked for modifications. Moreover, deallocated regions are also filled to help detect usage of dangling pointers. *Leak* allows the user to mark all allocated blocks as clean, and later performs a scan to see if new memory blocks were allocated. This is useful in iterative programs, where the total memory over various iterations should not increase. This assumes that the code does not reallocate new memory at every iteration. Isomalloc allocates each block inside different virtual memory pages. Each processor allocated memory from a unique portion of the total virtual address space available. This allows the block to be migrated to other processors while still maintaining it to the same virtual memory location.

5.2 The CharmDebug Memory Library

One of the memory libraries described in the previous section is built specifically for usage with CHARMDEBUG. This library, for every memory block requested by the user, allocates some additional space to store some metadata. The details of the extra space allocated are shown in Figure 5.2, and are described throughout this section. The layout refers to a 64-bit machine. The library returns to the user a pointer to the white region marked *user data* in the figure. A variation of the library allows the allocation of the metadata separately from the main user data, and for it to be stored inside a hash table.

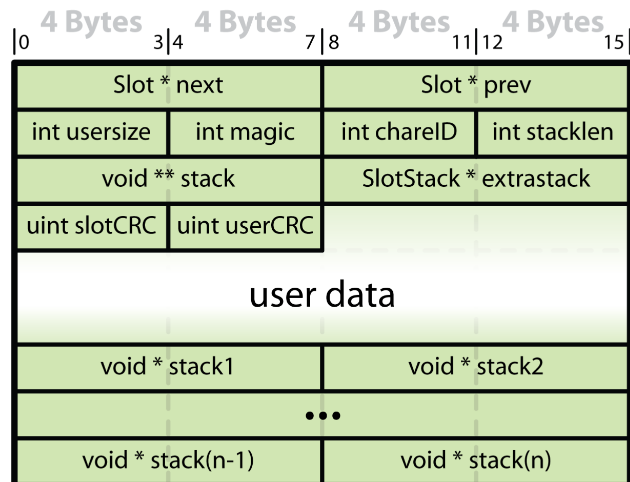


Figure 5.2: Layout of the extra space allocated by the CHARMDEBUG memory library on 64 bit machines. In shaded color the memory allocated for CHARMDEBUG purposes, in white the memory for the user.

We built upon the CHARMDEBUG existing framework to provide support for debugging memory related problems. The CHARMDEBUG memory library extends the existing CCS requests that the CHARMDEBUG plugin can serve by providing extra information regarding

the memory status.[2] A simple operation that CharmDebug can request is to view the memory of any given processor. Figure 5.3 shows how CharmDebug visualizes the information received from the application through CCS. The application in the figure performs a simple Jacobi computation on a two-dimensional matrix. Each allocation is colored in one of four different colors, according to its usage:

- memory that is occupied by a specific chare (in yellow);
- a message sent from one chare to another (in pink);
- memory allocated on the heap by the user code (in blue);
- memory allocated for the use of the Charm++ runtime system (in red).

Moreover, the CharmDebug memory library automatically collects stack trace information at every point where the user requested memory allocation. This information is stored at the end of the user buffer, as shown in Figure 5.2. The user can see this information at the bottom of the memory allocation view (Figure 5.3) by moving the mouse pointer over the allocated blocks.

Stack traces can also be combined by CharmDebug into *allocation trees*. An allocation tree is a tree rooted at the routine starting the program, typically main or a loader library routine. The children of a node are the functions that were called by the function represented by that node. The leaves are functions which called the malloc routine. This tree can become a forest if not all stack traces start from the same routine. This can happen, for example, in the presence of user-level threads with independent stacks. CharmDebug can construct an allocation tree for a single processor or for a subset of them. Allocation trees can be used for statistical analysis to provide insight of memory problems.

One of the operations that the CharmDebug memory library can perform is memory leak detection. Each processor parses stack and global variable locations for pointers to heap data. The heap memory blocks reachable by those pointers are further parsed for more pointers. This continues until all reachable locations are detected. Blocks not reachable are declared leaks. The result is reported in the same memory view described earlier (not shown here).

In the Charm++ environment, the user code always runs in the context of some chare. These chares, as we have seen in Section 5.1, are independent of each other, and should not interact except through messages. Therefore, another tag is automatically associated to each memory block, to identify which chare allocated it. This tag is shown in Figure 5.2 as *chareID*. Figure 5.4 shows the same Jacobi program with the highlighting of the memory associated with a particular chare. We shall see in the following sections how this tagging mechanism can be used to identify certain problems.

[2] The list of functions provided by this extension module can be found in Appendix A.

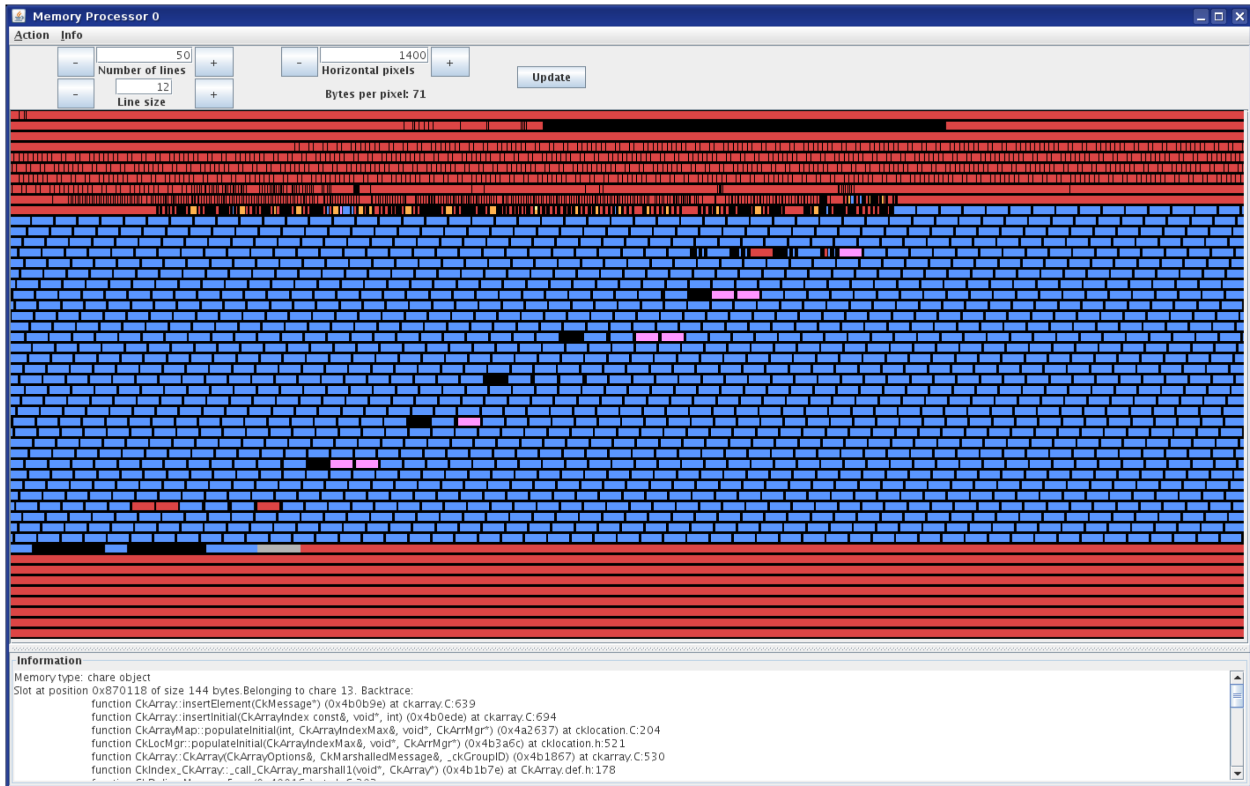


Figure 5.3: Memory view of the allocated memory for a simple hello world program. The color correspondence is: yellow-chare, pink-message, blue-user, red-system.

5.3 Detecting Cross-Object Memory Modifications

In a program like Jacobi, suppose chare A allocates a memory block for its local matrix, and then passes a pointer to the last row to chare B, instead of a newly allocated message with a copy of the last row inside. Chare B can access and modify the matrix of chare A during its computation if they are on the same processor. Nevertheless, Chare B is not supposed to modify chare A's state. More generally, different chares are not supposed to modify each other's memory since they are independent by definition. Based on this concept, we define each memory allocated by a chare to belong to that chare, and only that chare will be allowed to modify its content. The only exception is a message whose ownership will be passed from the creator to the chare that the message is delivered to.

The normal execution flow in a Charm++ program is depicted in Figure 5.5. The runtime system picks messages from a queue (on the left), and calls user-defined functions (on the right). This is indefinitely repeated throughout the whole execution. Figure 5.6 shows instead the modification necessary to intercept cross-object memory corruptions. These modifications lie on the interface between the system and the user code. In particular, before invoking the user-defined entry method, the runtime system resets the memory protection according to the object that is about to be invoked. Upon completion of the entry method,

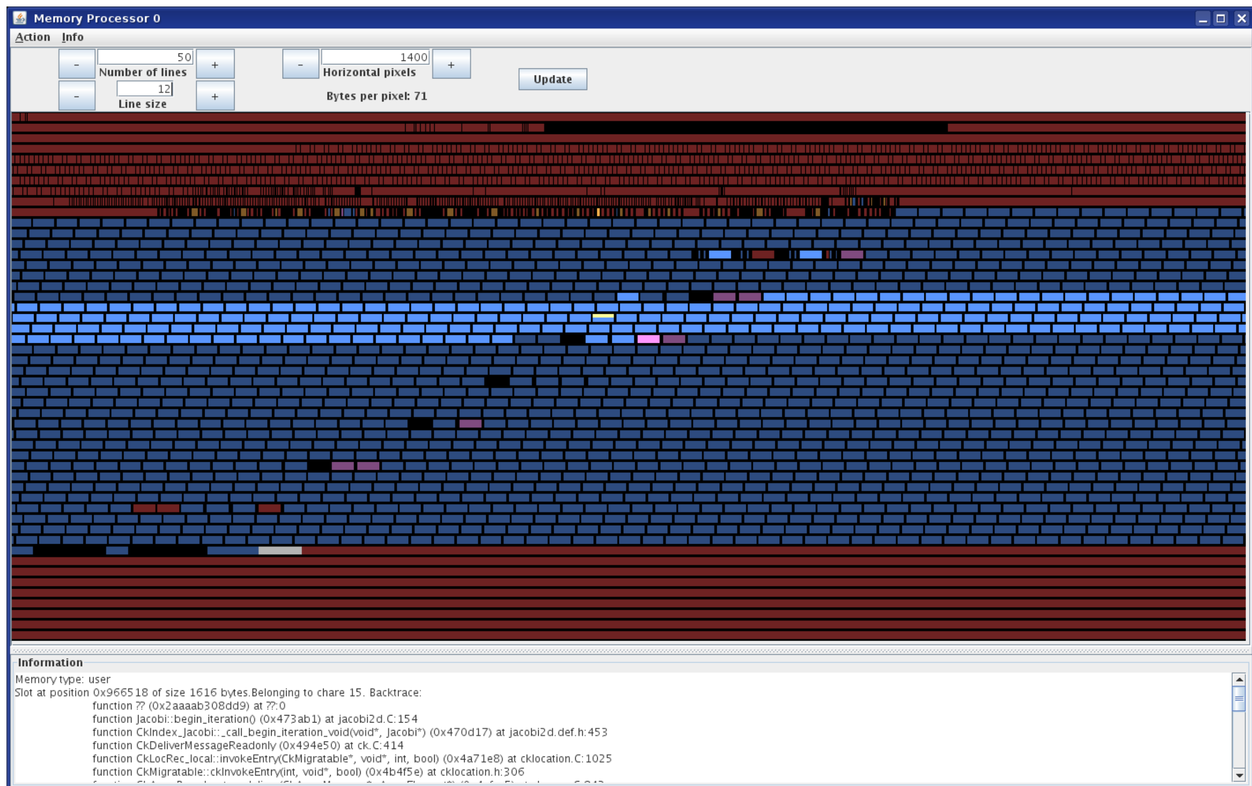


Figure 5.4: Dimmed memory view. In brighter colors are shown the regions of memory allocated by a specific chare, in darker colors all the others.

the system checks the status of the memory, and verifies if corruption has occurred. At this point, the system can suspend execution and notify the user, or issue a simple warning statement and continue.

Let us consider the example above, suppose we reset the protection for all allocated memory blocks before we deliver to chare B the message containing the pointer passed by A. Subsequently, we deliver the message to B and let it perform its computation. Let us assume B immediately uses the pointer to modify the matrix of A (if B uses the pointer in another entry method later, the same discussion applies for that entry method). After the entry method of B terminates, we check the memory protection. The protection for the block containing A's matrix will fail the check since B modified it. Since the block belongs to A, and not B, we can raise an exception and notify the user. The protection of some blocks belonging to B might also fail the check, but we ignore these since B was allowed to modify that memory.

The memory protection used can vary, and different needs may lead the user to choose one over another. We implemented three memory protection mechanisms: CRC, memory copy, and mprotect. These are illustrated in the following sections together with their strengths and weaknesses. Performance comparisons are presented in Section 5.5.

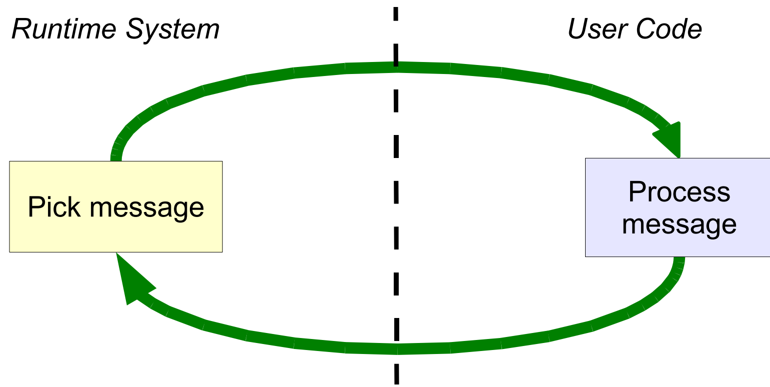


Figure 5.5: Normal execution flow of a CHARM++ application.

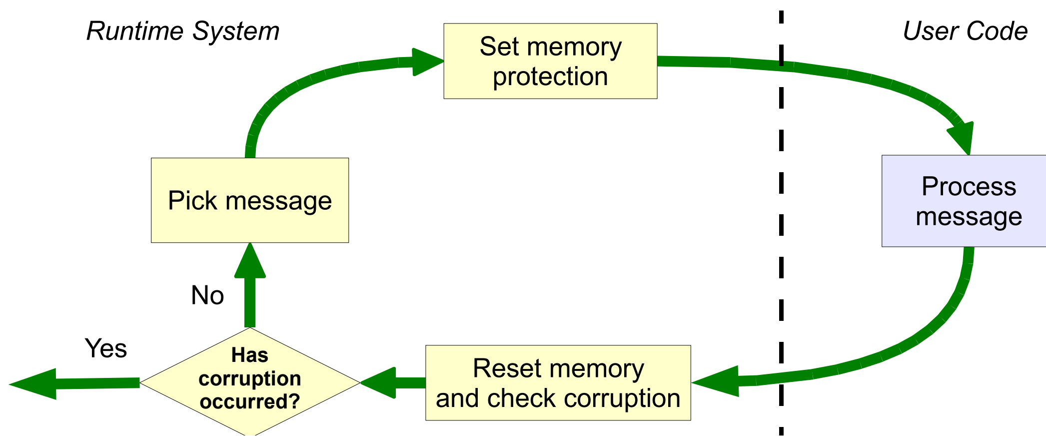


Figure 5.6: Modified execution flow of a CHARM++ application for cross-chare memory corruption detection.

5.3.1 Cyclic Redundancy Check

The first implementation of memory protection we implemented is based on Cyclic Redundancy Check (CRC) [61], in particular CRC-32. To reset the memory protection, a CRC is computed on every allocated memory block, and stored as part of the extra space allocated by the CHARMDEBUG memory library, as depicted in Figure 5.2 by the field *userCRC*. In addition, to protect the system metadata itself, a CRC is also computed for the metadata portion of the memory. This value is stored in the field named *slotCRC*.

Upon completion of the user entry method, all the CRCs are recomputed, and checked against those stored inside the metadata of each memory block. A discrepancy between the two values is an indication of a modification. The user is then provided with information about the error: which chare and which entry method are responsible for the modification, which memory block has been modified, and to whom that memory block belongs.

This method has the advantage that it changes the memory layout the least, and it requires a minimal amount of extra memory. All the information is contained in 16 bytes

within the metadata information. On the other hand, it has the drawback that it is very computational intensive. Computing the CRCs before and after every entry method can add a substantial overhead. A possible solution to overcome this limitation is explained with the performance evaluation.

One limitation is that if the faulty entry method internally spans a large amount of code, the culprit region of code can become very large. By allowing the user to request extra checks to be performed even in the middle of an entry method, the user can split the faulty code into subregions, and be notified about the region causing the exception. Another insufficiency of this method is when the program writes a value in memory, but the value happens to be identical to what is already in memory. In this case the method will fail to detect the change. In addition, with small probability, the method could also fail to detect a real change in the memory.

5.3.2 Memory Copy

The second solution we developed makes a copy of all the allocated memory. The memory is then left to be modified as the program desires. Upon return from the user code, the memory is compared with the saved copy, and corruption can be detected.

This method has similar characteristics to the previous one described in terms of corruptions that it can detect. It can detect any memory change, but not memory writes that do not change the memory. It also has the same limitations regarding the portion of code that is highlighted as responsible for a corruption. By using memory copy, a high pressure is posed on the memory system. In particular, the protection mechanism will double the memory allocated by the program. This can be a heavy burden on applications that already use a large portion of the system's available memory.

5.3.3 Mprotect

The third memory protection mechanism differs slightly from the other two with respect to its basic workflow. This new workflow is illustrated in Figure 5.7. Here, the protection is still set when switching from the system to the user code. In particular, all the memory not belonging to the chare processing the message is marked as read-only. When the control returns to the system, the protection is reset back to its original state, but no control is necessary on the memory. This comes from the fact that when a spurious write to a block not belonging to the executing chare, and thus marked as read-only, the signal SIGSEGV is sent to the application. Therefore, at the instruction where the corruption happens, the user can be immediately notified. This implies that this mechanism can resolve the corruption to the single line of code, thus being much more precise than the previous methods.

The main drawback of this method is that it requires every memory allocation to be performed via a call to `mmap`. This is necessary since for memory not allocated by `mmap` the behavior of `mprotect` is undefined [62]. Other than the problem of the existence of mmap on the system, by using mmap for every allocation, the memory layout will be substantially changed: every allocation will live on a different page. On machines where the page size is

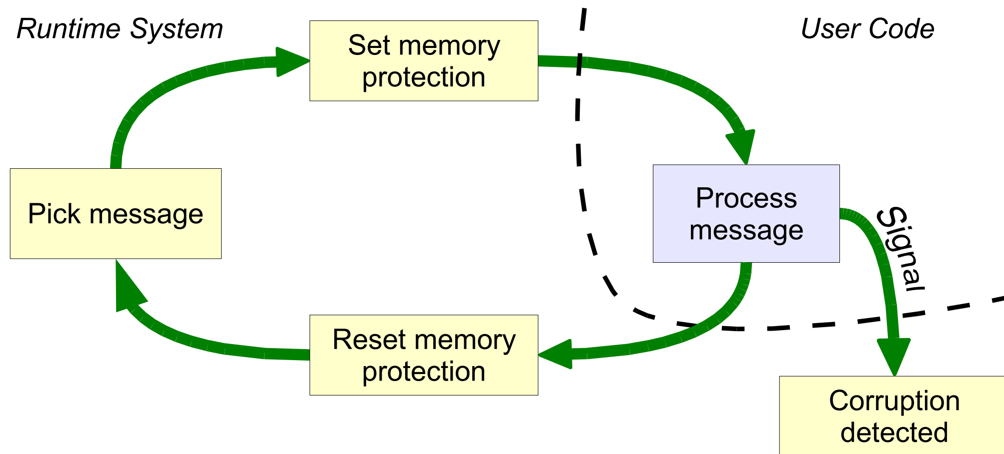


Figure 5.7: Execution flow of a Charm++ application when *mprotect* is used to detect cross-chare memory corruption.

large (on most supercomputers page size is nowadays 4 MB), this could lead to the exhaustion of the available memory very quickly.

A possible solution to overcome this limitation in the number of available memory pages is to allocate memory with mmap in large chunks, and then perform manual allocations from these regions. Each virtual entity will require an independent memory region from which to obtain its memory, otherwise individual protection of objects would not be possible. An exception to this approach are messages which by definition need to change ownership during the course of a program.

5.4 Detecting Buffer Overflow

Another common problem in applications is the corruption of adjacent blocks of memory due to overrun or underrun of array accesses. In the case of virtualized debugging, this memory could belong to a different virtual processor. A typical debugging technique is to allocate extra memory at the two ends of the user buffer, fill (or paint) it with a predefined pattern, and check if this has been overwritten by the program. One problem with this method is the granularity with which to perform the checks on the painted areas. Checking only when a block is deallocated is not enough, as it may never be deallocated or, even if deallocated, the region of code containing the error can contain very large portions of user code. Additional periodic checks on all the memory blocks may reveal the problem at an earlier stage, but it would still identify very poorly which lines of code are responsible for a corruption.

As for the case of cross-object modifications, we can use the entry method boundaries to perform the buffer overflow checks. Since CharmDebug already allocates extra space on both sides of the user allocated data, we can utilize this same portion of memory to check for buffer overflow corruptions. One necessary change in the detection scheme is that we now cannot paint that memory with a predefined pattern, since it contains valid information.

Nevertheless, the techniques previously described include the space before and after the user data in the protected memory. Therefore, similarly to cross-object modification detection, if a mismatch is found during the check of the protected memory, the fault can be attributed to the last entry method that executed. Of course, for this type of corruption, any mismatch found is a corruption. In particular, it does not matter if the memory modified belongs to the modifying object or not.

It should be noted that the mprotect mechanism is less capable of detecting this kind of bugs since it can only protect memory at the page boundary. For the other protection mechanisms, the same limitation on what kind of problem they can detect apply (write of an identical value to the one already in memory). As before, the user may specify a coarser granularity of checks, in which case there will be a set of entry methods that will be checked, or a finer granularity by adding extra checks inside his code.

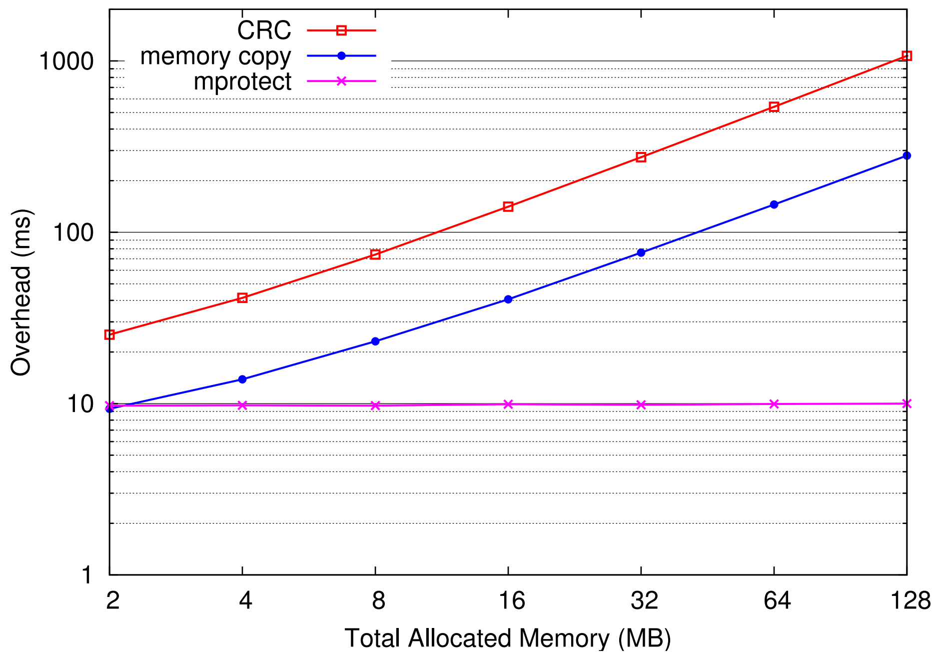
5.5 Performance Aspects

We analyzed the overhead imposed by our implementations, and compared the performance of the three protection mechanisms.[3] The test application was a simple ping-pong program in which two objects were exchanging the ping message. This program was executing on a single processor, since the overhead imposed does not depend on the total number of processors allocated, but only on the local characteristics of a processor's memory. The objects did not perform any computation upon receipt of a ping message, and no allocation was performed inside the entry methods. We performed 1,000 iterations of the ping-pong exchange, and timed the total execution. The benchmark was performed using a 2.0 GHz Intel Xeon E5405.

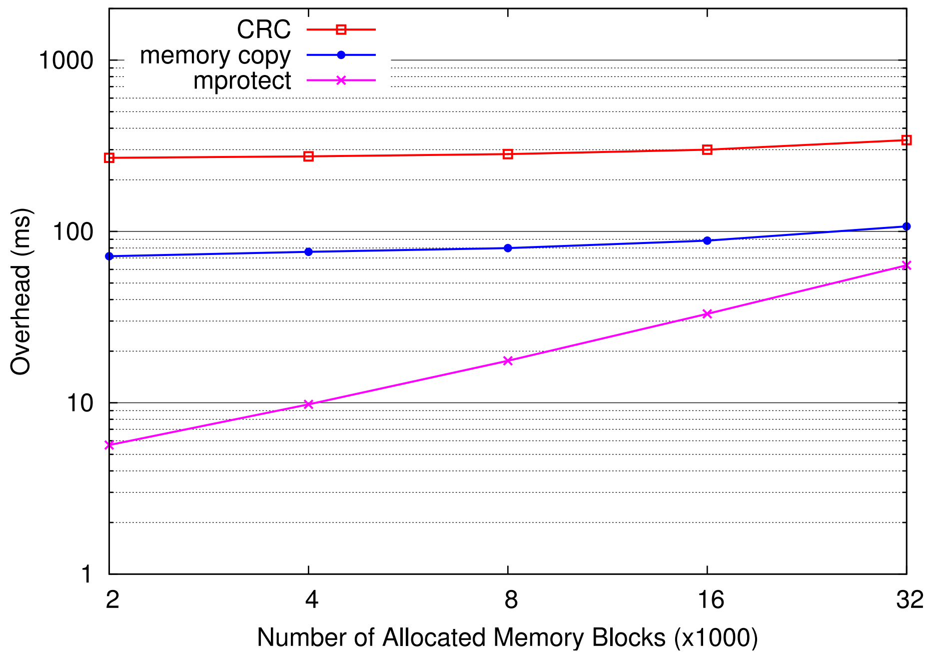
When executing the ping-pong program without any memory protection mechanism, the overhead incurred was around 2-3 μs. This time is to be attributed to the normal scheduling overhead of CHARM++. Since the purpose when using memory protection mechanisms is to debug an application, we did not use compiler optimizations. The memory libraries, instead, have been compiled with full optimization since they are not target of debugging.

Figures 5.8(a) and 5.8(b) show the overhead incurred by each method when varying the total amount of memory allocated, or the number of memory blocks allocated, respectively. In the first plot the memory was divided in 4,000 allocations, while in the second plot the total amount of memory allocated was 32 MB. Each number in the plots includes two scans of the entire memory, one to set the protection before the entry method is called, the other to reset it after the entry method finishes. It can be seen that mprotect is always the fastest, while CRC is always the slowest. Naturally other parameters may play a role in choosing one over the other, as each has strengths and weaknesses. Moreover, the first two methods have a strong dependency on the total allocated memory (1st plot), while mprotect has a strong dependency on the number of allocated blocks (2nd plot). This is not a surprise given the characteristics of each method.

[3] Data in this section was collected with help from Ramprasad Venkataraman.



(a) Varying amount of total memory. Memory distributed in 4,000 allocated memory blocks.



(b) Varying number of allocated memory blocks. 32 MB of total allocated memory.

Figure 5.8: Overhead to set and reset the memory protection for all the allocated memory using the different protection mechanisms.

To compute the slowdown an application suffers when using the different methods, one has to include the average granularity of the entry methods. Let g be the average time taken by an entry method, and $o(b, m)$ be the overhead of a protection mechanism in function of the number of allocated blocks, and allocated memory. Then the slowdown S of the application is:

$$S = \frac{o(b, m)}{g}$$

In particular, for coarse grain applications the slowdown is acceptable. For fine grain applications, the slowdown can be very high. Note that this slowdown applies when the application is left free to make progress. If the user is analyzing a portion of the code, and is delivering messages manually, then this overhead will generally not be perceived by the user when delivering a single message.

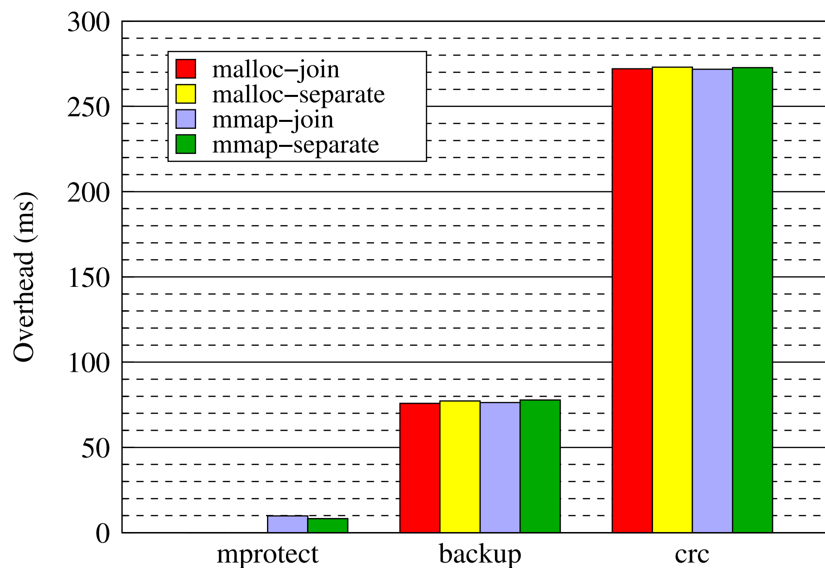


Figure 5.9: Overhead to set and reset the memory protection for 32 MB of memory allocated in 4,000 blocks using the different protection mechanisms. Comparison between the different CharmDebug memory libraries.

The experiments described have been conducted with the memory library compiled with support for mprotect, meaning each user allocation is internally transformed into an mmap operation, and the metadata positioned at the beginning of the user data. There are other three versions in which the CharmDebug memory library can be compiled. The four versions depend on the internal conversion of user allocations to `malloc` or `mmap`, and the positioning of the metadata—joined with the user data or separate into a hash table. The comparison between the different versions of the CharmDebug library are presented in Figure 5.9 for a single data point. Except for small variations, the performance is similar independently of the specific CharmDebug memory library used.

From Figure 5.9, two bars are missing in correspondence with mprotect used with `malloc`

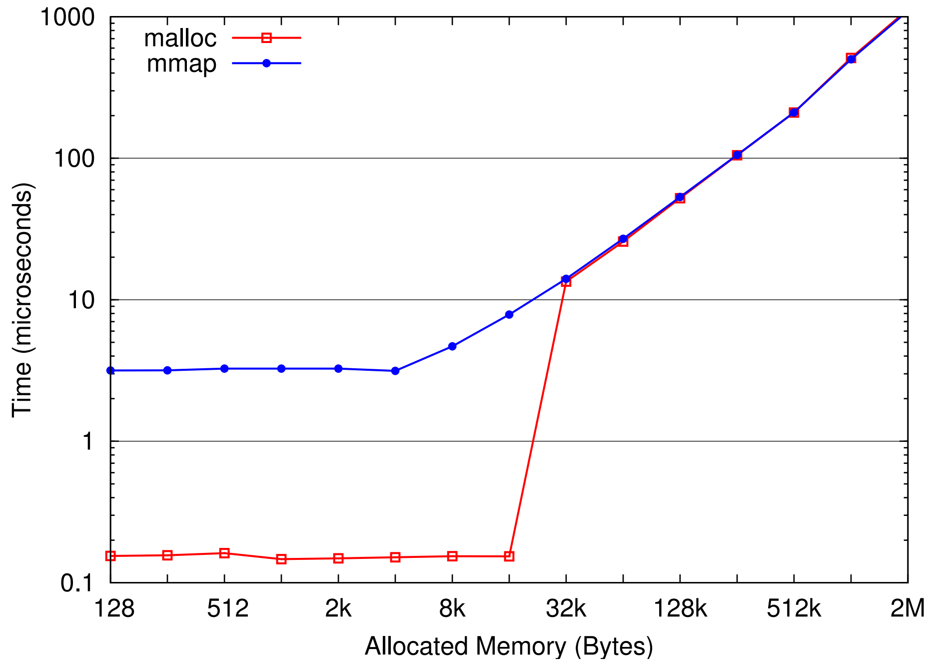


Figure 5.10: Cost to allocate and deallocate a memory block in CHARM++ when using the two CHARMDEBUG memory libraries based on `malloc` and `mmap`. Averaged over 10,000 allocations.

allocation. While the change between `malloc` default allocation and `mmap` does not affect the performance of the protection mechanisms, it affects the performance of the application in other ways. In particular, allocating a memory region with `mmap` can be more expensive than allocating it with a correspondent `malloc`. Figure 5.10 shows the different cost of allocating a block of memory in the two cases. When the allocated memory is large enough there is not much difference, but for small allocations the impact is significant. This is a potential drawback for mprotect in case of applications that perform many allocations of small data.

5.6 Related Work

There are various tools that help debugging shared accesses to a variable in a multithreaded environment. Intel Thread Checker [63] is one such tool. It can detect both read and write unsynchronized accesses to shared variables. It uses dynamic instrumentation to inspect each memory access performed by each thread, and returns statistics on threads using the same locations. Given that it needs to intercept and perform extra operations at every memory access, it significantly slows down the execution of the program. An improvement on this tool has been proposed [64] by filtering most memory accesses that are not likely to produce data races, and checking only those not filtered out.

Another tool is RecPlay [65], which combines record-replay and on-the-fly data race detection to efficiently inspect concurrent programs implemented using POSIX synchronization

operations, and detect data races. The algorithm requires the program to be run several times to obtain all the information to identify both the racing data and the racing instructions. Even though the code is executed multiple times, since most of the time the program runs without slowdown, the total overhead is reduced. This algorithm has the disadvantage that it can only detect the first race condition, and looses effectiveness if the user decides that he does not want to, or cannot, remove the data race.

All these tools are for shared memory accesses. In the scenario described in this thesis, where there is one single thread of execution, and the program is decomposed into independent objects, such tools would not be useful. Other tools for sequential programs, such as Valgrind [66], are capable of detecting buffer overflow and other memory related problems. These tools typically incur in acceptable overhead. Again, for the scenario described here, these tools provide little support to the user.

TotalView [67] is another powerful debugging tool capable of inspecting and analyzing the memory allocated by an application, and it supports parallel distributed systems such as MPI. TotalView allows the user to collect memory views and save them for future reference. These saved views can be compared against each other or against the status of the live application. By saving and comparing memory states, the user can simulate our comparison tool. Nevertheless, it is not possible to automate the collection of states and their comparison, forcing the user to undertake a tedious stepping through the code.

5.7 Future Work

The techniques presented in this thesis cover different possible ways to protect the memory of an object from corruption by another object present in the same application. Each of them has some strength and some weakness. One problem with all the proposed techniques is their performance. When the grain size of the application's entry methods becomes smaller, all the techniques introduce a high overhead. Several optimizations can be added to reduce their overhead. One was already presented during their discussion, the possibility to apply the protection only to some of the entry methods. Clearly this works only if the user trusts certain entry methods, but does not apply to the virtualized debugging scenario where each processor is a potential candidate for corruption.

As the techniques were described, an entire scan of all the memory is performed both before and after each entry method to set and reset the memory protection, respectively. Instead, the system can be modified to perform only one scan per entry method, and combine the protection reset of one entry method with the protection set of the next entry method. In some situation, if multiple messages are destined for the same entry method on the same object, both operations could be entirely skipped, and the two entry methods could share one protection. Clearly, now the user would have to distinguish which entry method caused the fault. In the case of virtualized debugging, if the only use of the protection mechanism is to prevent corruption between processors, only one protection could be applied for the entire scheduling of one virtual processor, and reset the protection only upon switch of virtual processor.

Regarding the mprotect protection mechanism, it is also possible to avoid the scan of the entire memory when setting the memory protection. This can be achieved by indexing the memory belonging to a specific object, and scan only through those memory blocks when changing the protection. Since the performance of mprotect is linearly dependant on the number of blocks processed, this would be a considerable performance improvement. Unfortunately, the other two protection mechanisms cannot benefit from this optimization since they still need to scan the entire memory to detect if any random location has been modified.

6 Processor Extraction

Parallel applications tend to behave non-deterministically, especially when they contain bugs. This means that even with the same input, the same application may produce different results over multiple runs. This can significantly complicate debugging, even in small scenarios. One common type of errors caused by non-determinism is race conditions. These are bugs where the outcome of the computation is unpredictable because it critically depends on the sequence and timing of the communication between processors. [1]

One possibility to solve this problem is to capture the non-determinism that the application manifests, and make it repeatable. This is generally performed with a technique called "Record-Replay" [69, 70]. This technique has a few requirements. First of all, the information recorded must be sufficient to allow the proper replay of the application deterministically. Secondly, the recording procedure must not perturb the application too much. If the recording has too much overhead, it might make the bug disappear, and render the technique useless. For example, a similar effect happened while debugging ChaNGa [18], a cosmological simulator developed as a joint collaboration between the University of Washington and the University of Illinois. In this application, we discovered that certain messages were racing, and caused the application to crash when a particular ordering was executed. Hence, when we added print statements for debugging purposes, the problem usually disappeared.

When the application is deployed in a production environment, or is benchmarked on much larger configurations, other latent race conditions may appear. Imagine the delays a network can introduce when routing packets through a torus or fat-tree interconnection, especially in the presence of congestion. On a small cluster, messages may never get out-of-order and expose a race condition. Furthermore, other bugs may appear only at large scale. For instance, the algorithm that distributes the application's input among the available processors may generate incorrect results when performing fine grain decompositions. These kinds of problems are very common in the early stages of production-level applications. They are also much more challenging to track, as they may manifest only when thousands of processors are involved in the computation, and disappear when fewer are used.

The programmer may try using smaller input datasets in order to reproduce the problem on a smaller numbers of processors. Unfortunately, this is not always possible. As many scientific applications have physical phenomena driving the advance of the computation, using smaller input datasets may hinder the approximation of the real world. This may render

[1] The work in this chapter is reprinted, with permission, from "Robust Non-Intrusive Record-Replay with Processor Extraction", in the Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII) ©ACM, 2010. http://doi.acm.org/10.1145/1866210.1866211 [68]

the application not usable even for debugging. On the other hand, using large datasets on a small number of processors may not be possible due to the amount of memory required, or because the bug simply disappears. For example, this happened while debugging Rocstar [59], a rocket simulation program developed by the Center for Simulation of Advanced Rockets (CSAR) at the University of Illinois. In this case, the problem only appeared when using more than 480 processors, and on a fairly large input dataset.

In both the examples given, the programmer could not reduce the number of processors and still see the bug. While record-replay helps in making the manifestation deterministic and easy to follow, there is still the problem that the execution must be followed on possibly thousands of processing units. Luckily, in most situations, the bug appears on a specific processor in a clear way; for example with a segmentation fault or an assertion failure. Moreover, if the non-determinacy is captured correctly by record-replay, the bug's effects/causes are confined to a few processors. At this point, the programmer does not care anymore about the rest of the system, and desires only to focus on a few processors.

What we propose is a new technique that encompasses both the advantages of a full record scheme, which allows a single processor to be "extracted" from the application and executed as a stand-alone, and the advantages of a minimal record scheme, which incurs little overhead in the application without hiding the bug. Our technique combines these two record-replay schemes into a simple, yet powerful, three-step procedure that a user may follow to debug his application. Furthermore, since debugging may be a long process and access to large machines may be limited, we paid special attention to reducing the need for large parallel machines to the minimum. We do this by using a virtualized environment supported by a parallel emulator, which emulates the large machine using only a limited amount of resources.

6.1 Related Work

In the field of debugging, record-replay techniques have been studied extensively. Several articles [71, 72] provide broad overviews of how this technique has been applied to parallel and distributed debugging. Most of the literature focuses on applications written for shared memory systems, where races are represented by threads writing the same locations in memory. Fewer articles discuss issues in a distributed memory environment, where message passing is the cause of non-determinism.

Of the implementations of record-replay that treated distributed processing, [73] and [69] were among the first. In particular, they record the full content of all the messages exchanged in the system. Currently, a modern tool integrated into the TotalView debugger is ReplayEngine [74]. While these tools allow the full recording of the execution and its later deterministic replay, they all incur a high overhead during the recording phase, which might cause the problem to disappear.

The amount of data recorded during the execution of the program has always been of concern. In [75], the minimum amount of information necessary to replay the execution is computed at runtime, and only this information is stored to disk. More recently, [76]

has proposed to reduce the amount of data stored by grouping processors, and storing full content only for messages between processors in different groups. For processors within the same group, only the message ordering is stored. Both these approaches achieve a significant reduction in disk space usage when compared to full record techniques, but they still have a considerable overhead in the recording phase.

Our approach differs from previous ones by imposing a negligible overhead during the most time critical phase of the application, when the non-determinism is captured. In this phase, any overhead is a potential for intrusion (i.e the Heisenbug principle), and can make the bug disappear. We also succeed in minimizing the data stored by having a second recording phase, where only processors of interest are recorded in full detail. Even for very large executions with thousands, or potentially millions, of processors involved, our scheme will record the full content of the messages only for a few processors selected by the user.

The replay time has also been considered and analyzed in literature. In [77], checkpoint has been combined with record-replay to allow the replayed program to reach the failure point more quickly. In [78], the replay time has been further analyzed to provide an upper bound: the system automatically makes a tradeoff between checkpoint and recorded data to meet the user-specified replay time. Furthermore, these checkpoints have also been used to allow backward movements in time, such as in [79] and [80].

6.2 The Three-step Procedure

The three-step record-replay procedure we propose is based on the following two algorithms. The first algorithm is a non-intrusive record-replay technique that records (in memory) only the minimum amount of information necessary to eliminate the non-determinism from the application. In particular, this recording consists of the ordering in which messages are processed by each processor. Since the amount of information is minimal, special care needs to be taken to detect situations where the information recorded becomes insufficient for the correct replay of the system. A technique based on the computation of checksum of the received messages is presented in Section 6.4.

The second algorithm is a more intrusive one, and records the full content of each message processed by a selected set of processors. The generated output can be big, but contains enough information to replay the recorded processor by itself as a stand-alone. Note that given the possibly high volume of data recorded, this recording is performed only on a subset of the processors specified by the user. Since this algorithm is more invasive, if used alone, it has the potential to disrupt the timing of message receipt between processors. By combining it with the first algorithm that records only the message ordering, we can obtain reliable results.

The three-step procedure that is based on these two algorithms is depicted in Figure 6.1. In the first step, the entire application is executed on the large target machine, and basic information about message ordering is recorded. Optionally, the user may decide to enable the self-correction feature provided by our technique. Note that this step is the only one that actually requires the use of a machine with as many processors as those required by the

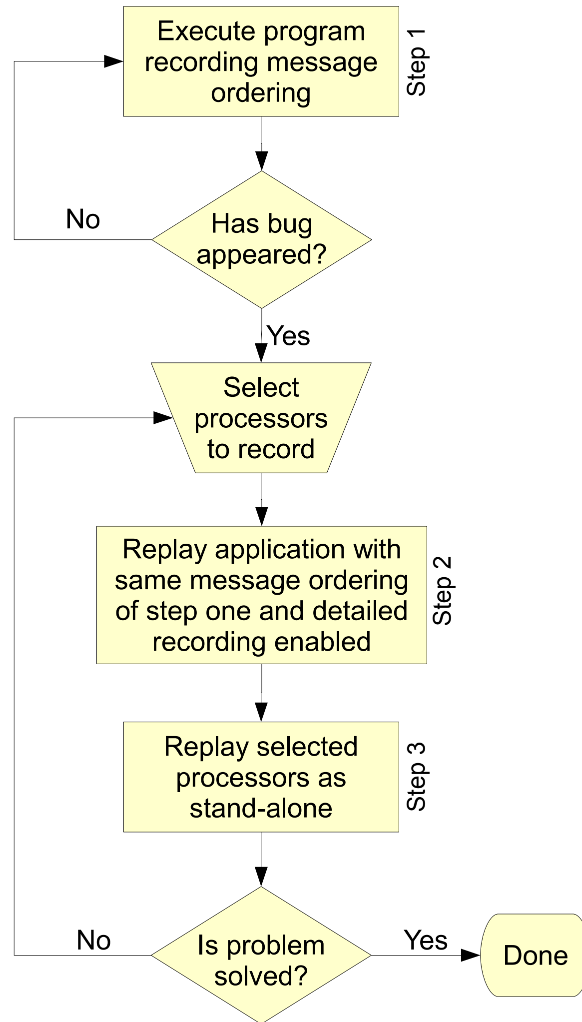


Figure 6.1: Flowchart of the three-step algorithm.

application.

For the second step, the programmer first identifies a set of processors to focus on. Good candidates are processors that crash, or processors generating incorrect output. The entire application is then replayed using the message ordering collected in step one, and the selected processors are recorded in detail. By using the same message ordering as in step one, we can guarantee the determinism of the execution, and the more intrusive recording necessary for processor extraction will not affect the bug appearance. In step two, the user may use the same large machine used in step one, but it does not have to, as we shall see in Section 6.6.

In the third step, the detailed traces recorded in step two are used to replay a selected processor using a single processor. This re-execution can happen either on the same machine where the traces were recorded, or on a local machine. The possibility to move to a local machine depends mainly on the compatibility between the architectures of the parallel and

local machines. On the replayed program, the programmer can use traditional sequential debuggers like GDB [23], and follow the problem in detail, re-executing it as many times as needed.

During the third step, the programmer may realize that some processor that has not been recorded is now needed. For example, he may realize that an extracted processor receives a corrupted message from a processor that was not recorded. By repeating step two, these missing processors may be extracted too. This establishes an iterative procedure that allows the programmer to identify an initial set of processors of interest, and expand this set later if necessary. Note that every time the second step is performed, the traces recorded during step one are used. Therefore, the same ordering of messages is guaranteed, and the processors extracted in different passes are compatible with the same manifestation of the bug under inspection.

As mentioned, of the three steps, only the first one actually requires a large machine to be used. Step three clearly requires only one processor to be allocated for the replay of an individual processor. As for step two, it can be performed using fewer physical processors than those needed by the user, by executing the application within a virtualized environment. This virtualized environment can still use the traces from step one to guarantee the deterministic replay of the application. Naturally, the time required by step two may increase as the application will have fewer computational resources available. The detailed traces generated in this virtualized environment can then be used in step three as before. We will discuss this in more detail in Section 6.6.

Another important consideration regards optimizations performed by compilers. Generally, during debugging, the user's application must be compiled with debugging symbols, and without optimization. Debugging an optimized code can lead the debugger to not correctly correlate the generated assembly code with the original source code. On the other hand, optimizations can radically change the performance of a program, and in particular its timing. Oftentimes, an application that crashes when compiled with optimization enabled may succeed when no optimization is used. This creates a big problem for standard record-replay techniques during the recording phase, when timing is essential for the bug to appear. One solution would be to support the debugging of fully optimized code. While this has been studied in literature [81, 82, 83], the commonly used compilers and debuggers still are not capable of correctly handling optimized code.

In our approach, since the information recorded in step one is independent of the particular compilation and depends only on the algorithm used, the user is allowed to switch between an optimized and a non-optimized code. In particular, he can use the optimized version in step one where timing is critical, and a non-optimized version in steps two and three. Since our message order recording scheme has a minimal impact on the application performance, as we shall see in Section 6.7, using an optimized version greatly reduces the possibility of the bug disappearing.

When the application has to run for many hours before the bug appears, maintaining all the logs in memory during step one becomes impossible. To solve this problem, we have two solutions. One involves the application manually flushing the logs to disk at appropriate

times, when the disk I/O does not disrupt the timing. As many scientific applications are iterative, and contain explicit barriers between iterations, adding one phase to flush the logs synchronously does not add significant overhead. Alternatively, the application can make use of the checkpoint/restart scheme available in Charm++ [84] to automatically checkpoint and restart from a point in time closer to the problem. In this way, the total amount of log data each processor has to hold in memory is kept small.

6.3 Ordering Messages

Although the idea is general enough to apply to any other message-passing system, we implemented the proposed record-replay techniques in the context of the Charm++ runtime system. In addition, our idea is trivially applicable to MPI applications by using the AMPI [15] implementation of the MPI standard.

In Charm++, the order in which entry methods are invoked, and threads are scheduled, is determined by a processor-level scheduler, and by the priority of each message (which can be set by the sender).

The processor-level scheduler, also called Converse scheduler, implements an infinite loop that examines different message queues in the system, and determines the order of execution. These queues are: (a) a network queue, which contains messages coming from other processors via network; (b) a node level queue that contains messages from other processors on the same SMP node; and (c) a local queue, which contains messages that objects on a processor send to other objects on the same processor. The messages from these three queues are combined together, and then messages are scheduled according to their priority.

Messages may arrive from the network in any order, and they are placed in the network queue in the order they arrive. Messages sent from the local processor will be picked up by the Converse scheduler sooner or later depending on the presence of messages in the network queue. As a result, race conditions between messages may occur, and this can lead to hard-to-find application bugs. Therefore, in order to capture the parallel behavior of an application for debugging purposes, it is important to record the message ordering.

A simple deterministic record and replay scheme in Charm++ has been available for several years [13]. This scheme is based on the assumption of piecewise deterministic execution [85]:

> Let *obj* be an object in the system with associated state *s1*, and *msg* a message sent to *obj*. Suppose the processing of *msg* by *obj* causes the state of *obj* to transition from *s1* to *s2*, and a set of messages *M* to be sent. Then, if we deliver the same message *msg* to the object *obj* in state *s1*, the object will always transition to state *s2*, and will always send the set of message *M*.

Under this assumption, the only source of non-determinism in the application is the order in which messages are processed. Therefore, by recording a tuple containing the

sending processor and a per-processor unique sequence number, the system can be replayed deterministically by re-ordering messages according to the recorded sequence. In addition to this tuple, the original scheme also saved the size of the message as a simple check to make sure the messages processed are indeed the same between executions.

6.4 Robustness and Accuracy

The scheme for recording the message ordering in CHARM++ applications can be used in the first step of our proposed three-step procedure. However, it has several problems regarding robustness and accuracy. One limitation is the assumption made about piecewise deterministic behavior. Although in general this condition should hold, some applications may not entail such determinism. For example, the application may use timers. Imagine the scenario where an entry method receives a message and, depending on the elapsed time since last invocation, performs different operations, possibly sending different messages. In this scenario, a different timing in the network, maybe due to a sudden congestion, can modify the behavior of the application, even if the same ordering of messages is maintained.

Capturing system-level calls, like timers, may solve the problem for the given example. By following this path, many other system calls, along with their return values, need to be included in the recorded traces. Particularly voluminous may be those reading from files. The complexity and amount of data to be stored would rapidly increase in this scenario. This would lead to an increase of the overhead incurred by the recording scheme, making it more likely to disrupt the precarious timing which leads to the manifestation of the bug. Therefore, to maintain the overhead to the minimum, we do not record these additional information, and limit the applicability of our technique to piecewise deterministic applications. Fortunately, the vast majority of applications do oblige to the piecewise deterministic assumption, and for these applications, the simple message ordering is sufficient.

The key now is to understand when the piecewise deterministic assumption is satisfied by the application, and to detect when it is not satisfied. The original scheme tried to do this by using only the message size. Unfortunately, most applications tend to have many messages with the same size, yet completely different content. In order to guarantee the piecewise determinism, in addition to the message ordering, we would like to include also the content of the messages. By having available the full content of the messages, it would be trivial to determine if all the messages processed during the replayed execution are identical to those processed during the recorded execution.

Before continuing, a word of caution is in order. Even by extending the recording to assure that communicated messages have the same content, we cannot prevent a processor from having internal non-determinism completely. For example, the processor may still make a decision based on the current time, and modify its local state differently in different executions. However, since by definition all outgoing messages are the same between two executions, and the only way to influence another processor is through explicit message passing, the local non-determinism cannot propagate to other processors. Therefore, while searching for a bug, the causal relationship between processors does not change.

Clearly, having the receiving processor store (in step one) the whole content of each message received would defeat the purpose of our 3-step procedure which aims at a non-intrusive mechanism during the first step. Instead, we compute the checksum of the received messages, and store only this information into the recorded traces. The amount of data added by the checksum is only a few bytes per message, therefore adding little overhead, as we shall see in Section 6.7. Of course, our technique can only capture a difference in the message content with high probability. If the content of a message in two different execution is such that the computed checksum is identical, then our method will fail to detect the change. Nevertheless, since this check is performed for every message processed, the probability that the non-determinism will remain latent as the application progresses is extremely low.

We implemented two commonly used checksums. Both of them produce a 32-bit integer value. The first is a simple XOR of the message data, reading four bytes at a time. This checksum is fast to compute, but has the disadvantage that it is easy for a message to contain differences not detected. The other is a more sophisticated Cyclic Redundancy Check (CRC32) checksum. This is a more computationally intensive algorithm, but can capture difference in the transmitted data with higher confidence. The programmer, during step one, can choose which of the algorithms to use. He can also choose to altogether skip this checksum computation to minimize the overhead.

In order for the checksum computation to yield correct results, it is important that the message content of each message be identical between the two executions. One problem is posed by the presence of garbage inside a message. Consider a data structure like that in Figure 6.2. In this case, the compiler has padded the data to maintain correct alignment of the data structures, in this case of the double type. When allocating this data structure, the padded memory region, which is shown in the light color, is not initialized, and may contain any random garbage. To overcome this, we developed a solution that makes use of the memory allocation sub-system present in CHARM++, and make sure every newly allocated memory is always initialized to a known pattern.

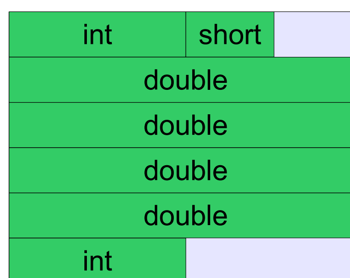


Figure 6.2: Example of data structure padded by the compiler.

In the memory sub-system implemented in the CHARM++ runtime system, there is an interface to easily re-implement memory related functions, such as malloc and free, and place them into a dynamically loadable library. There are three types of re-implementation. One uses the `glibc` memory arena internally, and wraps it with the new function definitions. The

other two are based on a direct usage of the memory allocator provided by the operating system. This is done either by dynamically loading the specific function pointers using `dlopen`, or by using the hooks present in the operating system in the case of the GNU implementation. For our purposes, we extended an implementation created specifically for debugging. This implementation can use all the methods described above to link to the underlying memory system, making it very portable. In this extension, we could easily add a call to `memset` before returning the allocated memory to the user. Note that the memory can be pre-filled with any known pattern.

Another problem that we encountered while using the original record-replay scheme in Charm++ was the lack of ordering of threaded entry methods. As mentioned in Section 5.1, Charm++ programs may declare an entry method as "threaded", thus creating a user-level thread for the execution of each invocation of the entry method. Since thread operations, such as suspend and resume, are treated at the lower level of the runtime, the record-replay module was not aware of them. This produced a lack of recorded information for which threaded entry methods could be executed, and resumed, without a specific ordering with respect to the other entry methods. This clearly was a problem. To solve it, we placed a hook in Converse's threaded library and exposed the occurrence of thread events to the record-replay module. This allowed these events to be properly logged.

Finally, we also considered the interaction of the record-replay scheme with load balancers. When a load balancer is present in Charm++, it will instrument the execution time of the application's entry methods, and migrate the objects accordingly to balance the load. Internally, timers are extensively used to record the execution times. This creates a problem given the lack of recording for this event in our scheme. However, what is important in this case is the decision taken, not the particular input data that was used to achieve such decision. Therefore, we decided to store to disk only the decision taken by the load balancer, ignoring all the collected timers. This also helps maintaining the log sizes small, since the time measurements can be very numerous. These decision messages, that are recorded during step one in a separate file from the message ordering log, and then loaded during replay in phase two. It is useful to note that, during the load balancing operation, the application is usually paused and awaiting for the decision to be executed before resuming operation. Therefore, it is generally not a problem to alter the timing during a load balancing phase.

With the enhancements described above, the new record-replay scheme has become a much more robust and accurate solution for record and replay. We understand that other problems may arise in the future, both from the introduction of new features in Charm++ and from old features not properly treated. Should new issues arise, we plan to expand the recording scheme accordingly, as we have done for threaded entry methods.

6.5 Processor Extraction

In step two, once the information we want to record is identified—all the messages received by a processor—the extraction is relatively simple. All we have to do is to record the content of the messages processed by the user-selected processors into a file, so that in step three a

modified Charm++ runtime system can replay the execution of any selected processor as a stand-alone. When replaying a processor, the Charm++ runtime system loads messages from the corresponding trace file instead of receiving them from the network, and since the processor is replayed without a network, all outgoing messages are discarded. More important is considering the implications involved with re-executing the application.

As we mentioned at the end of Section 6.2, we can always change the executable used between steps one and two, provided the application performs the same operations—for example by changing the optimization level. On the other hand, the information recorded in step two contains information that may be specific to a particular executable, and may change if the application is recompiled. For example, in ChaNGa, several function pointers are sent through messages between processors. During normal execution, this is acceptable since the executable is the same for all processors. Nevertheless, recompiling the code means changing the placement of these function pointers, and thus invalidating the data contained in the messages. For ChaNGa, therefore, the same executable must be used in steps two and three. For other applications, like Jacobi, where the content of the sent messages will not change even if the application is recompiled, the optimized version of the code may still be used in step two.

Another problem appears if the two executable are compiled for different versions of Charm++. Changing architecture would be desirable if the architectures of the parallel machine and of the local machine differ (say one used LAPI, the other pure ethernet). Unfortunately, this is not possible at the moment. In Charm++, a different architecture signifies a different header added to each message. These headers are not only different in content, but also in size. Furthermore, the content of the message stored will follow the endian-ness convention of the machine where the data was recorded, and translating messages to a different architecture is not simple.

6.6 Further Reducing the Need for Large Machines

In our proposed three-step procedure for recording and replaying a buggy application deterministically, the second step may be performed inside a virtualized environment. As we mentioned in Section 6.2, this is very useful to reduce the number of physical processors needed, and to reduce the contention on the availability of a large machine.

This is particularly important when the bug appears only on large processor counts. In this case, executing multiple times the buggy application on a large parallel machine to extract different sets of processors may introduce long delays in the debugging process. This can easily happen when submitting jobs to a batch schedulers on heavily used machines. By using a virtualized environment, instead, we can perform the processors extraction operation using a much smaller machine, and increase productivity. To demonstrate the feasibility of this approach, we used the BigSim Emulator [24].

6.6.1 Detailed Record-Replay in BigSim Emulator

For performance prediction reasons, the BigSim Emulator supports a detailed record and replay scheme that stores the full content of messages to disk. This is similar to the one used in the processor extraction described earlier, but under the virtualized environment. When emulating an application, the user may specify a subset of processors that he wants to record in detail. During the emulation, on each of these emulated (or target) processors, the scheduler stores a copy of the message to its own trace file before it executes the entry function associated to that message.

We incorporated the BigSim Emulator's record-replay capability into the proposed three-step procedure as an alternative to reduce the need for large machines in the second step. This new three-step scheme thus becomes: (1) execute an application on a big machine, and record the message ordering; (2) replay the application on a machine emulated under the virtualized environment and record the detailed traces; (3) replay the execution of a selected target processor sequentially. Note that if step two was performed within the emulated environment, step three must also be performed in the same environment. This comes from the fact that BigSim Emulator is considered by the application as another communication layer, and at the moment we cannot change this layer between step two and three. Nevertheless, we are considering extending the possibility to perform step three in a different scenario (say outside BigSim, or ethernet vs. LAPI).

In this new scheme, the emulator needs to be able to read the trace logs generated in the first step from the non-emulated execution on a full machine, and replay the application in the emulator using a small machine. One challenge was to match the two executions of the application on these two totally different environments. Specifically, when the trace log tells that the next expected message is (srcpe, msgID), where *srcpe* is the source processor ID, and *msgID* is the message ID of that message, the message IDs must be identical on the two environments. This is easy to guarantee as long as the emulator emulates the Charm++ runtime faithfully and both systems assign msgID to each message using a sequence number local to its sender processor. However, this becomes rather complicated when user-level threads are involved in an application. This is because emulated processors themselves are implemented as the same user-level threads. Therefore, tracing the suspend/resume events of user-level threads will mistakenly record the events of the emulator threads, creating mismatch of the thread event IDs between two executions. One way to handle this is to recognize two different categories of threads in the emulator – those created by the emulator system, and those created by the application, and ensure that only the events of threads that are created by the application are tracked. To do this, we used Converse user-level thread API which allows a user to insert hooks to the thread scheduling events such as at the time of suspend and resume. When the emulator creates user-level threads for the application, it sets up special record-replay hooks for these threads that track thread suspend and resume in the same way as how it is done in the non-emulated Charm++. When it creates internal user-level threads, it does not set these hooks.

In Section 6.7.2, we will demonstrate how the BigSim Emulator is useful in reducing the number of processors using a real world application.

6.7 Performance

We evaluated the overhead of the proposed record-replay scheme for all three steps of the procedure. We used synthetic benchmarks, as well as two real applications.

6.7.1 Synthetic Benchmarks

Our first test environment was Abe cluster, at the National Center for Supercomputing Applications (NCSA). This is a cluster of 1200 Dell PowerEdge 1955 server computers, each configured with two quad-core Intel Xeon processors, 8 gigabytes of memory, and infiniBand interconnect.

We tested a synthetic benchmark program called kNeighbor, and evaluated the overhead imposed by recording message orderings with and without checksums. KNeighbor creates a certain number of objects distributed on the parallel machine, and arranged in a 1-dimensional array. In each iteration, each object sends $2 * K + 1$ messages to its nearest K neighbors on each side, plus a message to itself. When an object receives $2 * K + 1$ messages, it performs a given amount of computation, and proceeds to the next iteration. In the following experiments, we used $k = 2$, and the total number of iterations was 100.

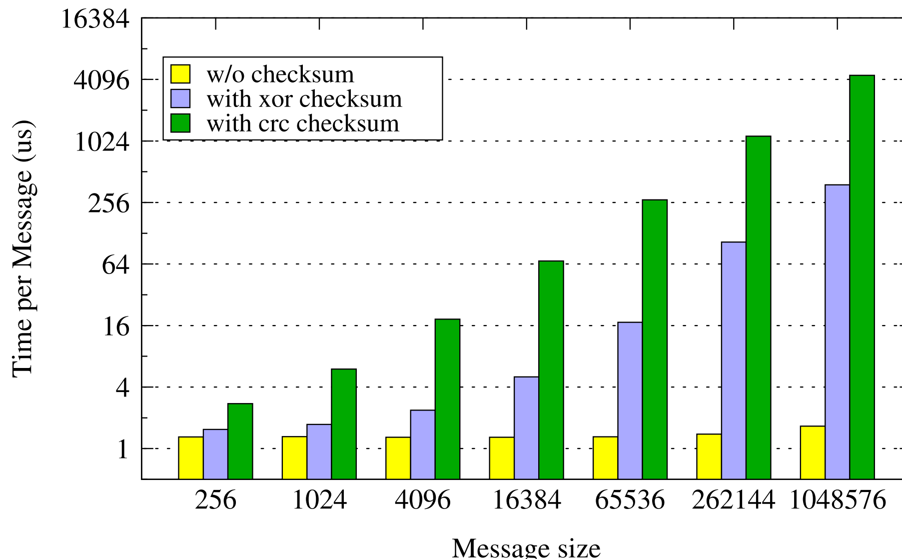


Figure 6.3: Recording overhead per message using the three schemes. Computed using kNeighbor test (NCSA Abe cluster).

First, we measured the average overhead per message during the recording phase in step one, varying the message size from 256 bytes to one megabyte. The results are shown in Figure 6.3. As expected, we can see that the overhead of the simple scheme remains about the same regardless of the message size. This is because, for each message, a constant amount of data is stored. When either XOR checksum or CRC checksum is calculated for each

message, the overhead per message increases proportionally to the increase in message sizes. This is because the runtime needs to traverse the whole message in order to compute the checksum. When checksums are computed, for very small messages, specifically 256 bytes, we observe only less than three microsecond overhead per message. However, when message size increases to one megabytes, both checksum methods incur a much higher overhead per message.

Next, we evaluated how this overhead affects the total execution of the program. To measure the total execution time, we ran the test program on a single processor. The total number of messages generated during each execution is about 5600. Again, we tested with a varying message size. Figure 6.4(a) shows the results of the comparison when the workload is very small. We can see that when the message size is small, 256 bytes, the total execution time is only 1.36 seconds for 100 iterations. When message size increases, the total execution time increases proportionally. This is due to the fact that the program has to process the message, and traverse all the data in it. We see that doing simple recording, without checksum, the execution time is not affected, even for large messages. Even with XOR checksum enabled, there is not much overhead. When switching to the more expensive CRC-based checksum, we can observe a significant overhead for large messages. However, for message sizes below 4KB, the overhead is still minimal.

When we increased the workload in Figure 6.4(b) (by 3 fold), and Figure 6.4(c) (by 6 fold), we observe a similar behavior. However, the results exhibit a decreasing overall affect of the CRC checksum computation, mainly due to the increasing computation-to-communication ratio. Similar results were also obtained on a different machine, called BluePrint, as shown in Figure 6.4(d). BluePrint is a Blue Waters [27] interim system also at the National Center for Supercomputing Applications (NCSA). It is a 2000-core IBM Power 5+ system.

These experiments show that simple recording scheme performs very well with almost no overhead to the execution time. XOR-based checksum is a cheap solution to improve the robustness of our scheme, and it incurs very little overhead. The more expensive CRC checksum computation indeed adds a significant overhead for very large messages. However, since most applications do not send large messages often, and if they do they generally perform a large computation thereafter, we believe this is not a problem for real-world applications.

To study the performance of the second step, we ran the kNeighbor benchmark on 256 processors of BluePrint. The results are illustrated in Figure 6.5. In each cluster in the figure, the first bar from the left is the total execution time without any overhead; the second bar represents the execution time when replaying on the same machine using the traces from step one; while the third bar represents the execution time of the benchmark both replaying the previously recorded message ordering, and recording the detailed information for one processor. We see that replaying on the full machine generally is slightly slower than the normal execution time for message sizes smaller than 64K bytes. However, for very large messages, the replay time tends to increase more drastically.

The number of processors recorded in detail during step two may affect the replay time due to file I/O. Figure 6.6 illustrates the effect on the total execution time of kNeighbor

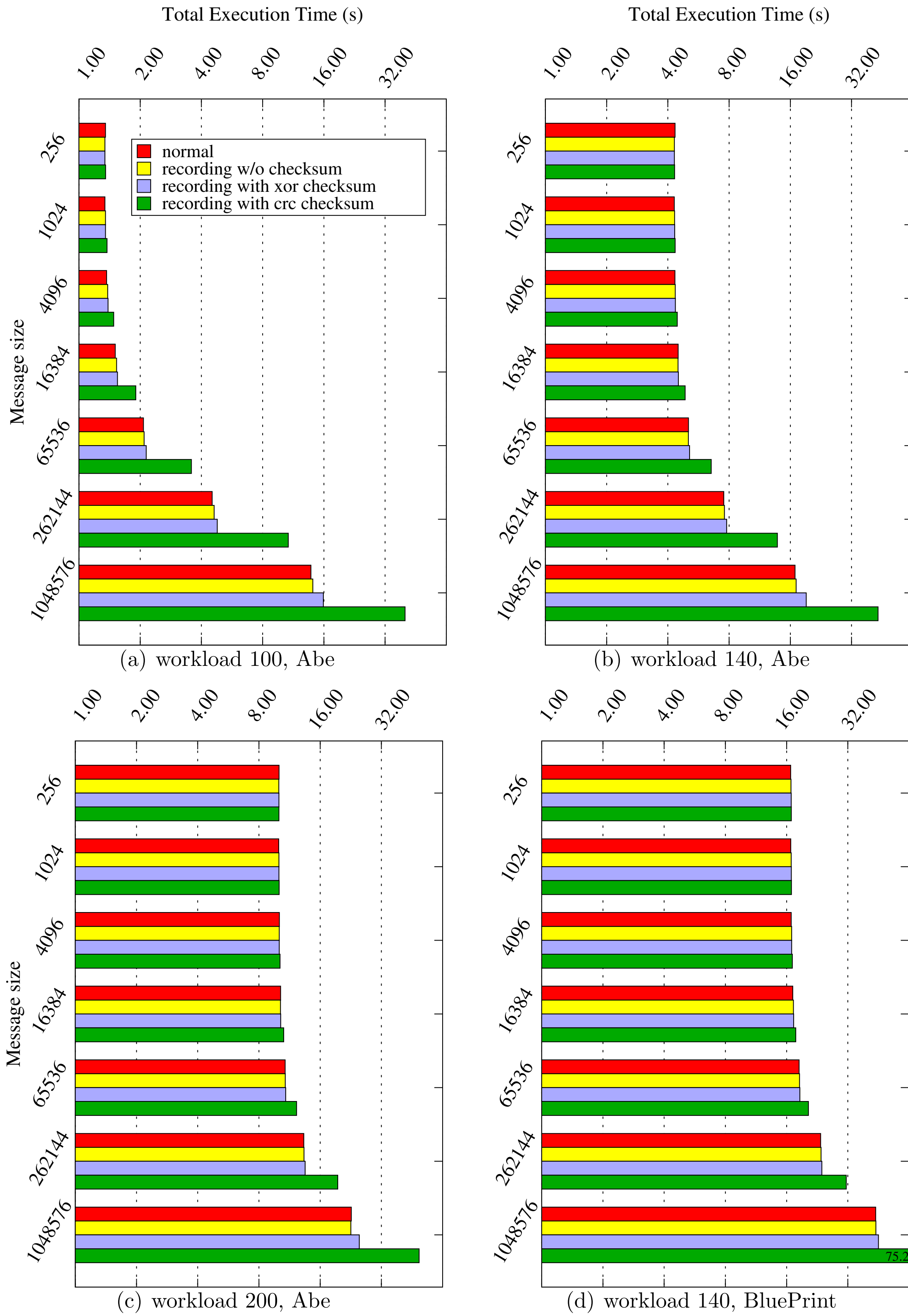


Figure 6.4: Comparison of kNeighbor total execution time with and without recording schemes (the total time includes file I/O at the end of execution).

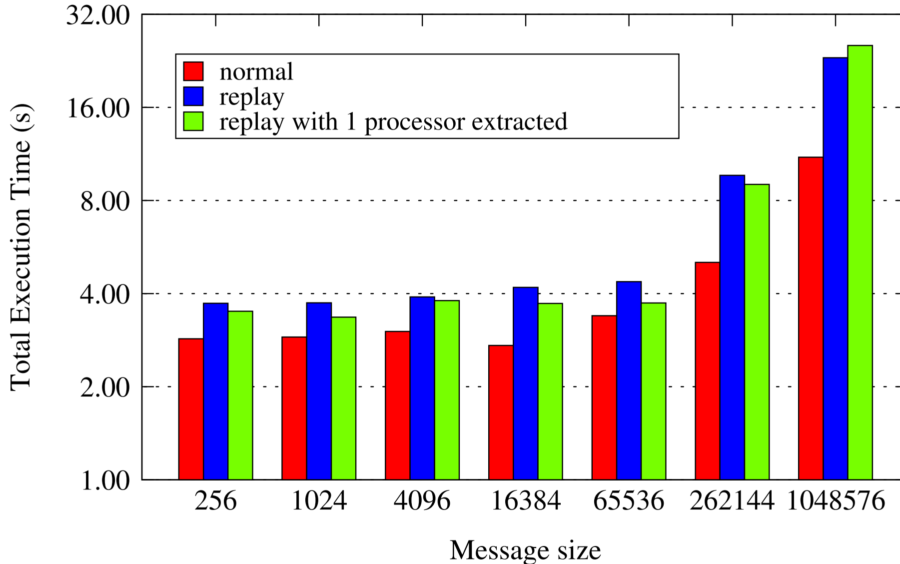


Figure 6.5: Total replay time in step two for kNeighbor (NCSA BluePrint 256 processors).

when varying the number of recorded processors during step two on 256 processors. We can see that for messages smaller than 1KB, the effect of recording full traces is minimal. As expected, when the message size increases, the overhead of recording message contents for more processors increases significantly due to the file system becoming a bottleneck. However, as we explained, this does not affect the correctness of the replay, since the message ordering is already guaranteed. In practice, we believe that a user does not need to extract all processors, and only a small subset of processors is usually enough to understand the nature of a bug.

6.7.2 Scientific Applications

In addition to synthetic benchmarks, we used two production-level scientific applications to show the performance impact of our approach. The two applications are ChaNGa and NAMD.

ChaNGa

ChaNGa [18] is a cosmological application used for the simulations of the evolution of the universe. It handles forces generated by both gravitational and hydrodynamic interaction. The benchmark we used is a snapshot of a multi-resolution simulation of a dwarf galaxy forming in a $28.5Mpc^3$ volume of the universe, with 30% dark matter and 70% dark energy. The dataset size is nearly five million particles, with most of the particles clustered in the center of the simulated volume. In our tests, we ran the application for three timesteps.

Figure 6.7 shows the performance of ChaNGa in step one, using a varying number of processors, on the NCSA BluePrint cluster. Each execution was repeated five times, and

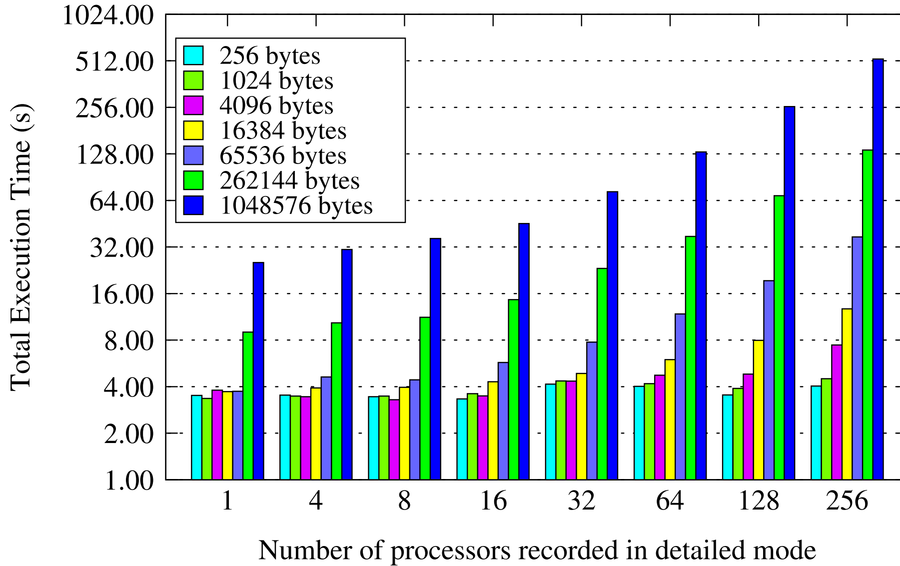


Figure 6.6: Comparison of kNeighbor total execution time in step two when recording varying number of processors in full detailed mode. (NCSA BluePrint 256 processors).

the average and standard deviation are plotted. As mentioned earlier, we could run the optimized code for step one of our 3-step procedure. As a comparison, the black bar (to the left) represents the execution time with a non-optimized version. The optimized code is more than twice as fast as the non-optimized one, and the interleaving of messages potentially very different. It can be seen that even on a highly optimized code the impact of the recording schemes is so small that it disappears when compared to the normal time fluctuation of ChaNGa, even when computing checksums. This re-emphasizes the negligible perturbation caused by our recording scheme.

Subsequently, with the recorded data from step one, we proceeded to test steps two and three. These are plotted in Figure 6.8. Again, each execution was repeated five times. In this case, we had to use the non-optimized version to be able to follow the code in a sequential debugger. Compared to the execution without record-replay enabled, the forced replay of the message ordering caused an overhead between 25% and 65%. This overhead is still very small considered to the potential that the scheme yields in terms of allowing a deterministic debugging.

Step three is represented on the fourth bar of each cluster in Figure 6.8. As different processors may have a different workload (we didn't apply load balancing), the variation in the execution time between different processors is very large. Surprisingly, the execution time of a single processor was greater than the time to execute the whole application. We suspect this might depend on the system pre-loading too many messages from the traces, and we plan to further investigate the reasons. Nevertheless, even with the current performance, the replay time is within a factor of two from the basic execution.

In addition to the overhead caused during the execution of the application, we also mea-

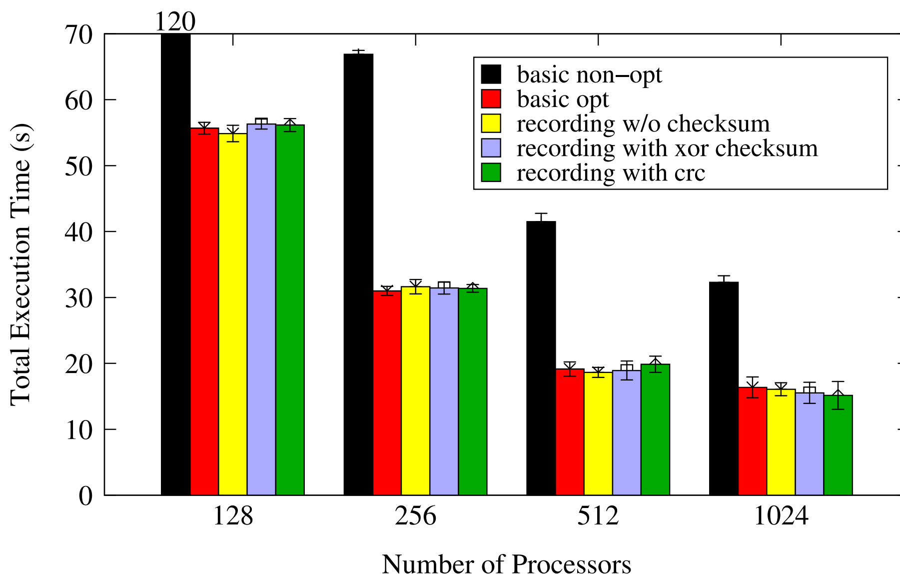


Figure 6.7: Recording overhead for ChaNGa application using the three schemes (on NCSA BluePrint cluster). The tests were performed with optimized code, except for the first bar in black.

sured the amount of information that is stored to disk during the various phases. Table 6.1 reports this information in megabytes. As it can be seen, the amount of information recorded per processor is quite small—less than one megabyte—and can be easily maintained completely in memory until the application shuts down. Therefore, flushing to disk is generally avoided during the first step. During the second step, we can see that the amount of data recorded is much larger. Nevertheless, this does not create a problem since usually only few processors are recorded in detail.

Number of processors		128	256	512	1024
Record	**per-proc.**	**0.87**	**0.67**	**0.54**	**0.44**
	total	112	173	279	453
Record+checksum	**per-proc.**	**1.49**	**1.14**	**0.92**	**0.75**
	total	190	292	473	765
Detailed record	**per-proc.**	**111**	**79**	**59**	**47**

Table 6.1: Amount of data stored to disk using different recording schemes for ChaNGa. All data in megabytes.

NAMD on BigSim Emulator

In this section, we demonstrate the utility of using BigSim to perform processor extraction using the 3-step procedure. The application we chose for this is NAMD [16, 26].

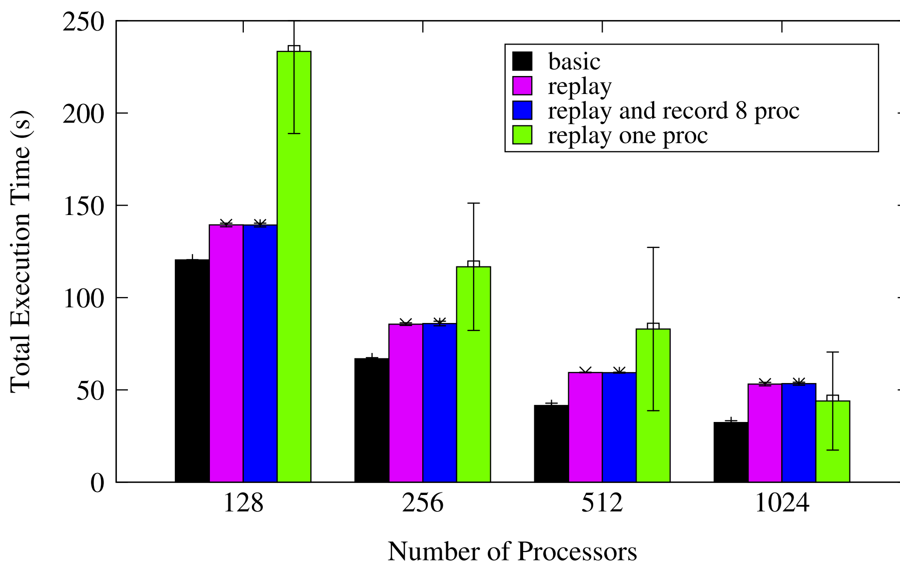


Figure 6.8: Overhead during replay in steps two and three for ChaNGa application (on NCSA BluePrint cluster).

NAMD is a scalable parallel application for Molecular Dynamics simulations written using the CHARM++ programming model. It is used for the simulation of biomolecules, and to understand their function. The following experiments were done on the NCSA BluePrint system.

First, we benchmarked step one by running NAMD on 1024 processors using the Apolipoprotein-A1 (ApoA1) benchmark for 100 timesteps, and repeating using all the recording schemes. The results are shown in Table 6.2. The "normal" column is the time when running without recording for comparison. Note that the total number of messages processed during the entire execution is about 20,000.

Mode	Normal	Record	Rec.+XOR	Rec.+CRC
NAMD Time	24.08	25.33	24.55	24.55
with I/O	-	27.99	26.85	25.82

Table 6.2: NAMD execution time in seconds with different recording schemes in step one, running on 1024 processors of NCSA BluePrint. The last row is the total time with file I/O.

We see that there is virtually no overhead to the NAMD actual execution while recording the message ordering, even when checksums are computed. This is because in NAMD the average message size is relatively small, around 1KB to 2KB. For 20,000 messages total, even the most expensive scheme using CRC checksum only cost about 0.27 second. Therefore, we believe that by using our recording schemes, the NAMD application behavior is not affected significantly. Furthermore, due to cache effects, the actual overhead of computing checksum may be even less, if the entry function triggered by the receipt of the message has to traverse the message data immediately. Similar to CHANGA, the NAMD traces recorded for each

processor are less than one megabyte in size. The process of flushing the traces to disk takes about 2 seconds, which increases the total execution time, as shown in the second row of the table. The file I/O time is constrained by the bandwidth of the file system, and may be stressed by simultaneous writing, in this case by 1024 processors. However, since this is done only at the very end of the execution, it does not affect the ordering of the messages during execution.

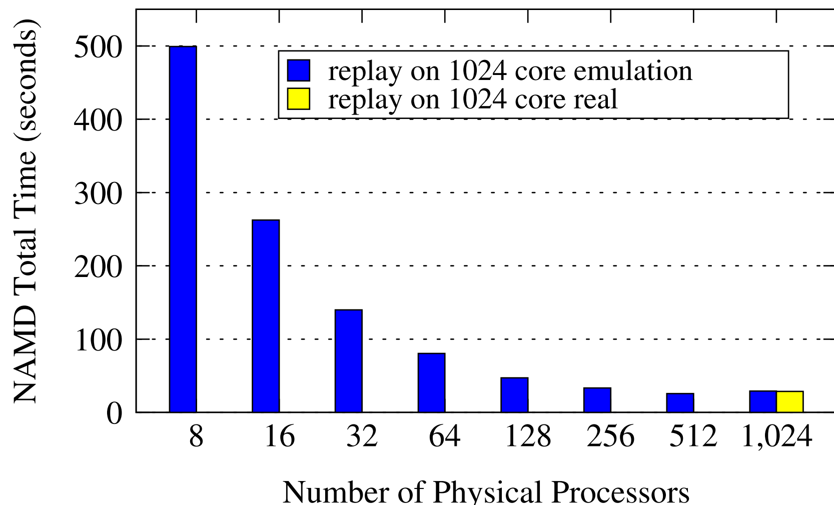


Figure 6.9: NAMD execution time in replay mode on 1024 emulated processors using varying numbers of physical processors, recording 16 emulated processors. The last bar is the actual runtime in the non-emulated replay mode on 1024 processors. (on NCSA BluePrint).

For the second step, we ran it under the BigSim Emulator, and we replayed NAMD using the message logs obtained from the first step. We instructed the emulator to emulate the same 1024 processors by using only a portion of the entire machine. While replaying, we also chose 16 emulated processors for detailed recording of message content. We measured NAMD total execution time running on the emulator using varying number of physical processors. The results are shown in Figure 6.9. When 1024 physical processors are used to emulate the 1024-processor machine, we see that replaying NAMD on the emulator is about as fast as when replaying it on the real 1024 processor BluePrint machine, showing little overhead of the emulator. Moreover, on 512 processors, NAMD replaying in the emulation mode is even slightly faster than the actual replaying run on 1024 processors. This is due to the saving in the startup: faster global synchronization on fewer nodes.

This demonstrates that in terms of the time-cost, it is feasible to replay an application in a virtualized environment under the emulator using fewer processors. Although it takes much longer (17 times slowdown) to replay NAMD under the emulator when using only 8 physical processors, being able to replay an application on a much smaller machine, and generate detailed trace logs, greatly reduces the need for large machine during interactive debugging.

```
BgReplay> Emulation replay finished at 25.304625
          due to end of log.
BgReplay> Replayed 12288 local records and 7891 remote
          records, trace log is of 14539488 bytes.
```

(a) Processor 0

```
BgReplay> Emulation replay finished at 5.690778
          due to end of log.
BgReplay> Replayed 19714 local records and 10822 remote
          records, trace log is of 24904148 bytes.
```

(b) Processor 960

Figure 6.10: Screen outputs from replaying two different processors under the emulator.

In the third step, the detailed NAMD trace logs recorded in the second step were used to replay a selected processor using a single processor. Figure 6.10 shows the last few lines of screen output of replaying processors number 0 and 960 on the emulator respectively. For this benchmark, each detailed trace log was about one to two megabytes, as shown in the output. The replay time of processor 0 on the emulator finished in about 23.8 seconds, which matches the total execution time of 24 seconds when running NAMD in normal parallel execution. The replay time of processor 960, however, took much less time (only 5.5 seconds). This is because during NAMD start-up, most of the work is done by processor 0, and the other processors are mostly idle. Since this is a short simulation that has only 100 timesteps, most time was spent in the start up. On 1024 processors, the start-up time is measured around 19 seconds.

In summary, this example of NAMD on the BigSim emulator demonstrates that it is feasible to use an emulator in the second step as an alternative way of replaying an application using the message ordering logs obtained from the previous step, and producing detailed trace logs to be used in the third step. This approach incurs reasonably low overhead, and the overhead itself can be considered proportional to the reduced number of physical processors used for emulation.

6.8 Case Study

To assess the usability of our technique, we used the ChaNGa application, and searched for the bug we mentioned at the beginning of this chapter. This bug has already been fixed using standard techniques, such as print statements, and a tedious process given that the bug often disappeared after code modifications. We re-introduced it in the application temporarily. We ran the application using a relatively small dataset (a simulation of a LCDM concordance cosmology large volume with 48^3 particles and 300 Mpc on a side). The bug did not appear on eight processors, but started to appear on sixteen processors or more. The manifestation was intermittent, sometimes right at the beginning, sometimes after a few timesteps of the application. Also, the processor in which an assertion failed kept changing from execution

to execution.

According to our 3-step procedure, we first executed the application with the message ordering recorded. We used CRC checksum as robustness protection. In the execution we recorded, processor seven triggered the assertion. At this point we re-executed the application in replay mode, and recorded the faulty processor. We also repeated the execution in replay mode a few times to confirm that processor seven was always the culprit. With the detailed trace of processor seven, we executed ChaNGa sequentially under CharmDebug, and followed the problem. To track the bug we had to repeat the sequential execution a couple of times, each time setting a few different breakpoints. Compared to the way the original bug was hunted, this new procedure allowed for the parallel problem to be transposed into a sequential one, without compromising the timing of the application, and without allowing the problem itself to disappear.

```
../charmrun +p16 ../ChaNGa cube300.param +record +recplay-crc
../charmrun +p16 ../ChaNGa cube300.param +replay \
        +recplay-crc +record-detail 7
gdb ../ChaNGa
>> run cube300.param +replay-detail 7/16
```

Figure 6.11: 3-step procedure used for debugging ChaNGa.

To perform the processor extraction, and subsequent analysis of the extracted processors, the user can either use the command line interface to run directly his application or use CharmDebuggraphical tool. The command line interface is especially useful when jobs have to be submitted through a batch scheduler. This, of course, can be true for the first two steps of our procedure. The commands we used in our example case study are reported in Figure 6.11. The last command included is the launching of the GDB debugger, which is an alternative to CharmDebug in following the faulty processor during step three.

The alternative with CharmDebug is presented in Figures 6.12 and 6.13. The first one is a screenshot of the window where the input parameters can be set. In this view, the user can select which record-replay step he would like to perform on the left side, and which protection mechanism he would like to use (if any) on the right. The view shows the setup for the execution of the third step of our example case study. Figure 6.13 shows instead the program being executed under the controlled environment in step three. Here, the user can proceed in his debugging analysis using the selected processor (in our case 7). This processor is the only one available, as shown by the circled drop-down box. The user can also inspect the application status as he normally would (on the lower part of the view).

6.9 Future Work

In this chapter we described a new procedure to extract processors from a parallel application, and replay any of them on a local cluster. This allows a programmer to debug his application using a local workstation, or a small cluster, even when the application being

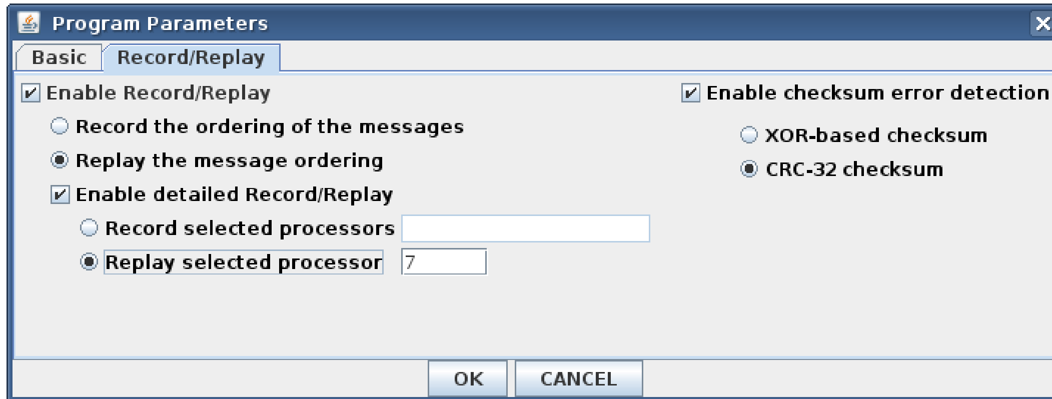


Figure 6.12: CharmDebug window to set the record-replay parameters. This view shows an execution of the third step of the procedure.

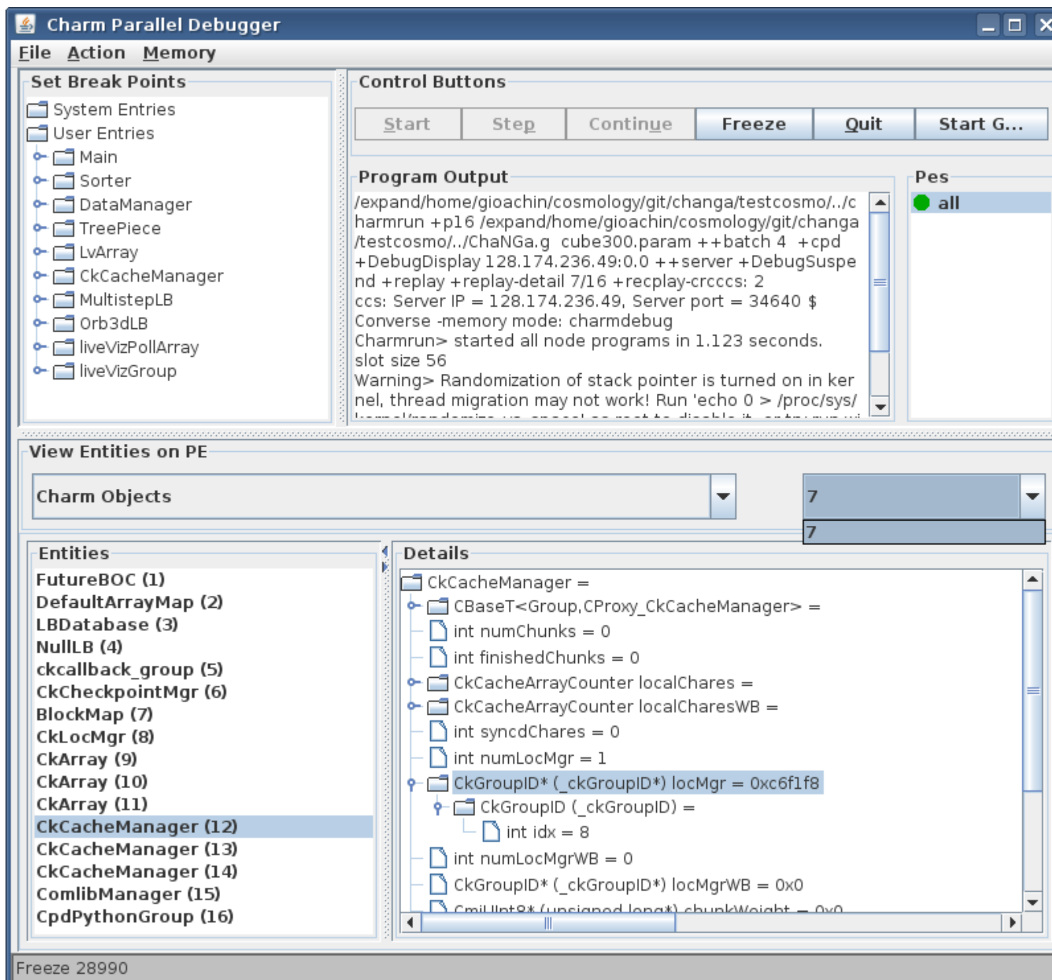


Figure 6.13: CharmDebug's main view during the debugging of an application on the third step of the procedure. Only one processor is available.

debugged misbehaves only on very large configurations. It allows to decouple the long and slow debugging process involving the user understanding the problem in his application from the need to have a large machine allocation available during that whole period of time.

There are several research directions that are possible for the future. One of them regards the possibility to use different executables during different steps. As mentioned in Section 6.5, if step two has been executed using a specific communication layer or parallel machine, the user may be bound to that architecture also in phase three. Some work has been done in order to be able to translate messages from one architecture to another [86]. Integrating this capability into the record-replay scheme will allow developers to further decouple the use of a parallel machine to their specific needs during debugging.

Another direction is to detect other information that our current scheme is lacking, and include that into the recorded traces. For example, when creating single chares in Charm++, the system may not be fully deterministic, and decide to inserted the new chare into an underloaded processor. When replaying the system, these chares will have to be inserted into the same processors as the original execution in order to preserve the correct execution of the program.

7 Provisional Message Delivery

As illustrated in the previous chapter, by recording the message ordering during execution, and replaying it at will thereafter, a programmer can reproduce an application's bug identically until the cause is found. This procedure has a drawback: it requires a first execution, in which the bug manifests itself, to be captured. Sometimes the bug appears at every execution, only on different processors each time; for this kind of problems standard record-replay techniques work fine. Other times, the bug may appear sporadically and recording the bug may be a challenging task in itself. For example, imagine a situation where processor A sends two messages: α to processor C, and β to processor B; B, upon receiving β, sends message γ to C; and the application misbehaves only when γ is processed before α. If α is sent before β, α will generally arrive at C before γ, and execute first. Nevertheless, α could be lost and require retransmission on the network, making γ execute first. This kind of scenario is difficult to capture, as it may occur only once every thousands of executions. Increasing the size of the machine may trigger the problem more frequently, for example α and β could take different paths in the underlying hardware network. Nevertheless, the problem remains: how many times does the user have to repeat the experiment before obtaining a trace with the bug? If the crash happens late in a program's execution, how many resources will be consumed in the recording process?

Another problem that may hinder the recording of the bug is if the recording itself corrupts the timings of the communications. While we have taken special care to be as unintrusive as possible, it cannot be taken for granted that the bug will appear. For some applications, moreover, the minimal recording necessary to capture the bug may not be the simple message ordering, but it may include other data, like timers or other system calls. Again, how long is the user willing to try and see if the bug can be captured during a recorded execution?

Record-replay techniques guarantee that messages are processed during replay in the same order in which they were processed during the recording phase. The opposite is also possible: forcing messages to be delivered out-of-order. This may expose the bug very early in the application, and may not require large machines to be allocated to discover the problem. For instance, in the example above, where messages α and β were racing, one could try imposing the delivery of β before α even though α is in the queue before β. This operation can be done as easily on a small machine as on a big one, thus relaxing the need for large machines to obtain a suitable trace for the record-replay technique. We shall see later in Section 7.7 why this randomization of messages in the queue cannot be trivially performed automatically.

When looking at the messages enqueued on a given processor, there are two ways to

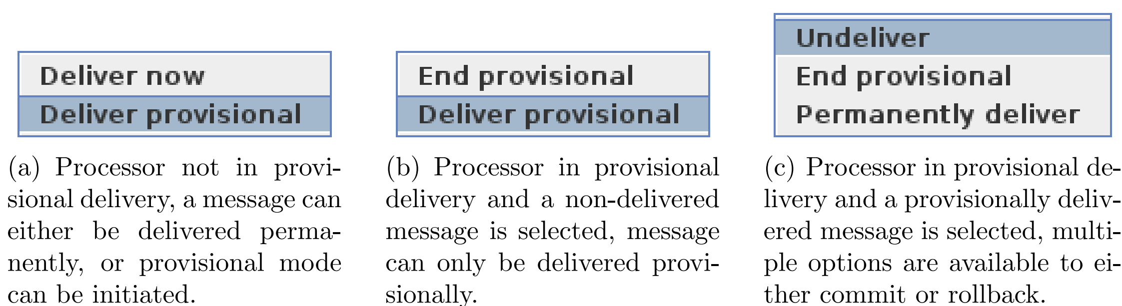
determine what to process next. One is having the user try certain combinations. These combinations of messages could be random, or due to his understanding of the program, or a hunch he might have. The other, more automatic, would be to have the system explore the possible delivery orderings, and report to the user when different orderings generate different solutions. In this thesis, we shall lay the foundations to allow the testing of possible different execution paths, but consider only paths specified by the user. Towards the end of the chapter we shall expand on automatic search, and further challenges that it poses.

7.1 Provisionally Delivering Messages

When considering how the user should be allowed to interact with the system to test his hypothesis, several decisions can be made. In particular, these involve how the system will perform the delivery operation, and how it will allow the rollback if the user decides to undo the delivery operation. We call these delivery operations "provisional" for their property of being not fully committed into the application, and the possibility to annul them. Before entering into the details of the system, let us start with the user's perspective.

In a typical debugging scenario, the user will select a message from a processor's queue, and issue the delivery command. This can be performed by using the CharmDebug's GUI with a simple mouse click. The options available depend on the state of the system and the message selected. Figure 7.1 shows the different options available. When a processor is not in *provisional mode* (Figure 7.1(a)), the user can either permanently deliver a specific message or initiate the provisional delivery with the selected message. Once inside this mode, messages still in the queue can only be delivered provisionally (Figure 7.1(b)). For messages that have been provisionally delivered (Figure 7.1(c)), two options are available: 1) rollback the system until the selected message has not been delivered, or 2) commit the message permanently on the processor. The user can distinguish between messages already delivered provisionally and messages still in the queue graphically: messages already delivered provisionally are shown in purple at the beginning of the queue (Figure 7.2).

Suppose the user has decided to provisionally deliver messages from the queue in the following order: α, β, γ, δ, ϵ. After provisionally executing this sequence, and examining the resultant state, the user may select the third message (γ), and using one of the options



(a) Processor not in provisional delivery, a message can either be delivered permanently, or provisional mode can be initiated.

(b) Processor in provisional delivery and a non-delivered message is selected, message can only be delivered provisionally.

(c) Processor in provisional delivery and a provisionally delivered message is selected, multiple options are available to either commit or rollback.

Figure 7.1: Options available for different system status and message selected.

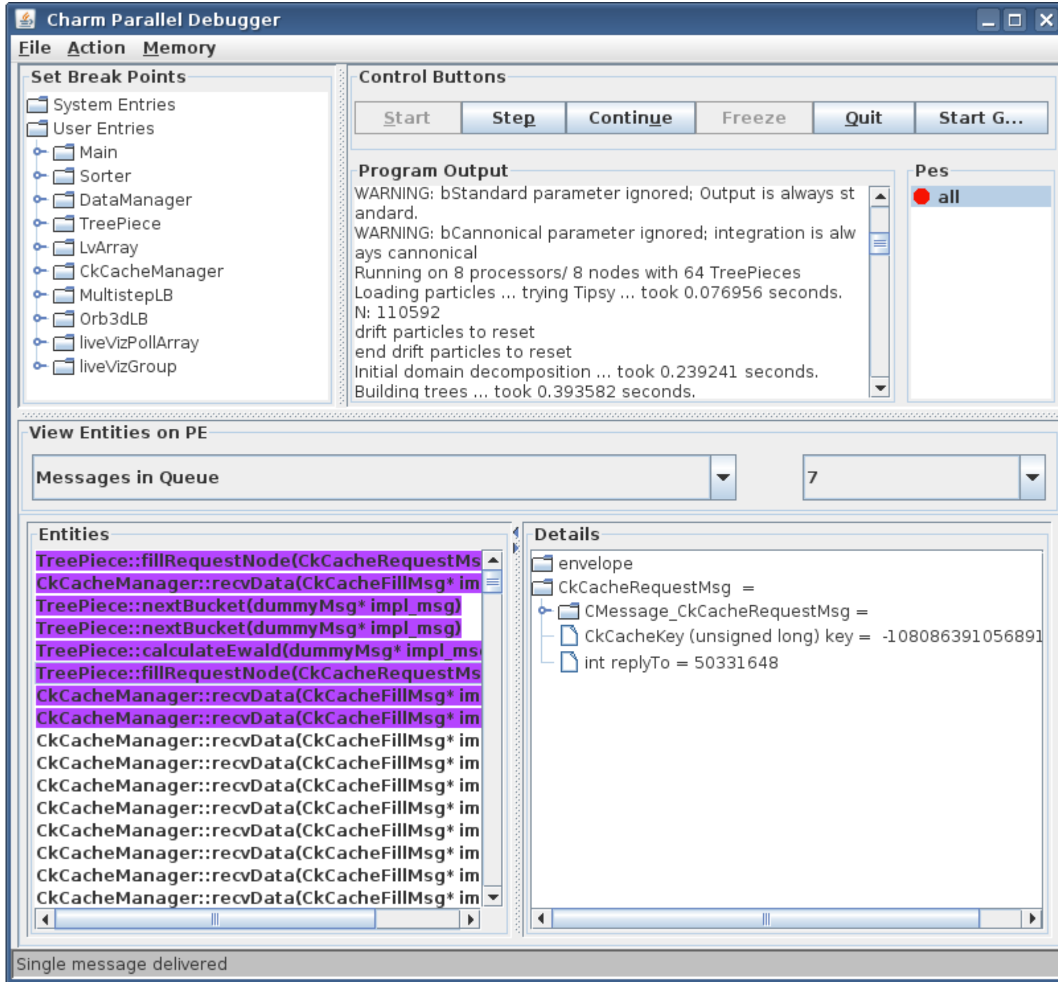


Figure 7.2: Screenshot of the message queue. Some messages have been delivered provisionally (in purple on top), while others are still in the regular queue.

shown at that time, decide to roll back the system and remain in provisional mode with messages α and β, and have messages γ, δ, and ϵ returned to the normal (undelivered) queue. Alternatively, he can permanently deliver message γ and all its predecessors (therefore messages α, β, and γ in order). Messages δ and ϵ will be left as provisionally delivered. It is important to notice that when committing message γ we cannot ignore messages α and β since they were provisionally delivered before message γ. If we ignored them, the system may behave differently, as the order in which messages are processed would have changed. In particular, it could terminate abnormally. The same discussion applies also for messages δ and ϵ when rolling back. For the rollback case, once the system has voided the changes provoked by the processing of the three messages, an option could be given to the user to provisionally re-deliver messages δ and ϵ.

With this execution flow in mind, the system must meet certain conditions to be useful. First of all, it ought to survive crashes when in provisional mode. During normal execution,

when a failure appears (such as an assertion failure or a termination signal), the system freezes the faulty processor for inspection by the user. While the status of the crashed processor is still retrievable, the program cannot continue execution beyond the crashing point. This is because the computation executed might have left the processor in an unclean state. Therefore the user can only restart the application after he finishes inspecting it. In the case of a message provisionally delivered causing a fault in the application, the user must still have the capability to roll back the application to the point in time before the crash, when the faulty message had not been delivered. From this rolled back state, the user must be able to continue execution normally, maybe specifying a different message to be delivered.

Another condition to be met by the system is that it should be usable interactively. The user may want to try different options quickly, and see if the system produces the expected output. If the system has a long response time, the debugging may become impractical.

7.2 Exploring Solutions

We considered several possible alternatives to deliver messages provisionally before committing to a specific implementation. First we considered the possibility to restart the application from the beginning at every rollback. This approach has the advantage of requiring the least amount of changes to the runtime system, and only have the debugger control the termination and restart of the execution. It also provides a clean environment not corrupted from the delivery of the messages provisionally.

This approach has several critical problems, the major being its performance. By restarting the application at every rollback, the whole initialization process has to be performed over and over again, and it can take a significant amount of time. Moreover, the application might already be at an advanced stage in the execution, possibly requiring a very long time to re-execute. Another obstacle is the difficulty to restart the application. Job schedulers deployed on parallel machines may decide to terminate the processor allocation when the application ends, therefore making it impossible to restart a new execution without waiting in the queue for a new allocation. Moreover, if the user desires to provisionally deliver messages on more than one processor, in order to roll back a single processor, the whole system will have to suffer a full rollback, and every processor that is still in provisional mode ought to re-deliver the messages provisionally. Record-replay techniques are also needed to guarantee the delivery of the messages in the same order up to the point where the user has started the provisional delivery.

To prevent having to restart the application from the beginning, a different approach could be used with support from fault tolerance protocols. The debugger could issue a global checkpoint command when initiating a provisional delivery, and simulate a processor fault when it needs to roll back. The fault tolerance scheme will internally roll back the application to the point in time when the checkpoint was taken. The time to roll back now depends on the time that the fault tolerance mechanism will take to restart the application from the checkpoint. The basic technique of storing the checkpoint to disk and restart the application by loading the image from disk has similar disadvantages as a full restart: it

may still have to wait in the job scheduler's queue, and it will force the rollback of all the processors in the system. It will only avoid the complication of record-replay techniques to guarantee message ordering.

The other two fault tolerance schemes present in Charm++ can provide better support to cover the problems mentioned above. Double in-memory checkpoint [84] can tolerate the rollback without having to restart the application from disk, and therefore avoiding potential problems with job schedulers. Message logging [87] can further avoid the rollback of all the processors when only one needs to terminate the provisional mode. Nevertheless, in the current implementation, the processor that is rolled back due to a fault, real or simulated, is supposed to be a newly started process which has to join the set of the already running processes composing the rest of the application. For example, the underlying network communication system will have to be updated to reflect the change of process. In LAPI or MPI, communicators will most likely have to be re-instantiated; in UDP, port numbers will need to be re-synchronized.

One disadvantage of all fault tolerance schemes is that they often require some modification in the application to support it, and will result in a non-fully transparent approach. For example, in Charm++, the user will have to explicitly provide Pack/Unpack routines capable of migrating all the objects to and from storage. These routines are generally not needed without fault tolerance. Even when these routines are present, they might not allow the checkpoint to happen at any given point in time, as they may be optimized for performance by restricting the point in time where a checkpoint can be taken. With all the different approaches currently available, there is still the problem of guaranteeing that the state of the application after the restart is identical to that before the checkpoint was taken. Memory layout can be significantly different, and the application's control flow may also be modified. These differences may hinder the determinism of the execution and void the whole provisional delivery scheme.

The last approach that was considered, and was later decided upon, uses `fork` system calls to spawn a new subordinate process to carry on the provisional delivery without touching all the intricate connections already established between the application's processors. The parent process of the fork operation always contains the saved state of the application, and by reverting the control back to it, the application can be easily rolled back. This scheme involves neither disk I/O nor job schedulers. Moreover, the capability of operating systems to perform copy-on-write of the virtual address space during the fork operation allows for fast switch between provisional delivery mode and normal mode. This approach also offers the advantage that, upon rollback, the state of the application is exactly as it was at the moment the application entered provisional delivery mode, including the memory layout. This makes it easier to track bugs that depend on the relative memory location between distinctly allocated memory blocks.

One aspect that is not covered by the process forking approach is that input and output operations are not masked by the runtime system. This implies that if the execution of a particular entry method prints a string to standard output, and the user later rolls back the execution of that entry method, the printed string will not be deleted from the output

stream, and a new execution of the method will print the string again. The same is valid for operations on open files or other system calls. In particular, stored data could be corrupted. Solutions can be built to avoid this problem, for example by intercepting certain system calls, and providing a provisional execution environment where also input/output is treated correctly. Nevertheless, for the purposes of this thesis, these issues will not be considered further, and are treated as future work.

This solution using process forking, as well as the others, builds upon the piecewise deterministic assumption described in Section 6.3. If the outcome of a computation changes depending on conditions other than the state of the system and the content of the message processed, provisionally delivering a message at a certain point in time would yield a different result each time. This would make testing executions paths harder since the user will have to consider the possibility that re-delivering a message could produce a different result each time. Fortunately, applications tend be behave piecewise deterministically, therefore not hindering the applicability of the methods illustrated. For applications not in this category, more robust solutions can be sought as an extension of this work.

7.3 Implementation

Each processor in the application is treated independently from all others, and the user is allowed to independently decide to deliver a message provisionally on any of them. The automata describing the behavior of a given processor is presented in Figure 7.3. When in normal mode, the user can decide to deliver a message immediately, and remain in normal mode, or provisionally, and transition to the provisional mode. In both cases, the entry method associated with the delivered message is invoked on the target processor. When in provisional mode, the user can either deliver more messages provisionally, or rollback and undeliver some messages. If a rollback is performed, the processor transitions to normal mode only if all the messages that have been provisionally delivered on the processor are undelivered, otherwise it remains in provisional mode.

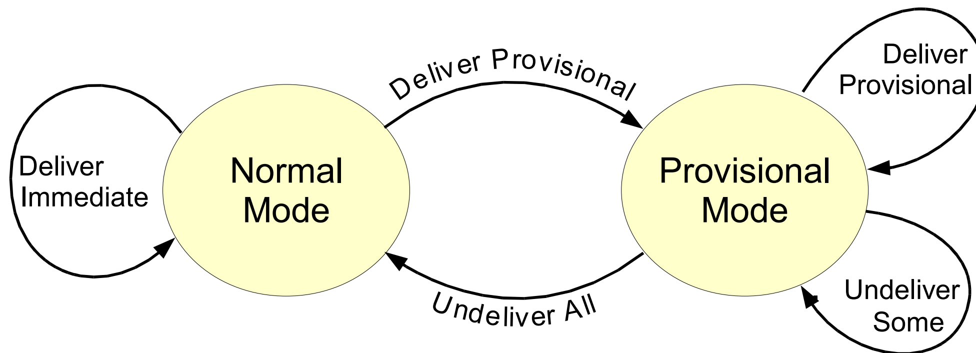


Figure 7.3: Description of the behavior of a processor when provisionally delivering messages.

As discussed earlier, we decided to adopt a solution based on the `fork` system call. When the system is in normal mode, messages from other processors and CCS requests are enqueued in the local processor's queue, the latter are also processed immediately by the system scheduler. This is illustrated in Figure 7.4 for process *Pe X*. When instead the system enters provisional mode, a new process is forked, and the message is delivered in the child process. When the user decides to rollback the application, the child is destroyed and the parent resumes execution. We shall consider multiple message delivery in the following section.

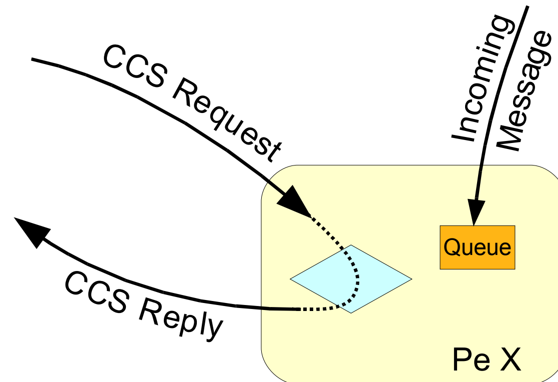
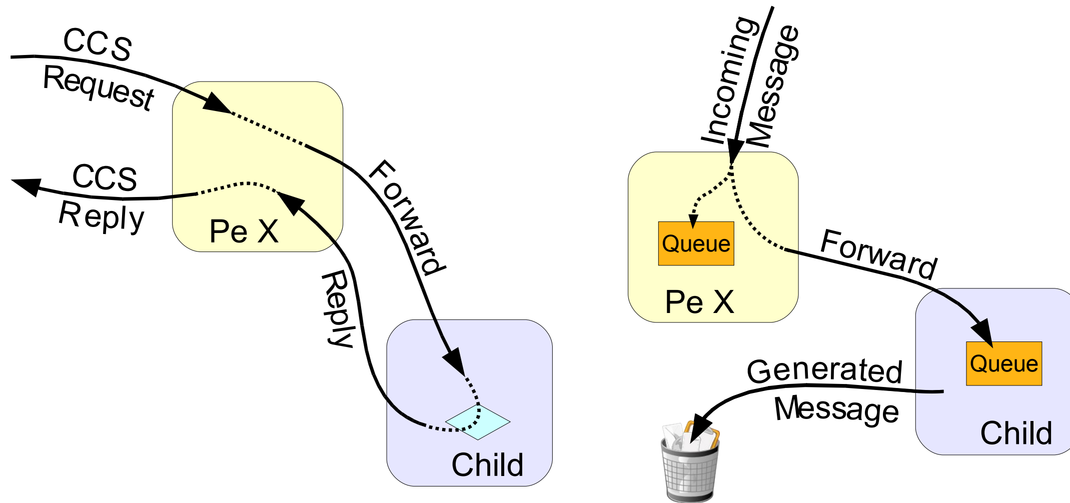


Figure 7.4: Control flow of a processor in normal mode. When a CCS request arrives, the processor handles it and replies to the sender.

In theory, the provisional message could be delivered either in the parent or in the child process. Nevertheless, delivering it in the parent process has several drawbacks. If the parent were to deliver the provisional message, then its memory state would be modified, and only the child would be able to continue execution after rollback. Unfortunately, if the parent terminates, then the entire application may be terminated by the job scheduler which will perceive one of the application's processes ending execution. Instead, terminating the child process has no consequence. Furthermore, having the child process continue execution and use the communication infrastructure is a more fragile solution since in some implementations only the parent may be allowed to use the communication device.

Figure 7.5 illustrates the control flow of processor *Pe X* in provisional mode when either a CCS request or an internal message arrives. In all scenarios, it can be seen that the parent process is always in charge of the communication with the external world, and the child process only communicates internally with its parent. This is to prevent potential corruption of the network state if the child were to use it directly. The communication between the two processes happens mainly through an anonymous pipe. The most common scenario is in Figure 7.5(a) since CCS requests are likely to arrive from the CharmDebug GUI as a consequence of an action from the user. In this case, the request is forwarded to the child for handling, and then the reply is forwarded back to the client. Note that only the



(a) Response to a CCS external message.

(b) Response to an internal message from another processor in the application.

Figure 7.5: Control flow of a processor when in provisional mode.

child process is capable of correctly handling the request for $Pe\ X$ since its memory reflects the delivery of the message. For example, if the provisionally delivered message changed a local variable Var from 5 to 3, then the parent would answer 5 to a request for Var, while the child will answer with the correct value 3.

When a regular message (α) arrives from another processor in the application, as shown in Figure 7.5(b), this message is both enqueued in the parent's local queue, as well as forwarded to the child where it will be enqueued in the local queue as well. For correctness, the message α must be enqueued in both processes. It must be enqueued in the parent process since it still has to appear in the processor's queue after rollback, when the child is gone. If it was not recorded by the parent, once the child terminates execution, α would be lost. It has to be received by the child process otherwise the CharmDebug GUI would not display it: remember that the list of messages enqueued on a processor is gathered through a CCS request that, as just described, is handled by the child process during provisional mode.

It may appear that regular messages cannot arrive on a processor while it is in provisional mode. However, regular messages can be received by a processor in provisional mode for at least two reasons. 1) Some other processor in the system has not suspended execution, and is still processing messages normally and sending out messages as a result. 2) The system is entirely suspended, but the user issued an immediate delivery command on a processor, and a message was generated as a consequence.

While handling a provisionally delivered message α, the child process may possibly generate some message β. If this message β were permitted to leave processor X and reach its destination Y, then there would be a causal dependency from processor X to processor Y on the order in which messages have been delivered on processor X. If the destination processor handles β, and then the user decides to rollback the delivery of α on X, the execution of β

on Y must also be rolled back since β has not been created by X anymore. This implies that processor Y must be able to rollback and undeliver β. In our implementation, we solve this problem by not allowing any message to cross the boundary of a processor until the entry method which generated the message has been committed by the user. Thus any message generated as a consequence of a provisional delivery is discarded.

One could envision an extension to our system where messages like β are allowed to leave the boundary of a processor, and can be delivered provisionally on the destination processor. Naturally the dependency introduced has to be tracked and treated accordingly. If the source message α is undelivered, then an undeliver command must also be issued on message β. Conversely, if β is permanently committed, also message α (and all its predecessors) ought to be permanently committed. This dependency can clearly be chained several times, thus producing potentially complicated dependency graphs.

7.3.1 Delivery of Multiple Messages

Until now we have discussed how we can deliver a single message provisionally, without considering what happens when multiple messages are delivered provisionally. We shall now extend our system to include multiple subsequent messages provisionally delivered. In this scenario, we want the capability to roll back the application to any point in time between message deliveries. As the system becomes more complicated, we shall introduce another communication mechanism between the forked processes: shared memory.

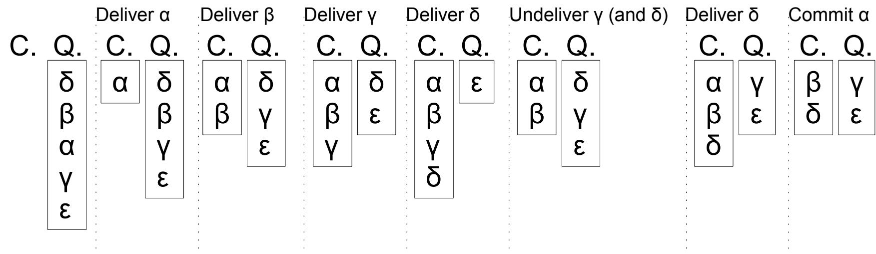
Throughout this section, we will extensively use an example scenario to simplify the descriptions. Given a processor X represented by the system process P and at an initial state S_0, we provisionally deliver messages α, β, γ, and δ in this order. Later, we undeliver message γ (and δ as a consequence). Subsequently, we again deliver δ, thus having α, β, δ provisionally delivered. Finally, we permanently commit the delivery of α (thus leaving β and δ as provisionally delivered). Figure 7.6(a) shows the messages in the queue as well as the messages provisionally delivered after each operation.

When delivering a first message provisionally, the system will fork a sub-process which will handle the message, while the parent process is used to later resume execution after rollback. When a second message is delivered provisionally, there are two options available: handle the second message directly in the child process, or fork another process to handle the second message. These two options are shown in Figure 7.6.

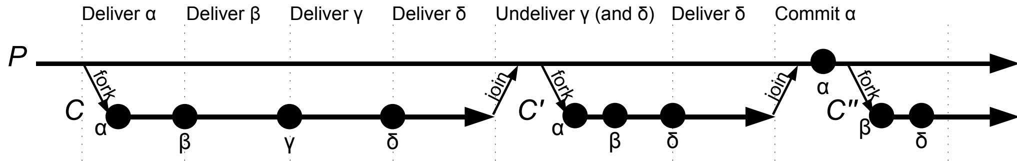
Given the fact that they both provide the same capability, and only their performance may be different, we implemented only one of the two methods, leaving the implementation of the other as future work. In particular, we implemented the method that uses a single child to deliver all the messages provisionally delivered.

7.3.2 Single Forked Process

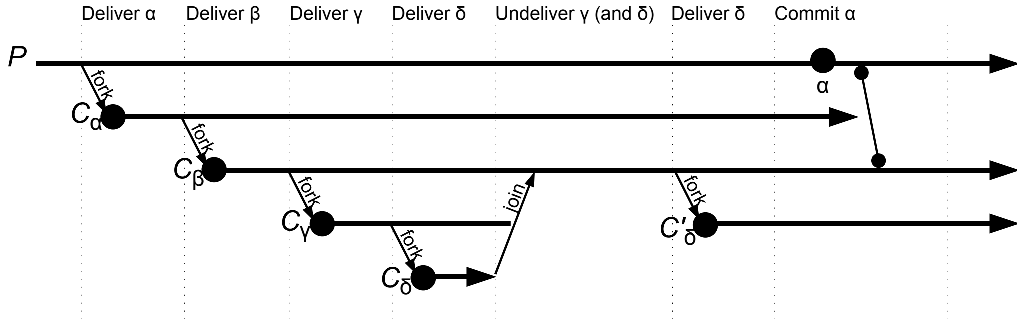
Figure 7.6(b) shows the execution flow of the application for the example given earlier. At the first provisionally delivered message, a child process (C) is forked. From this point on all the CCS requests are handled exclusively by C. When the following three requests for



(a) Message in the queue at every step, distinguishing in messages provisionally delivered "C." and message in the regular queue "Q."



(b) Using a single child process to deliver all messages.



(c) Using a new child process for every new message delivered provisionally.

Figure 7.6: Timeline of the execution on a processor when provisionally delivering messages: several messages delivered provisionally, and a rollback.

provisional delivery arrive at the child process, they are treated as immediate delivery, and process C delivers messages β, γ, and δ to the respective recipient objects.

When the request to undeliver γ arrives, process C terminates execution, indicating that a rollback should be performed. The parent process P is notified of the termination of the child via the closing of the pipe which connects the two processes. At this point, P needs to fork another child process C' to return in provisional mode. The number of messages that the newly forked C' has to deliver (two in this case), and their order, is written by process C on the shared memory segment before terminating execution. This segment is established when the first message α is provisionally delivered. This implies that process P can distinguish between the case "Undeliver All" and "Undeliver Some" (of Figure 7.3) by looking at the number written in the shared memory segment: if the number is greater than zero the system will remain in provisional mode.

The following request to deliver message δ is again interpreted as an immediate delivery

by the child process. Finally, when the request to permanently commit message α arrives, it is again interpreted by process C' which will terminate execution. As before, C' writes the number of messages that have to be re-delivered provisionally on the shared memory segment before terminating. This number is always identical to the number of messages that are provisionally delivered (three in our example). Moreover, this time C' also writes the number of messages that the parent process has to execute before forking process C''. In our example one (only α). Process P will look at these two numbers, deliver one message immediately (α), decrement the number of messages to provisionally deliver accordingly, and finally fork the new child C'' to handle the provisional delivery of the other two messages (β and δ).

Note that the combination of these two numbers covers any possible operation the user may want, from full rollback $\{0, 0\}$, to full commit $\{n, 0\}$ (where n is the number of messages provisionally delivered), plus anything in the middle $\{n - k, k\}$ ($\forall k : 0 < k < n$). Also note that the shared memory segment is only used to store permanent data that both parent and child need to see. It is not used to trigger events in the other process; for this purpose only the bidirectional pipe is used. In other words, none of the processes probes the share memory for value changes.

7.3.3 Multiple Forked Processes

Another alternative to manage multiple provisional delivery of messages is to fork a new child for every message that is provisionally delivered. This is shown in Figure 7.6(c). Each of the four messages α, β, γ, and δ generates a new process that we will denote C_α, C_β, C_γ, and C_δ. Between each couple of parent-child processes there is a communication channel. A single shared memory segment is also shared among all processes. Clearly, the performance of this new scheme when delivering multiple messages is poorer than the previous method since a new fork operation has to take place. Even with the copy-on-write cloning of the virtual address space, this can be a substantial penalty.

Note that when a CCS request arrives from a client, it will be seen and served only by the innermost child process. If the requested operation involves only the local state of the processor, such as the collection of the message queue, no extra communication is needed. Otherwise, if the operation involves more than the innermost child process, for example in the case of a rollback, the innermost process will inform the other processes via the bidirectional pipes.

Any CCS request that arrives on a processor is initially received by the topmost ancestor P since it is responsible for receiving all the messages from the external world. From P, the request needs to be transmitted to the bottommost descendant. If the message has to travel through all the intermediate forked processes, this operation per se would be very expensive. Instead, we envision an additional pipe connecting the topmost ancestor P with the bottommost descendant. Since all the forked processes are the bottommost descendant at some point in time, this pipe needs to be connected to all the children in turn. Luckily, this is simplified by the semantic of forks. Since the pipe established between P and a child process is maintained open across the fork operation, a new child automatically inherits

this direct connection to process P. A potential problem is that all the processes will have simultaneous access to this pipe. Nevertheless, if only the bottommost process is allowed to use it, then only one process will be using it at any time and no contention will arise.

When the request to undeliver γ arrives, process C_γ will terminate execution and write the total number of messages to undeliver on the shared memory segment. This number is equivalent to the number of processes that need to terminate execution. Following the pipe connecting each process to its parent, process C_δ will receive the undeliver command and terminate itself. When the command reaches C_β, this process will not terminate, and instead continue execution normally as the bottommost descendant. In particular, it will reply to the awaiting client debugger. Subsequently, a new process C'_δ will be created when the request to deliver δ reaches process C_β.

It is interesting to note that this approach with multiple processes does not require re-delivery of messages when performing a rollback, but only the destruction of processes. This can lead to a cleaner interface to the user than the previous method of using a single child. This comes from the fact that by re-delivering the same message multiple times during rollbacks might have side effects that could be difficult to hide from the user. For example, if an entry method prints a string, and this entry method is re-executed during rollback, the system will print once again that string, possibly confusing the user.

Finally, when the command to permanently commit α arrives, process C'_γ will inform the original process P that α has to be permanently delivered. In general, the number and order of messages to permanently deliver will be written in the shared memory segment. P will then proceed to deliver the desired messages. At this point, no other operation would be required for the correctness of the method. However, leaving processes like C_α alive can potentially lead to a rapid increase in the number of processes used. These processes can clobber the operating system resources and create problems. Thus, some method of garbage collecting them is necessary. This can be done lazily every time a commit command is issued. After C'_γ writes the number of messages to permanently deliver and sends a message to P, it can also send another message up the pipe connecting it to its parent C_β. This message can then travel up the pipes connecting each process with its parent until it reaches the processes which are not needed anymore (C_α in this case). C_α can then terminate itself while leaving the other processes alive. Note that C_β will also need to modify its parent pipe to point directly to the topmost ancestor instead of C_α, thus completing the bypass of C_α.

7.4 Performance Evaluation

Each of the two methods for delivering multiple messages provisionally has some advantage and some disadvantage. Some of these were already highlighted while explaining the two methods. In this section, we shall focus on the performance of the described method. We gathered experimental data for the single process fork, and we infer some performance information for the other method with multiple processes. The configuration we used is a dual quad-core 2.0 GHz Intel Xeon workstation. Both the client CharmDebug and the parallel Charm++ application were running locally. We measured the time both on the

server side as well as at the client side. This second measurement includes the pre- and post-processing performed by the Java GUI. Table 7.1 shows the performance for various operations performed by delivering and undelivering messages provisionally.

	Server side μs	Client time μs
First provisional message	375 $\pm$ 294	2,061 $\pm$ 90
Following provisional messages	48 $\pm$ 20	1,519 $\pm$ 32
End provisional	240 $\pm$ 65	1,583 $\pm$ 26
Undeliver (+5 redeliver)	681 $\pm$ 161	2,100 $\pm$ 43
Commit 1 (+4 redeliver)	594 $\pm$ 102	2,169 $\pm$ 31

Table 7.1: Performance of single forked process during various provisional delivery operations with relative standard deviations. Measurements in microseconds.

The main consideration is that all the latencies are very small, on the order of a couple of milliseconds perceived by the client. This means that a user issuing a command to deliver a message, or to roll back the application, will be perceived as an instantaneous operation. It is important to notice that this time, and the relatively large difference between the server time and the client time, is mainly due to the sequence of operations performed by the CharmDebug debugger. After executing the desired user command, it reloads the state of the application and the list of messages present in the queue, thus adding two more requests to the server. This is necessary since the delivery of a message might have generated a change in the system that ought to be displayed to the user.

On a more detailed analysis, it can be seen that when delivering messages provisionally, the first one suffers a much bigger overhead than the following ones. This is due to the fork operation necessary to create the child process when entering provisional mode. The subsequent messages are delivered without the need of this operation, thus they are much faster. To exit provisional mode and return to normal mode, the time is slightly lower than in the other direction, but still significantly higher than a single message delivery. To undeliver only some messages, or to commit some messages permanently, the time doubles. This is due to the need to destroy a process and recreate a new one. Note that this time can increase significantly if the number of messages to re-deliver provisionally is large, or if these messages require a long execution time.

An analytical comparison can be made between the two provision delivery methods. The first message is going to take similar time for both systems, while the following messages are going to be much more expensive when using multiple processes. Further, let n be the total number of messages provisionally delivered up to the current time, and let k be the number of messages we are undelivering. By forking one new process for every message provisionally delivered, the rollback speed is independent of n, and depends only on k. On the other hand, by using one single child process, the speed depends on the number of messages that we are not undelivering, in our example $(n - k)$, and how much time these messages take to execute. Clearly, none of the two methods is faster under all conditions, and there will be a crossing point for certain values of n and k. When committing k messages permanently, a similar discussion applies, this time with the single process dependent on n and the multiple

processes dependent on k. An analytical model for the time taken by each operation is presented in Table 7.2. Given the large variances obtained in the experimental setup, we leave this model in symbolic notation.

	Single process	Multiple processes
First provisional message	$c+m$	$c+m$
Following provisional messages	m	$c+m$
End provisional	d	$d \cdot n$
Undeliver k messages	$d+c+m \cdot (n-k)$	$d \cdot k$
Commit k messages	$d+c+m \cdot n$	$(m+d) \cdot k$

Table 7.2: Analytical comparison of the two provisional delivery methods for multiple subsequent deliveries. n is the total number of messages provisionally delivered, k the number of messages being undelivered/committed; c, d, m the time for creation of a process, destruction of a process, and delivery of a message, respectively.

7.5 Case Study

In this section we present a simple case study where the ability to quickly deliver messages and test the outcome of the operation can lead the user to a quick solution to the bug. The example we chose is parallel prefix. This is a standard computation where, given an array with n elements, at the end of the computation the array will be like follows: $a_i = \sum_{k=1}^{i} a_k$. The operational flow in parallel is described in Figure 7.7. At each step i of the algorithm, processor p sends its current value to processor $p + 2^i$.

If a barrier is placed at every step of the algorithm, no problem is present. However, to increase the performance, this barrier can be relaxed, and computation can be allowed to overlap. A naive remove of the barrier will nevertheless result in race conditions (buffering is necessary for a correct implementation). Assume that element p_0 is proceeding fast, and it sends out messages marked m_{01}, m_{02} and m_{04} in rapid succession. Assume further that element p_2 is late and the message m_{24} is delayed. What can happen is that element p_4 receives the message from p_0 before receiving the message from p_2. At this point, p_4 will incorrectly update its local value, and the algorithm will generate a wrong solution. Buffering incoming messages if they arrive too early is a common solution to this problem.

Let us review how a programmer can debug his application using CharmDebug and the provisional message delivery system. After the application has been started, the user can see several messages in the queue to pass the local value to the next element. One of these messages for a later phase is highlighted in Figure 7.8. The user can then decide to provisionally deliver this message. He can then switch to inspect the destination object of that message (element 4 in our case), and notice that its local value has been updated to an incorrect value (i.e not valid according to the parallel prefix algorithm). Alternatively, he can inspect the new messages that appear in the local queue (generated by the provisional

Figure 7.7: Parallel algorithm for prefix computation.

delivery of the message), and notice that again the wrong value is sent out. If one single message delivered provisionally is not enough, the user can deliver more messages. He can also rollback the system, and try a completely different order.

A similar problem occurred in ChaNGa cosmological simulator. This has already been described in the previous chapter. A series of messages were racing, and some ordering among the messages was generating a stall in the application. By using the provisional delivery mechanism, we could easily look at the messages in the queue and try the delivery of some of them. The outcome of the execution would be an additional help to the programmer to understand why and how the messages were racing. This problem was the initial trigger to develop this provisional delivery mechanism.

7.6 Related Work

In addition to providing a mechanism to record and replay an application deterministically, some tools also provide the capability to modify the order in which messages are handled, and thus test different execution paths. In [88], the authors consider the possibility to detect races between messages by grouping them into "waves". The algorithm proposed assumes that the set of messages generated by a program does not change if some messages are delivered in a different order. The paper also evaluates how to find all possible messages that can be delivered at any point in time in a systematic way. More recently, extensions on how to identify possible races between messages by efficiently scanning the search space have been presented for distributed systems [89] and for MPI applications [90]. An interesting extension to the generation of possible orderings of message delivery, and how to explore the

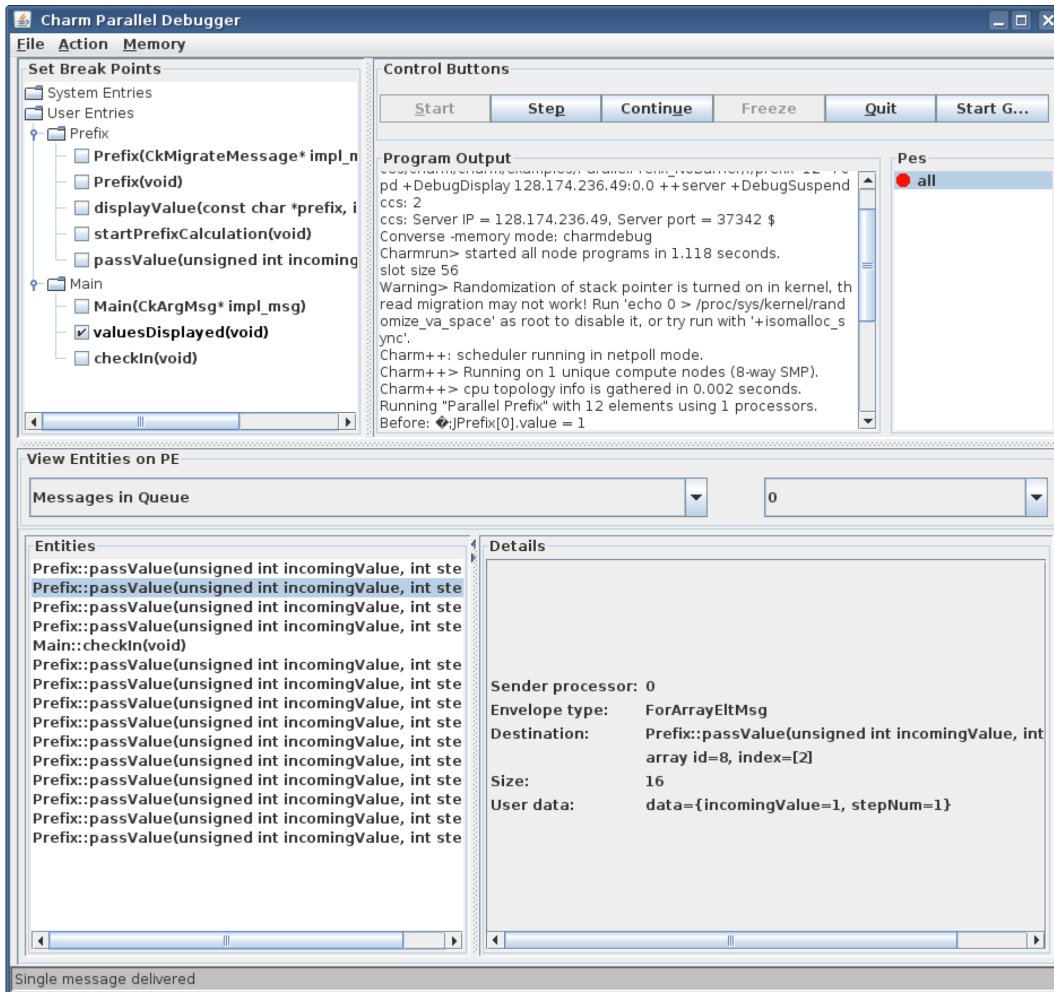


Figure 7.8: Screenshot of CharmDebug while debugging the parallel prefix application. Multiple messages are enqueued from different steps.

generated space without maintaining all possible executions active, is presented in [91].

A tool to allow the user to select messages to be delivered in different order is MAD [92]. In this case, the tool allows any message to be exchanged when a wildcard receive is issued by the MPI program. The system, upon the user decision, will re-execute the entire application with the modified message ordering.

A similar implementation, using the fork system call, is available through the GDB debugger [23] for sequential programs. When debugging his sequential program, a user can issue the `checkpoint` command, and have the program store a copy of itself in a cloned process. This procedure could be applied to one of the processes that is part of a parallel application. However, the effects of the execution of the cloned process will be immediately visible on the other processors composing the parallel application. This would prevent rollback, and render the operation useless.

7.7 Future Work

The procedure described in this chapter allows a programmer to take an application, and decide which messages are to be delivered, and in which order, overwriting the default CHARM++ queue. This mechanism could also be used by automatic tools that would perform an unsupervised search of all (or certain) orderings of messages. These tools would then notify the user when a discrepancy is detected on the final states generated by two different message orderings. For this automatic mechanism to work, we can identify two challenges that need to be overcome.

The first challenge relates to identifying whether two final states, generated by two different message orderings, are identical or not. Note that simply comparing the state of the memory is not enough. For example, a linked list could contain elements in different order, but if this does not affect the algorithm, then the two states should be considered equivalent. Floating point values may also be bit-wise different, but still represent correct final states. One possible solution to this problem is to have the programmer insert a specification if two states of the same CHARM++ object are equivalent. This could be in the form of an explicit equal operator, or with the use of specialized Pack/Unpack routines.

The second challenge to consider is which message orderings are valid. Let us first define a buggy application as an application where, for a particular input, the output result can be incorrect. From this definition comes the corollary that an application does not contain a bug if, for any possible valid input, the output produced is always correct. This translates this second challenge as preventing the system from giving false positives.

At first glance one may say that any permutation of messages present in the queue is a valid ordering. This can be true for programming models like MPI, where upon a wildcard receive, all messages that can match that receive are to be assumed equally valid. However, in CHARM++ this is not true. Consider, for example, the situation where an object sends itself (or to a local group) two messages, A and B. If A is sent before B, then in CHARM++ message A will always be processed before message B, given how the local queue behaves.[1] If the messages had priorities, these would need to be considered as well. In the same example, if A has a lower priority than B, and the two messages are sent from within the same entry method, then in CHARM++ message B will always be delivered first. Therefore, given the definition above, an ordering that exposes a bug when the messages are processed in an order which never happens in a normal execution is a false positive. Finding exactly which orderings are valid and which are not is a challenging task, especially when considering the transitivity property in message orderings.

---

[1] The processing order of local message in a CHARM++ application is valid at the time this paper is written. Future releases of CHARM++ may alter the scheduler's behavior, and programmers should refer to the CHARM++ manual for assumptions that can be made about the scheduler.

8 Conclusions

In this thesis, we have addressed the issue of debugging parallel applications, in particular when a large number of processors is required to reproduce a problem. We presented a novel approach that leverages a tight integration with the runtime system underlying the parallel application to allow the debugger to scale to as many processors as the parallel runtime system does. In this scenario, we have illustrated how unsupervised execution can be applied to relieve the user from controlling the entire set of processors allocated, and concentrate only on those of interest.

While the application is running, both the runtime system and the user can specify conditions that should raise a notification. For the runtime system, this includes both signals sent to the applications, and more sophisticated memory-related checks. In particular, we developed a memory infrastructure where different objects co-existing inside a common address space and sharing the memory can be protected from each other. Three protection mechanism to detect when a cross-object corruption happens were presented, and their strengths and weaknesses were studied.

The user can also insert conditions that will trigger the notification system. The traditional method of compiling assertions inside the code is supported in a scalable manner. In addition, a new method is proposed to dynamically insert correctness checks into the running parallel application without restarting it. This is done through a generic interface that allows Python scripts to be inserted into a running application, and an inspection framework to deal with the lack of reflectiveness in the C/C++ language. This interface is also a contribution of this thesis, and it is generic enough to allow uses other than debugging; some examples are computational steering and data analysis.

Another contribution of this thesis is the consideration of the challenge of allocating a large number of processors for long debugging sessions. Programmers already encounter many difficulties to debug their applications at scale, and the issues are likely to increase as machines grow larger. In spite of that, little or no work has been done prior to this thesis. To alleviate this problem and facilitate programmers to debug their application at scale, in this thesis we presented three approaches. The first solution exploits object-level virtualization to emulate a large machine using a smaller one, and enables the user to debug his application as if it were running on the real large machine. The second solution is called "processor extraction", and it combines two record-replay techniques into a three-step procedure. The first being a non-intrusive light-weight solution to record only the application non-determinism, the second being a more comprehensive record scheme that allows a processor to be replayed in disjunction with the rest of the parallel application. This procedure allows both the reduction of the disruption in the manifestation of the application's problem, and the reduction

of the storage needed to re-execute processors in isolation. The last solution we proposed to reduce the need for large machines is to allow the user to test what effects are generated when messages are delivered in a particular order. To enable a fast response time from the system, we developed a mechanism to deliver a message “provisionally”, and rollback the application without restarting it. The effects of the provisional delivery are erased upon rollback.

Finally, throughout this thesis we have used the CHARM++ runtime system as the implementation platform of our techniques and ideas. As a consequence, one contribution of this thesis is the environment available to CHARM++ developers to help them debug their applications. While this thesis focused primarily on large scale problems, the existing debugging environment has been greatly improved also for parallel applications using only few processors. All the implementation described are available with source code as part of the CHARM++ and CHARMDEBUG systems.

References

[1] Fred P. Brooks, Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, page 193, New York, NY, USA, 1975. ACM.

[2] TotalView Technologies. TotalView® debugger. http://www.totalviewtech.com/TotalView.

[3] Allinea. The distributed debugging tool (DDT). http://www.allinea.com/index.php?page=48.

[4] The Eclipse Foundation. Eclipse - an open development platform. http://www.eclipse.org/.

[5] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proceedings of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.

[6] Steve Otto Marc Snir and etc. *MPI: The Complete Reference*, volume 1. The MIT Press, 1998.

[7] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference, October 6–8, 1993, Mississippi State, Mississippi*, pages 160–165, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[8] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[10] 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.

[11] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[12] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[13] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for Charm++. In *PADTAD Workshop for IPDPS 2004*, page 294. IEEE Press, 2004.

[14] Filippo Gioachin, Chee Wai Lee, and Laxmikant V. Kalé. Scalable Interaction with Parallel Applications. In *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[15] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance Evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[16] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[17] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[18] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[19] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[20] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[21] Sameer Kumar. *Optimizing Communication for Massively Parallel Processing*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[22] Department of Computer Science,University of Illinois at Urbana-Champaign, Urbana, IL. *The CONVERSE programming language manual*, 2006.

[23] Free Software Foundation. GDB: The GNU Project Debugger. http://www.gnu.org/software/gdb/.

[24] Gengbin Zheng, Arun Kumar Singla, Joshua Mostkoff Unger, and Laxmikant V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, Fort Lauderdale, FL, April 2002.

[25] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

[26] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Baltimore, MD, September 2002.

[27] National Center for Supercomputing Applications. Blue Waters project. http://www.ncsa.illinois.edu/BlueWaters/.

[28] Chao Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2007. http://charm.cs.uiuc.edu/papers/ChaoMeiMSThesis07.shtml.

[29] Mani Potnuru. Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

[30] L. V. Kale, Milind Bhandarkar, Robert Brunner, 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.

[31] Filippo Gioachin and Laxmikant V. Kalé. Dynamic High-Level Scripting in Parallel Applications. In *In Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Rome, Italy, May 2009.

[32] Allinea Software Jacques Philouze. Debugging the future - GPU's and petascale. HPC Advisory Council European Workshop, May 2010. (presentation).

[33] Gregory L. Lee, Dong H. Ahn, Dorian C. Arnold, Bronis R. de Supinski, Matthew Legendre, Barton P. Miller, Martin Schulz, and Ben Liblit. Lessons learned at 208k: towards debugging millions of cores. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–9, Piscataway, NJ, USA, 2008. IEEE Press.

[34] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *SC2003*, 2003.

[35] Cray Inc. *Programming Environments Release Announcement*, S-9401-0909 edition, September 2009. http://docs.cray.com/books/S-9401-0909/.

[36] Oscar Nierstrasz, Re Bergel, Marcus Denker, Stphane Ducasse, Markus Glli, and Roel Wuyts. On the revival of dynamic languages. In *Proceedings of Software Composition 2005. LNCS*, pages 1–13, 2005.

[37] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31:23–30, 1998.

[38] Python Software Foundation. Python/C API Reference Manual, 2008. http://docs.python.org/api/api.html.

[39] Gengbin Zheng, Orion Sky Lawlor, and Laxmikant V. Kalé. Multiple flows of control in migratable parallel programs. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 435–444, Columbus, Ohio, August 2006. IEEE Computer Society.

[40] Gregory R. Watson and Craig E. Rasmussen. A strategy for addressing the needs of advanced scientific computing using eclipse as a parallel tools platform. Technical Report LA-UR-05-9114, Los Alamos National Laboratory, December 2005.

[41] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[42] C.C. Williams and J.K. Hollingsworth. Interactive binary instrumentation. *IEE Seminar Digests*, 2004(915):25–28, 2004.

[43] Paolo Falcarin and Gustavo Alonso. Software architecture evolution through dynamic aop. In *AOP , European Workshop on Software Architecture (EWSA 2004)*, pages 57–73. Springer-Verlag, 2004.

[44] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[45] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. *Lecture Notes in Computer Science*, 707:482–??, 1993.

[46] Tyng-Ruey Chuang, Y. S. Kuo, and Chien-Min Wang. Non-intrusive object introspection in C++. *Software– Practice and Experience*, 32(2):191–207, 2002.

[47] J. Hamilton, R. Klarer, M. Mendell, and B. Thomson. Using SOM with C++. C++ report, August 1995.

[48] Hermanpreet Singh. Introspective c++. Master's thesis, Computer Science Department, Virginia Polytechnic Institute and State University, 2004.

[49] Kurt Stephens. Xvf: C++ introspection by extensible visitation. *SIGPLAN Not.*, 38(8):55–59, 2003.

[50] Filippo Gioachin, Gengbin Zheng, and Laxmikant V. Kalé. Debugging Large Scale Applications in a Virtualized Environment. In *Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, number 10-11, Huston, TX (USA), October 2010.

[51] Mark F. Mergen, Volkmar Uhlig, Orran Krieger, and Jimi Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev.*, 40(2):8–11, 2006.

[52] Yi Pan, Norihiro Abe, Kazuaki Tanaka, and Hirokazu Taki. The virtual debugging system for developing embedded software using virtual machinery. *Embedded and Ubiquitous Computing*, 3207:139–147, 2004.

[53] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. Diecast: Testing distributed systems with an accurate scale model. In *In Proceedings of the 5th USENIX Symposium on Networked System Design and Implementation (NSDI08)*. USENIX Association, 2008.

[54] Paula Ta-Shma, Guy Laden, Muli Ben-Yehuda, and Michael Factor. Virtual machine time travel using continuous data protection and checkpointing. *SIGOPS Oper. Syst. Rev.*, 42(1):127–134, 2008.

[55] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[56] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[57] The research hypervisor: A multi-platform, multi-purpose research hypervisor. http://www.research.ibm.com/hypervisor/.

[58] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kale, and Paul M. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*, number 10-14, Ischia/Naples/Italy, August 2010.

[59] Xiangmin Jiao, Gengbin Zheng, Phillip A. Alexander, Michael T. Campbell, Orion S. Lawlor, John Norris, Andreas Haselbacher, and Michael T. Heath. A system integration framework for coupled multiphysics simulations. *Engineering with Computers*, 22(3):293–309, 2006.

[60] Filippo Gioachin and Laxmikant V. Kalé. Memory Tagging in Charm++. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD)*, Seattle, Washington, USA, July 2008.

[61] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228 –235, jan. 1961.

[62] Ieee standard for information technology- portable operating system interface (posix). *IEEE Std 1003.1-2001/Cor 2-2004*, pages 0_1 –91, 2004.

[63] Intel Corporation. Intel Thread Checker. http://www.intel.com.

[64] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 34–41, New York, NY, USA, 2006. ACM.

[65] Michiel Ronsse and Koenraad De Bosschere. Non-intrusive on-the-fly data race detection using execution replay. In *Automated and Algorithmic Debugging*, 2000.

[66] Julian Seward and Nicholas Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2005. USENIX Association.

[67] TotalView Technologies. *Debugging Memory Problems Using TotalView Debugger*. http://www.totalviewtech.com.

[68] Filippo Gioachin, Gengbin Zheng, and Laxmikant V. Kalé. Robust Record-Replay with Processor Extraction. In *PADTAD '10: Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pages 9–19. ACM, July 2010.

[69] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Not.*, 24(1):124–129, 1989.

[70] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Trans. Comput. Syst.*, 17(2):133–152, 1999.

[71] Carl Dionne, Marc Feeley, and Jocelyn Desbiens. A Taxonomy of Distributed Debuggers Based on Execution Replay. In *In Proceedings of the 1996 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 203–214, 1996.

[72] Michiel Ronsse, Koenraad De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Automated and Algorithmic Debugging*, 2000.

[73] Larry D. Wittie. Debugging distributed C programs by real time replay. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):57–67, January 1989.

[74] Chris Gottbrath. Quickly Identifying the Cause of Software Bugs with ReplayEngine. Technical report, TotalView Technologies, August 2008.

[75] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, Minneapolis, MN, November 1992.

[76] Ruini Xue, Xuezheng Liu, Ming Wu, Zhenyu Guo, Wenguang Chen, Weimin Zheng, Zheng Zhang, and Geoffrey M. Voelker. MPIWiz: Subgroup reproducible replay of MPI applications. In *In PPoPP*, 2009.

[77] Franco Zambonelli and Robert H.B. Netzer. An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications. In *In Proceedings of the 13th International and 10th Symposium on Parallel and Distributed Processing*, pages 392–398. IEEE, 1999.

[78] Robert H. B. Netzer, Sairam Subramanian, and Jian Xu. Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs. Technical report, Brown University, Providence, RI, USA, 1994.

[79] Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM.

[80] Chris Gottbrath. Reverse Debugging with the TotalView Debugger. In *CDROM*. Cray User Group Conference 2008, May 2009.

[81] Polle Trescott Zellweger. *Interactive source-level debugging for optimized programs (compilation, high-level)*. PhD thesis, University of California, Berkeley, 1984.

[82] Max Copperman and Charles E. McDowell. Debugging optimized code without surprises. In *Proceedings of Supercomputer Debugging Workshop '91*, pages 1–16, November 1991.

[83] Caroline Mae Tice. *Non-Transparent Debugging of Optimized Code*. PhD thesis, EECS Department, University of California, Berkeley, Nov 1999.

[84] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing*, pages 93–103, San Dieago, CA, September 2004.

[85] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, 1985.

[86] David M. Kunzman and Laxmikant V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[87] Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[88] Richard Kilgore and Craig Chase. Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages. In *In Proceedings of the International Conference on System Sciences*, page 423, 1997.

[89] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.

[90] Sarvani Vakkalanka, Michael DeLisi, Ganesh Gopalakrishnan, Robert Kirby, Rajeev Thakur, and William Gropp. Implementing efficient dynamic formal verification methods for mpi programs. In Alexey Lastovetsky, Tahar Kechadi, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 5205 of *Lecture Notes in Computer Science*, pages 248–256. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87475-1-34.

[91] Basile Schaeli and Roger D. Hersch. Dynamic testing of flow graph based parallel applications. In *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*, pages 1–10, New York, NY, USA, 2008. ACM.

[92] Dieter Kranzlmüller, Christian Schaubschläger, and Jens Volkert. A Brief Overview of the MAD Debugging Activities. In *AADEBUG*, 2000.

Appendix A: Debugging API

In this appendix, we review the operations that are available to the user through the CharmDebug interface, and how these can be extended by additional modules. All these queries are performed via the generic Converse Client Server (CCS) protocol.

A.1 Queries to a Charm++ Application

The queries that a debugger (like CharmDebug) can issue to a running Charm++ application are listed below. Each query is represented by an alpha-numeric string that will be matched by the runtime system to determine the operation to be performed. The list can be divided into two categories.

Queries in the first category have direct access to the communication layer, and are implemented at the lowest level. If they can be performed in parallel, an appropriate reduction mechanism must be registered at startup. Moreover, the output returned to the user does not have a predefined format, and is specific to each query. Queries in this category are mostly commands to the application, and they generally return a simple confirmation as a reply. They can also return complex data structures if necessary. The naming convention consists of the word "debug" at the beginning, followed by the name of the module in charge of the operation, followed by the operation performed.

"debug/converse/freeze" Suspend the execution on the given processors. Messages in the queue will not be processed further.

"debug/converse/status" Gather the status of the process (i.e suspended, running, etc).

"debug/converse/arch" Gather information about the architecture on which the application is running.

"debug/charm/bp/set" Set a breakpoint on the entry method specified as a parameter on the given processor set.

"debug/charm/bp/remove" Remove the breakpoint on the entry method specified as a parameter on the given processor set.

"debug/charm/bp/removeall" Remove all breakpoints from the requested processor set.

"debug/charm/continue" Issue a continue command. The scheduler returns to process messages normally.

“debug/charm/next” Deliver a single message in the queue on the given processors. The message processed is the next in the queue.

“debug/converse/startgdb” Start a sequential debugger attached to the given processors.

“debug/converse/quit” Terminate the parallel program.

“debug/charm/deliver” Deliver a specific message in the queue, overwriting the natural order. The message index is passed as input to the query.

“debug/provisional/deliver” As “debug/charm/deliver”, only that the message is delivered provisionally, with possibility of rollback.

“debug/provisional/rollback” Undeliver a certain number of messages that had been delivered provisionally. Possibly return to normal mode if all the provisional messages are undelivered.

“debug/provisional/commit” Commit permanently a certain number of messages that had been delivered provisionally. Possibly return to normal mode if all the provisional messages are committed.

Queries in the second category have a specific output format, and they return lists of objects. These use Pack/UnPack (PUP) routines typical of Charm++ applications to construct the reply. Internally, the system PUP::er class encodes the reply. A facility class is available on the client debugger to help decode the data. The naming convention is similar to the other category, the only difference being the lack of the “debug” prefix.

“converse/lists” All the retrievable items (returns this list itself).

“charm/chares” All the chare types available in the application.

“charm/entries” All the entry methods for all the chare types.

“charm/messages” All the message types.

“charm/mains” The mainchares instantiated.

“charm/objectNames” The chare groups/nodegroups instantiated.

“charm/arrayElements” The array elements mapped on a specific processor.

“converse/localqueue” The messages enqueued on a specific processor.

“charm/readonly” The readonly variables declared by the application.

“charm/readonlyMsg” The readonly messages declared by the application.

“charm/messageStack” The list of messages that are being delivered (for nested entry methods).

A.2 Extending Query Set

To implement a debugging functionality not present in the CHARM++ core, a user can link an external module into the application, and register in this module extensions to the debugging capabilities. Both sets of queries (pre-formatted and non) can be extended. The `charmdebug` memory module (available by default as one of the possible memory allocators) extends this interface, and it provides the following queries.

Queries without a pre-formatted reply. To extend this set, the developer has to provide a new "CCS" handler for the newly provided function, and register it to the CCS framework. If the operation can be performed on multiple processors simultaneously, a reduction function must be registered with the operation. The memory charmdebug module provides:

"debug/memory/stat" Gather statistics about the allocated memory.

"debug/memory/allocationTree" Gather a tree-based view of the allocated memory (based on the stack trace at the allocation instant).

"debug/memory/leak" Perform a search for memory leaks (a parameter specifies which method should be used).

"debug/memory/mark" Mark/unmark all the currently allocated memory as non-leak.

Pre-formatted queries. To extend this set, the developer has to register the new tag with the CpdList framework, and either extend the C++ class `CpdListAccessor` or its equivalent C extension. The memory charmdebug module provides:

"memory/list" A list of all the memory allocated on a given processor.

"memory/data" The content of a given allocated memory block as a byte buffer.

A.3 Notification Events

When something of interest happens on a particular processor, the parallel runtime system can notify the attached debugger of the event occurred, and possibly suspend execution. In the CHARM++/CHARMDEBUG system, this is implemented via a function named `CpdNotify`. To send a notification, the application calls this function with a notification code (listed below) and its related parameters as arguments. On the debugger side, the notification is received via a callback mechanism. Following the standard semantic used in Java programming, components (both CHARMDEBUG GUI or any add-on) can register itself as a NotifyListener. Upon receipt of a notification, the corresponding method on each registered NotifyListener is invoked. The list of notifications currently available follows.

SIGNAL A signal was sent to a processor.

ABORT The application failed an assertion or has explicitly called abort().

FREEZE A processor has suspended message processing.

BREAKPOINT A processor hit a breakpoint.

CORRUPTION The memory has been corrupted.

Appendix B: Python/Charm++ Interaction API

In this appendix, we enumerate the functions that are provided to Python scripts inserted into a running application.

Inside the Python interpreter, before the user code is executed, the "ck" module is pre-loaded. The functions currently available in the "ck" module are:

printclient Print a string on the client (pulled via PythonPrint queries).

printstr Print a string on the server.

mype Return the index of the processor where the script is executing.

numpes Return the total number of processors the application is using.

myindex Return the index of the chare array element connected with the script. This value has the same dimensionality as the chare array connected. "none" is returned if the attached chare is not part of an array.

read Return the content of a specific variable. This function calls the "read" method of the associated chare type. The arguments passed into this function must match those expected by the read method of the chare type. The returned type can be any complex data structure, and its content is specified by the chare type's read method.

write Overwrite a specified variable with the provided value. This function calls the "write" method of the associated chare type. The arguments passed into this function must match those expected by the write method of the chare type.

When writing Python code through the CharmDebug interface. The "charm" module is also pre-loaded with some predefined methods available. These methods help to inspect the application data structures, and they are:

getMessage Return a handle to the message currently being delivered.

getStatic Return the content of a specified readonly variable.

getArray Return a new handle for the specified element of the array data structure pointed by the specified handle.

getValue Return the content of a named variable inside the specified object handle.

getCast Reinterpret the handle as a different type. This is important for C++ classes with multiple inheritance were the handle can change when casting a type to another type.

Curriculum Vitæ

Office Address:
4219, Siebel Center For Comp. Sc.
201, N. Goodwin Ave.
Urbana, IL 61801-2302

Contact Information:
E-mail: gioachin@ieee.org
Phone: +1-217-721-2978
Work: +1-217-333-4764

Filippo Gioachin

Education

- **Doctor of Philosophy in Computer Science**, *September 2010*
  University of Illinois at Urbana-Champaign, IL, USA. GPA: 3.8/4.0
  Thesis title: "Debugging large scale applications with virtualization".
  Advisor: Professor Laxmikant Kalé.

- **Laurea in Computer Science and Engineering** (5 year program), *July 2002*
  University of Padova, Italy. Overall score: 110/110 *cum laude*
  Thesis title: "Parallelization of a tree-sph code".
  Advisor: Professor Gianfranco Bilardi.

Research Experience

- **Research Assistant**, *August 2003 - present*
  Parallel Programming Laboratory, Dept. of Computer Science, University of Illinois at Urbana-Champaign, USA. Worked on the development of:

  Charm++ parallel runtime system—Worked on most aspect of the system, including new communication substrates, scheduling, load balancing, and communication optimization;

  Parallel debugging—Developed new techniques for debugging large scale applications. Of particular interest, techniques to handle non-deterministic and memory-related bugs;

  Application development—Worked on ChaNGa, a cosmological simulator. Optimized the code to scale to thousands of processors. Worked on its porting to new architectures such as Cell/BE and GPGPU.

- **Internship**, Debugging group, Cray Inc., USA. *June 2007 - August 2007*
  Enhanced the support for debugging tools in the Cray/XT environment; active participation in discussions regarding next generation debugging tools.

- **Research Assistant**, *September 2002 - July 2003*
  Center of Excellence "Science and Applications of Advanced Computing Paradigms", University of Padova, Italy. Worked on a parallel cosmological simulator. Parallelized and optimized the existing algorithm, as well as explored new algorithms yielding greater accuracy.

Teaching and Mentoring Experience

- **Mentor**, University of Illinois at Urbana-Champaign, USA. *Fall 2009*
  Mentored undergraduate students in semester-long projects.

- **Teacher**, *October 2008*
  "3rd Specialized Parallel Programming School", Cineca, Italy.
  Taught a full-day tutorial. For the occasion, I adapted previously existing material for an audience of young researchers approaching the field of parallel computing.

- **Research Assistant**, *September 2002 - July 2003*
  Department of Computer Science and Engineering, University of Padova, Italy.
  Helped with guest lectures and exams grading.

Publications

- **Filippo Gioachin**, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kalé, Thomas R. Quinn: Scalable Cosmological Simulations on Parallel Machines, in *Proceedings of the 7th International Meeting on High Performance Computing for Computational Science (VECPAR 2006). LNCS 4395, pp 476-489, 2007*

- **Filippo Gioachin**, Gengbin Zheng, Laxmikant V. Kalé: Robust Non-Intrusive Record-Replay with Processor Extraction, in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII)*, 2010

- **Filippo Gioachin**, Gengbin Zheng, Laxmikant V. Kalé: Debugging Large Scale Applications in a Virtualized Environment, *in Proceedings of the 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*

- Chao Mei, Gengbin Zheng, **Filippo Gioachin**, Laxmikant V. Kalé: Optimizing a Parallel Runtime System for Multicore Clusters: a Case Study, *in Proceedings of the 5th annual TeraGrid Conference (TG'10)*, 2010

- Pritish Jetley, Lukasz Wesolowski, **Filippo Gioachin**, Laxmikant V. Kalé, Thomas R. Quinn: Scaling Hierarchical N-body Simulations on GPU Clusters, *to appear in the Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC10)*, 2010
- **Filippo Gioachin**, Laxmikant V. Kalé: Dynamic High-Level Scripting in Parallel Applications, in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09)*
- **Filippo Gioachin**, Chee Wai Lee, Laxmikant V. Kalé: Scalable Interaction with Parallel Applications, in *Proceedings of TeraGrid '09*
- **Filippo Gioachin**, Laxmikant V. Kalé: Memory Tagging in Charm++, in *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*
- Pritish Jetley, **Filippo Gioachin**, Celso Mendes, Laxmikant V. Kalé, Thomas R. Quinn: Massively Parallel Cosmological Simulations with ChaNGa, in *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS '08)*
- **Filippo Gioachin**, Ravinder Shankesi, Michael J. May, Carl A. Gunter, Wook Shin: Emergency Alerts as RSS Feeds with Interdomain Authorization, in *IARIA International Conference on Internet Monitoring and Protection (ICIMP '07), 2007*
- **Filippo Gioachin**, Laxmikant V. Kalé: Debugging Parallel Applications via Provisional Execution, in *PPL Technical Report, October 2010*
- **Filippo Gioachin**, Pritish Jetley, Celso L. Mendes, Laxmikant V. Kalé, Thomas R. Quinn: Toward Petascale Comological Simulations with ChaNGa, in *PPL Technical Report, April 2007*

Poster Presentations

- **Filippo Gioachin**, Sayantan Chakravorty, Celso Mendes, Laxmikant Kalé, Thomas Quinn: Cosmological Simulations on Supercomputers, at *International Conference for High Performance Computing, Networking, Storage and Analysis (SC06)*, Tampa, FL, USA
- Thomas Quinn, Laxmikant Kalé, **Filippo Gioachin**, Orion Lawlor, Graeme Lufkin, Gregory Stinson: Salsa: a parallel, interactive, particle-based analysis tool, at *International Conference for High Performance Computing, Networking and Storage (SC2004)*, Pittsburgh, PA, USA

Awards and Achievements

- **Best Paper Award** for paper "Memory Tagging in Charm++" at the *6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD '08)*
- Selected for the **Doctoral Showcase** presentation at the *International Conference for High Performance Computing, Networking, Storage and Analysis (SC08)*, Austin, TX, USA
- **Graduation Honour (cum laude)** at University of Padova, Italy, 2002
- **Conference Travel Award** for IPDPS conference, Rome, Italy, 2009

Professional Activities and Affiliations

- *Reviewer* for the 16th annual IEEE International Conference on High Performance Computing (HiPC 2009)
- *Student Volunteer* for four years at Supercomputing conference
- *IEEE and IEEE Computer Society* member since 2000