Debugging in the Context of Charm++

Rashmi Jyothi, Orion Sky Lawlor, L. V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
jyothi@uiuc.edu, olawlor@acm.org, kale@cs.uiuc.edu

Abstract

This paper mostly describes a parallel debugger and the related debugging support implemented for CHARM++, a data-driven parallel programming language. Though the paper specifically talks about the debugging techniques for CHARM++, it takes a generic stand by bringing into focus the various challenges often faced during debugging of parallel applications systems.

1 Introduction

Parallel programming introduces many additional challenges with respect to program correctness, robustness and reliability. The challenges faced during the design of efficient debugging tools for a parallel program include portability of such tools and examining the dynamic state information of a parallel program which is much more than a sequential program. [1, 2, 3]

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

Parallel debugging techniques aim at overcoming the shortcomings of employing sequential debugging schemes in conjunction with a parallel program. The primary goal of a parallel debugger is to provide an integrated debugging environment which allows the programmer to examine the changing state of the parallel program during the course of its execution.

2 Charm++ and Debugging Requirements

CHARM++[4, 5] is an object oriented parallel programming language based on C++ and is essentially a thin wrapper on Converse [6] which is a framework for parallel programming that supports multi-lingual interoperability. CHARM++ programs are portable and run without change on all MIMD machines and thereby satisfy the key requirement for large scale development of parallel software, portability. The runtime system also supports dynamic load balancing strategies which means that dynamic creation of parallel work is allowed. Dynamic load balancing is necessary when there are irregular parallel computations and load is unevenly distributed among the processor elements.

The execution model of CHARM++ is message-driven [7] wherein Converse treats the parallel machine as a collection of nodes that communicate primarily via messages. Each node is comprised of a number of processors that share memory. When a message arrives at a processor it triggers the execution of a handler function[8].

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

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

The parallel programming model of CHARM++ is based on the concept of virtualization [9], where the programmer divides the work into a large number of chunks, and lets the runtime system map these entities to processors. The system also has the capability to change the mapping at runtime without the user program having to specify it. The number of parts a computation is broken into is typically independent of the number of processors *N* and is more often than not larger than *N*. The CHARM++ programmer does not refer to processors in their code but programs in terms of the interaction between the virtual entities. These virtual entities are called *chares*. In other words, the basic unit of parallel computation in CHARM++ programs is the chare, which can be created on any available processor and the methods of which can be invoked from remote processors.

Chares can create new chares and can send messages to each other. In accord with the message-driven execution model of CHARM++ all computations are initiated in response to messages being received. The system calls in CHARM++ are non-blocking and therefore, asynchronous [10]. CHARM++ entities can contain private data and public methods like regular C++ objects. The significant difference is that these methods can be invoked from remote processors asynchronously. Asynchronous method invocation implies that the caller does not wait for the method to be executed or in other words does not wait for the method to return a value. Such a method that can be invoked remotely in CHARM++ is called an *entry method* or an *entry point*. An entry method does not have a return value and is always a part of a chare.

The other important programming entity available to a CHARM++ programmer is the *Chare Array* [11]. A *Chare Array* is a collection of chares and the size of the array is not constrained by the underlying parallel machine such as number of processors or nodes. Therefore, a chare array can have any number of elements. Each array element of a chare array has a globally unique index and messages are addressed to that index. The dynamic load balancing framework which kicks off when a CHARM++ program starts, treats array elements as objects that can be migrated across processors [11].

The state of a sequential program includes contents of all its variables and registers whereas a parallel program in addition contains data that is shared and in transit between processors. There is a need for a way to examine this data in whatever form the parallel programming model presents it. Keeping in mind the language entities and the programming model of CHARM++, a programmer who wishes to debug their CHARM++ program or verify it for its correctness would require means of inspecting the contents of the parallel entities like the array elements during program execution. The other informative entities are the messages in the scheduler queues [12] on each of the processors which represent the running state of the program. The ability to examine the messages in the queues and their contents would be very handy in the process of debugging and in providing a clear picture of the running state of the parallel program to the CHARM++ programmer. The debugging framework would have to provide the programmer ability to freeze the program execution at will, to examine the entities. Also, the programmer would need to set breakpoints in terms of entry points as entry points capture the true essence of control-flow in a CHARM++ program. Also necessary is the ability to debug in a parallel environment through a collection of instances of sequential debuggers, each attached to a constituent process in the parallel program.

Nondeterminism of a parallel program adds to the complexity of the debugging process. Certain bugs manifest themselves only due to a specific ordering of messages and may not show up in a re-run of the program due to a different ordering of messages in the re-run [13, 14]. Hence, provision to reproduce such bugs deterministically by recording the states of the parallel program and replaying the execution using the recorded data, is necessary for a parallel programming model like CHARM++.

Taking into account the debugging requirements identified, the debugging support implemented for CHARM++ can be summarized as follows.

- Ability to use traditional sequential debugging methods in the context of a CHARM++ program.
- Making it possible for the programmer to freeze or

unfreeze the execution of the program.

- Provision to set and remove breakpoints at the entry point [10] level in a CHARM++ program.
- Providing means to view the contents of entities like array elements and messages in queues across the parallel set-up during program execution.
- A flexible way of attaching specified processes of the parallel program to the sequential debugger, during the course of program execution. This is specially useful when the number of processor elements is high and sequential debugging is sought on specific processors.
- A record-replay mechanism for reproducing bugs which happen once in a while depending on the order in which messages are processed.

3 Debugging Solutions

CHARM++ provides the user ways to debug a parallel program using the traditional sequential debugging techniques. The means of using a sequential debugger for debugging a parallel program is resorted to in CHARM++ [4, 5, 7] in the form of the command-line run-time option "*++debug*", where each individual process of the parallel program running on a processor element is attached to an instance of a sequential debugger like *gdb* or *dbx* and opened in separate terminal windows at the beginning of program execution. The approach of logging is also used for debugging by the way of *CkPrintf* statements. *CkPrintf* is the parallelized version of the *printf* function. Additionally, if the command-line parameter "*+syncprint*" is passed to a CHARM++ program, the *CkPrintf* actually blocks until the output is queued, allowing the logging to happen in causal order, at the cost of dramatically slowing down the output.

A parallel debugger for CHARM++ is implemented which transcends the sequential techniques by allowing the programmer to perform various actions like setting/removing break points in terms of entry points, freezing/unfreezing the execution, examining entities in the parallel program and attaching specific processes of the parallel program to instances of a sequential debugger. The prototype for the user interface of the parallel debugger is implemented as a *Java* client. The successful functioning of the parallel debugger is entirely dependent on the debugging support incorporated into the CHARM++ runtime system or the Charm kernel.

The CHARM++ programmer starts the debugger client from the command-line specifying the program to be debugged, its parameters and the number of processor elements it should run on as command-line parameters. Alternatively, the program and the parameters could be set via a menu item provided by the debugger GUI. The menu usage is shown in Figure 1.

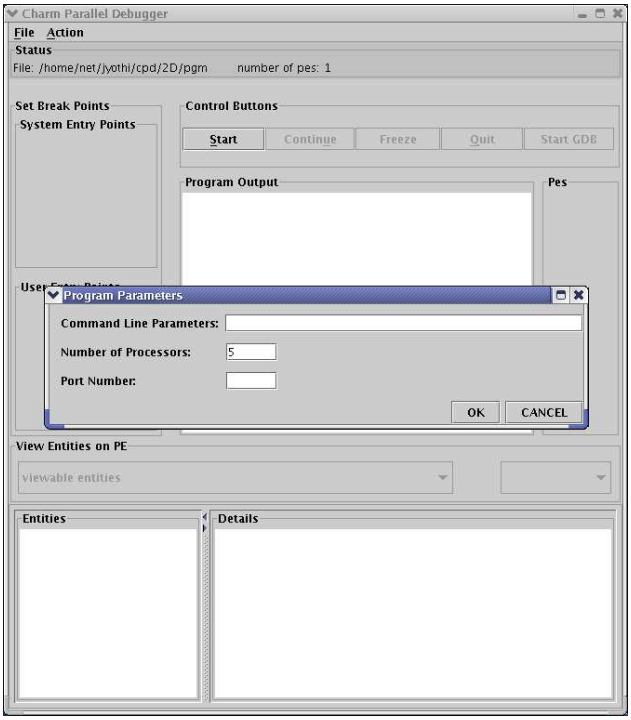

Figure 1: Using the menu to set parameters for the CHARM++ program being debugged

Once the debugger client's GUI loads, the programmer triggers the program execution by clicking the *Start* button. The program starts off displaying the user and system entry points as a list of check boxes, freezing at the onset. The system entry points are entry methods written for the chares in the charm kernel while the user entry points are the entry methods in the application program being debugged by the programmer. The programmer could choose to set or remove breakpoints by checking or unchecking the checkboxes corresponding to the entry points and kick off execution by clicking the *Continue* Button. The program freezes when a breakpoint is reached. Figure 2 shows a snapshot of the debugger when a breakpoint is reached.

Clicking the *Freeze* button during the execution of the

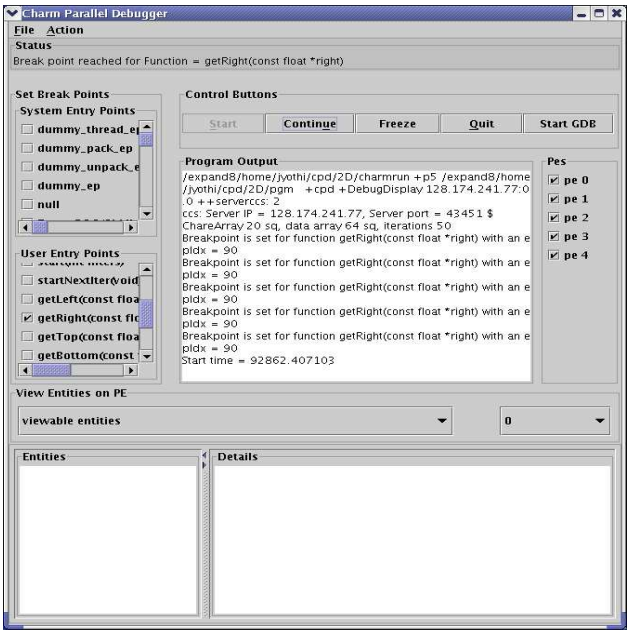

Figure 2: Parallel debugger when a break point is reached

program freezes execution, while *Continue* button resumes execution. *Quit* button can be used to suspend execution at any point of time. Entities (for instance, array elements) and their contents on any processor can be viewed at any point in time during execution as illustrated in Figure 3. Every array element inherits a virtual debugging function used by the debugging framework in the Charm kernel. The programmer could choose to override this virtual function for an array element which it inherits from the array element prototype in the Charm kernel and thereby control the information displayed by the debugger for the array elements. The programmer could make the data more readable by specifically choosing what contents of the array element should be displayed and by inserting appropriate comments. In case this debugging function is not implemented, by default, the regular pack/unpack function of the array element as part of Charm's PUP framework (Sec. 4) is used to retrieve its contents. The Converse scheduler which is the core of the charm kernel interacts with a pool of messages, placed in queues on each of the processors. These messages could be generated locally or could be from remote processors. In CHARM++, a message could be due to an entry method invocation, a ready thread, a message sent to a ready thread or a handler posted previously. A message is a chunk of memory with a header and data. The debugger allows the user to freeze the program and inspect the messages in the queues. From the data part of the CHARM++ message the debugging framework encodes the destination object, the method being invoked and the parameters for the user to interpret.

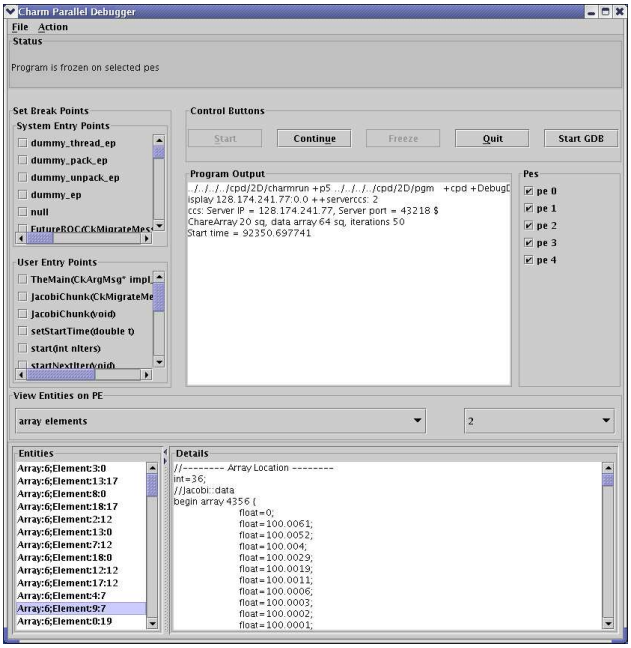

Figure 3: Freezing program execution and viewing the contents of an array element using the Parallel Debugger

Specific individual processes of the CHARM++ program can be attached to instances of *gdb* during the course of program execution as shown in Figure 4. This is specially useful when the number of processor elements is high and sequential debugging is sought on specific processors. Using the runtime "*++debug*" option in such a situation is cumbersome as it becomes very difficult to keep track of the large number of *xterm* windows opened at the onset of execution for each instance of the sequential debugger attached to every process in the parallel program.

The record and replay mechanism allows a user to reproduce a program's execution and thereby catch bugs which occur due to order of message execution. The key idea here is to identify atomic events and record their order of occurrence in the execution. The execution can be replayed by re-executing the atomic events in their recorded order of occurrence.

The first execution run or the record run is used to collect minimum trace data for every message processed in the CHARM++ program. Since messages are atomic events in a Charm program this information is sufficient for replay. This trace is used to replay the program ex-

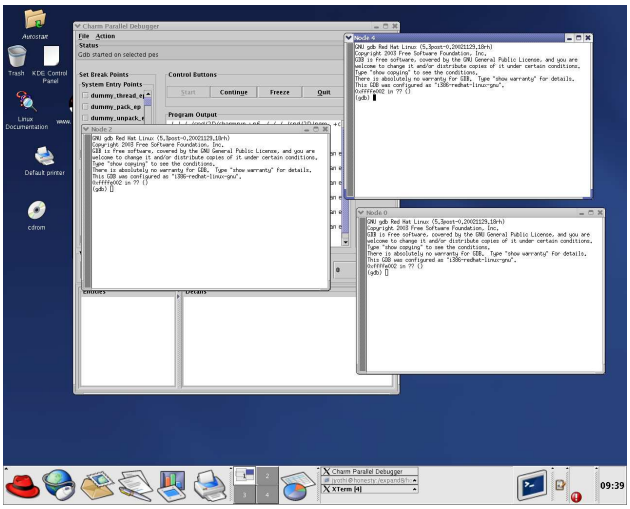

Figure 4: Parallel debugger showing instances of *gdb* open for the selected processor elements

ecution by re-executing each message on each processor as recorded in the trace. The only information needed to replay a CHARM++ program is the order of processing of events on each processor. An event can be uniquely identified by a tuple consisting the following pair (i) message-sequence-id (ii) processor-id. The trace data necessary for replay is an ordered set of such tuples where the ordering is imposed by the order in which events occurred on that processor. To enable the required tracing for record and replay, a CHARM++ program has to be linked with the appriopriate trace module, which is made possible via a link-time option "*-tracemode recordreplay*". A "*+record*" option provided at runtime to a CHARM++ program writes such a trace to a file, which is later read when the "*+replay*" runtime option is provided to ensure messages are processed in the same order as the recorded run. These runtime options are passed as parameters to the parallel program and used in conjunction with the parallel debugger, when the bugs need to be reproduced deterministically in the program.

4 Implementation

Implementation of debugging solutions for CHARM++ is two-fold - (i) incorporating the required support into the CHARM++ core or the Charm kernel and (ii) Building the user interface. While the implementation of debugging support in the Charm Kernel relies heavily on the pack/unpack or PUP framework available in the core, the debugger's user interface is modelled as a client in the Converse Client-Server (CCS) Interface provided by CHARM++.

The pack/unpack framework or the "PUP" framework is a generic way provided by CHARM++ to describe the data in an object. It is a suite of classes that enables objects in CHARM++ (for example, array elements) to migrate from one processor to another. In a nutshell, the framework provides services to any operation that requires a traversal of the object state in terms of its data members. The CHARM++ system uses the generic description of an object, provided by the PUP framework, to pack the particular object into a message and later unpack the message into a new object on another processor. Thus the name, pack/unpack framework. Besides being used in transporting objects intact across processors during migration, the PUP framework can also be used to serialize an object's data to disk. The framework can also be used to retrieve an object's data in some interpretable form and this functionality is used in the implementation of the debugger. The PUP framework requires the programmer of a particular class to implement a single method, called the `pup` routine. A significant part of the debugging framework is the introspection API which allows the debugger to retrieve the contents of parallel entities during program execution using the PUP framework. This API is used to register a list of items with pup routines and retrieve the items' data out in a readable format.

The Converse Client-Server (CCS) module enables Converse [6] programs to act as parallel servers, responding to requests from non-Converse programs. The CCS module is split into two parts - client and server. The server side is used by a Converse program while the client side is used by arbitrary non-Converse programs. A CCS client accesses a running Converse program by talking to CCS server side which receives the CCS requests and relays them to the appropriate processor. These CCS requests trigger the invocation of appropriate pre-registered handlers in the Charm kernel, which perform the required actions. The parallel debugger acts as the CCS client and sends messages to the CHARM++ program being debugged which is started as a parallel server in the CCS model and carries out the appropriate debugging actions.

5 Conclusions and Future Work

In devising debugging solutions for CHARM++ many of the challenges posed by a data-driven parallel program-

ming paradigm had to be got around. The parallel debugger allows the CHARM++ programmer to inspect the state of the parallel program during execution, keep track of the control flow in the parallel program by setting break points at entry points and retrieve contents of entities like array elements nd the messages in the queues on each processor . In this way, via the debugger, the programmer is provided a means of examining the dynamic state of the program. The implemented support also provides the programmer a way of going about sequential debugging on selected processors on the fly. The record and replay mechanism allows the programmer to deterministically reproduce a program's behavior.

The functionality of the parallel debugger can be extended such that it could be used in conjunction with the existing performance analysis tool for CHARM++, **Projections** [15]. There is scope for enhancement of the debugging support to make use of the huge amounts of trace data produced by **Projections**. It would also be very useful to incorporate methods to access network statistics into the existing debugging support. These are just a few of the many possible extensions. The topic of debugging in CHARM++ is a green area and there is lots of scope for future work.

References

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

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

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

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

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

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

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

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

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

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

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

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

[13] A.B. Sinha. *Performance Analysis of Object Based and Message Driven Programs.* PhD thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, January 1995.

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

[15] L.V. Kalé and Amitabh Sinha. Projections : A scalable performance tool. In *Parallel Systems Fair, International Parallel Processing Sympos ium*, pages 108–114, April 1993.