Debugging Parallel Applications via Provisional Execution

Filippo Gioachin
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: gioachin@ieee.org

Laxmikant Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Email: kale@illinois.edu

***Abstract*—While debugging message-passing parallel applications, a notoriously difficult bug to solve is a race condition: messages from different sources may arrive at a processor in any order, and the processing of these messages in some particular order is not handled correctly by the application. What compounds the difficulty is that this bug may not manifest itself until the application is deployed at large scale, and even then it may happen only in a small fraction of runs.**

By allowing a developer to quickly test the outcome of the computation resulting from handling several messages in a specific order while the application is running, we enable a fast way to inspect the application for the existence of race conditions. Furthermore, this can help reduce the need for large machine allocations to discover race conditions. A fundamental component of the feature is the capability to perform the provisional delivery of a message quickly, providing an equally fast rollback.

I. Introduction

Debugging is a fundamental component of software development, and one of the most time consuming. When it comes to parallel computing, discovering the cause of a problem becomes even more difficult. One of the reasons for this increased complexity is the presence of race conditions. This means that when the application is executed repeatedly with the same input, the application can behave differently. This is distinct from the concept of non-determinism, where an application may produce different results, but all valid solutions. In a race condition, some of the possible message interleavings produce an incorrect result. Different message interleaving can happen, for example, when the latency introduced by the switches in the communication network delays some messages more than others.

To eliminate non-determinism from an application, a commonly used technique is record-replay. The idea is to record the order in which messages are processed during an execution, and enforce this order in subsequent executions. Several tools have explored this possibility [1], [2], [3]. With this technique, 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 some recording techniques are less intrusive than others, and they can reduce the probe effect, it cannot be taken for granted that the bug will appear while recording is enabled. For some applications, moreover, the minimal recording necessary to capture the bug may include large amount of data, such as timers or other system calls. Again, how much time will the user be willing to spend waiting for the bug to present itself in 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 IX 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 to try certain combinations. These permutations of messages could be random, or due to his understanding of the program, or a hunch he might have. The other, more automatic method, would be to have the system explore the possible delivery orderings, and report to the user when different orderings generate different solutions. In this paper, we shall lay the foundations to allow the testing of possible different execution paths, but consider only paths specified by the user. In the future work section, we shall expand on automatic search, and further challenges that it poses.

In this paper, after some background, we illustrate what features are useful to a programmer debugging his or her parallel application in Section III, and in Section IV we explore possible ways in which a debugging system can provide to a programmer the capability to force a particular processing order of messages, while retaining the possibility to rollback and try a different ordering. We describe an implementation of such debugging system in Section V, we demonstrate that our system performs well and can be used interactively in Section VI, and present a a case study in Section VII. Related work will be compared in Section VIII, followed by conclusions and future work.

II. Background

The technique presented in this paper has been deployed in the context of the CHARM++ parallel programming model and the CHARMDEBUG debugger. In this section, we review the main concepts and infrastructure that are relevant to this work.

A. Charm++

CHARM++ is an increasingly popular programming model to develop parallel applications. It is based on the concepts of object-based virtualization and message driven execution [4], [5]. In CHARM++, the programmer is responsible for decomposing the computation into logically independent entities, called *chares*, which live in a global object space. This space spans the entire application, and chares can communicate with each other by sending messages. The parallel computation proceeds as messages trigger the execution of methods on the destination chares. The parallel runtime system is responsible for mapping the chares to the allocated processors. This mapping can be modified dynamically as the application evolves to compensate for load imbalance or to optimize the communication among chares. The runtime system is also responsible for routing the messages, and delivering them to their destination chares, wherever the destination chares are located.

Messages exchanged are asynchronous, and the order in which they are delivered to the destination chares is non-deterministic. In particular, this is valid for messages transferred over the network, where latencies may alter the receipt order at the destination processor. CHARM++ provides a mechanism to record the order in which messages were processed in one execution, and replay it later, thus eliminating the non-determinism from the application [3]. This mechanism is based on the piecewise deterministic assumption. This states that if a chare *C* processes a message *M* when in state S_0, then it will always transition to the same state S_1 and send the same messages $\{msgs\}$. I.e, if the order in which messages are processed by a chare remains the same (along with their contents), the chare will perform the same computation and send out the same sequence of messages.

B. CharmDebug

CHARMDEBUG is a parallel debugging environment where the user interacts with a graphical user interface, and is tailored to CHARM++ applications. This environment is tightly integrated with the runtime system itself, and is therefore capable of providing the user with the most relevant information, such as the messages in the system, the state of the chares, and their entry methods [6].

The communication between the CHARMDEBUG GUI, which may be running on the user's personal machine, and the parallel application running on a large machine, happens through the Converse Client-Server (CCS) protocol [7]. This protocol is used over a single communication link between the GUI and the parallel application. This, in conjunction with the possibility to use the message driven execution model to interleave debugging requests with normal application messages, enables the debugging infrastructure to scale to very large machines together with the application itself.

III. Provisionally Delivering Messages

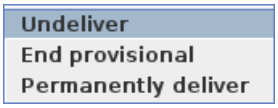
When considering how the user should be allowed to interact with the system to test his hypothesis, several decisions can be made. In particular, these involve how the system will perform the delivery operation, and how it will allow the rollback if the user decides to undo the delivery operation. We 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. The options available depends on the state of the system and the message selected. Figure 1 shows the different options available. When a processor is not in *provisional mode* (Figure 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 1(b)). For messages that have been provisionally delivered (Figure 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 2).

(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 1. Options available for different system status and message selected.

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 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.

IV. 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

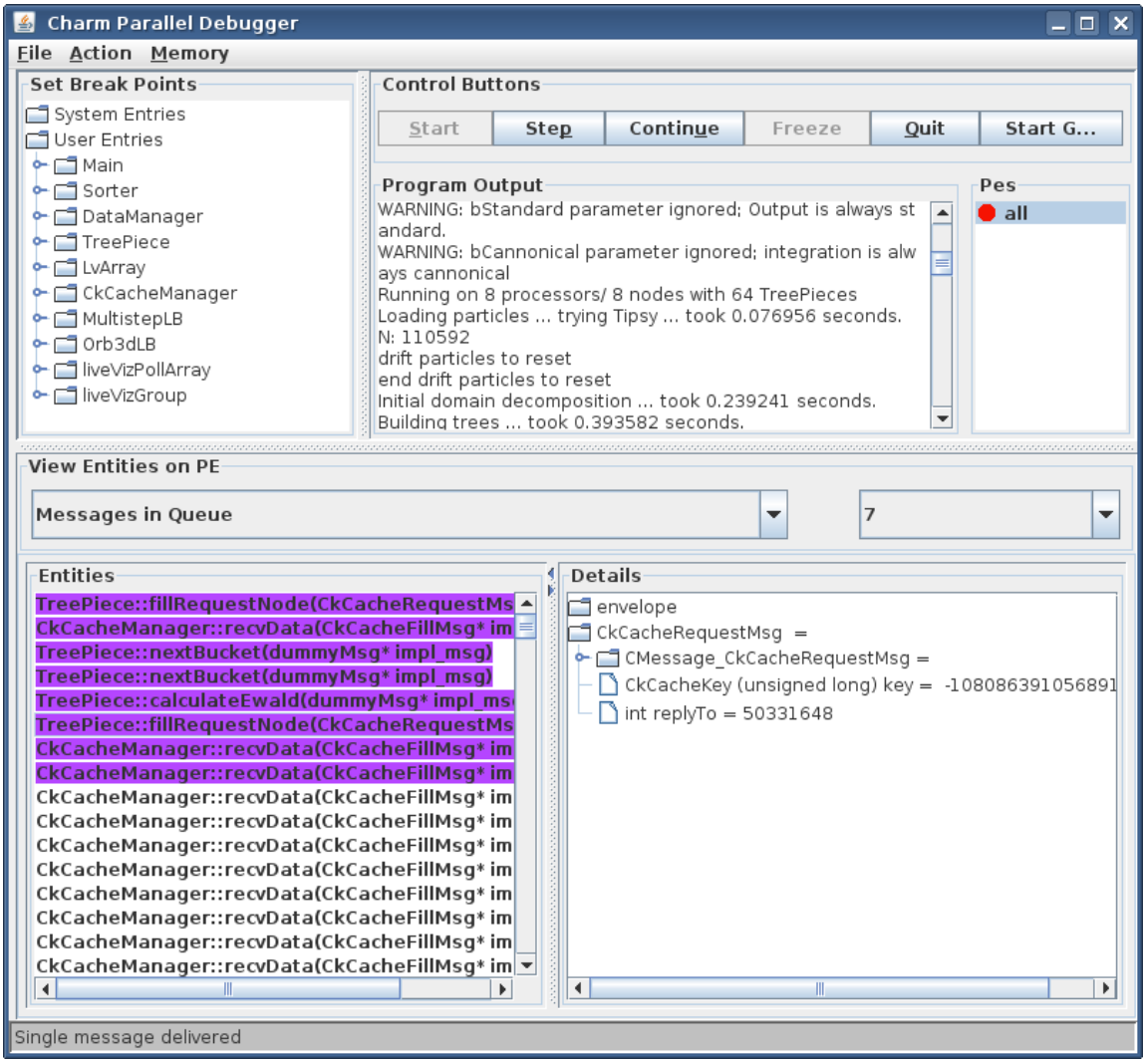


Figure 2. Screenshot of the message queue. Some messages have been delivered provisionally (in purple on top), while others are still in the regular queue.

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 in CHARM++ can provide better support to cover the problems mentioned above. Double in-memory checkpoint [8] can tolerate the rollback without having to restart the application from disk, and therefore avoiding potential problems with job schedulers. Message logging [9] 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 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 different. 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. For the purposes of this paper, these issues will not be considered further.

This solution using process forking, as well as the others, builds upon the piecewise deterministic assumption described in Section II-A. 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.

V. 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 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.

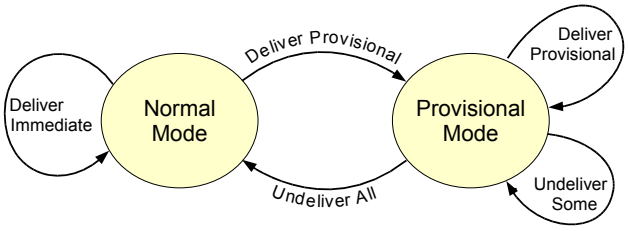


Figure 3. Description of the behavior of a processor when provisionally delivering messages.

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 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.

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.

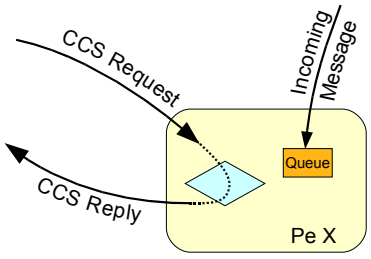
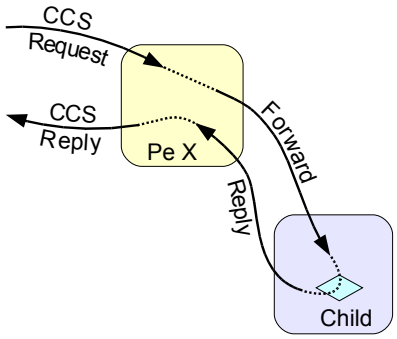
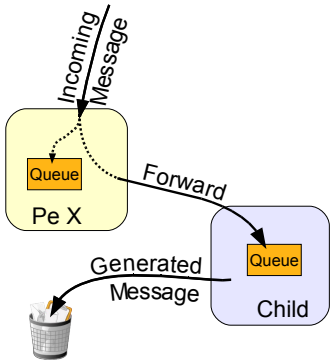


Figure 4. Control flow of a processor in normal mode. When a CCS request arrives, the processor handles it and replies to the sender.



(a) Response to a CCS external message.



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

Figure 5. Control flow of a processor when in provisional mode.

Figure 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 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 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 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.

A. 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 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 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.

B. Single Forked Process

Figure 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 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.

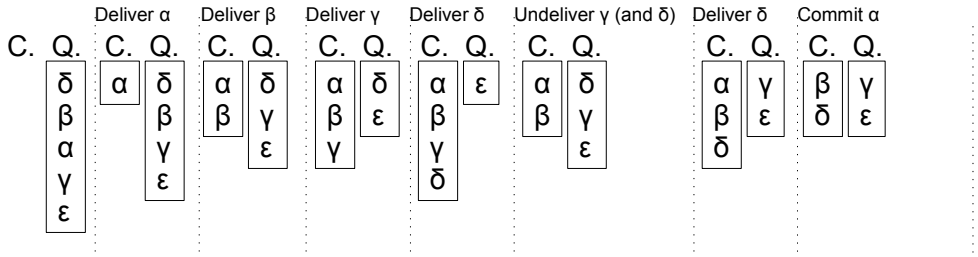
When the request to undeliver γ arrives, process C terminates execution, indicating that a rollback should be performed. The parent process P is notified of the termination of the child via the closing of the pipe which connects the two processes. At this point, P needs to fork another child process 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 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).

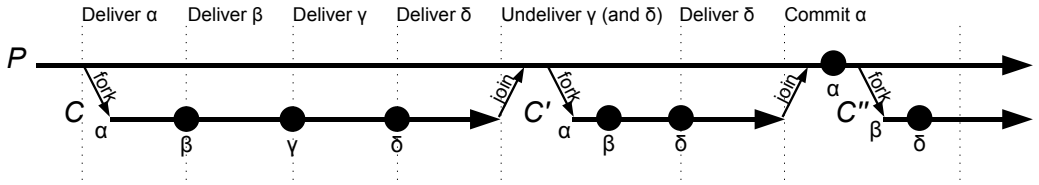
Note that the combination of these two numbers covers any possible operation the user may want, from full rollback $\{0, 0\}$, to full commit $\{n, 0\}$ (where n is the number of messages provisionally delivered), plus anything in the middle $\{n - k, k\}$ ($\forall k : 0 < k < n$). Also note that the 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.

C. Multiple Forked Processes

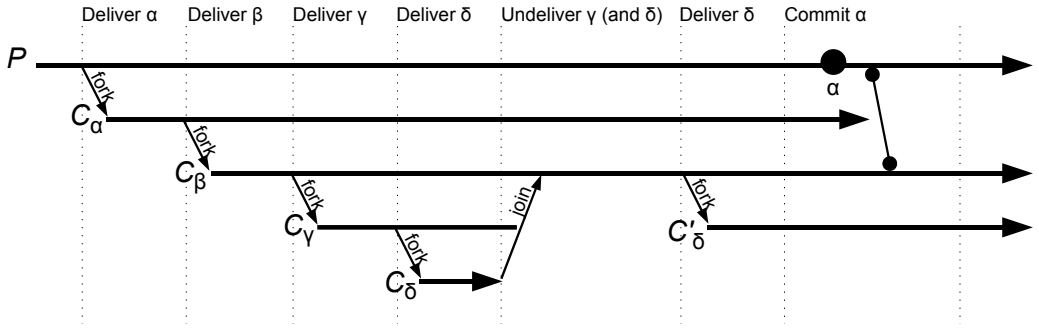
Another alternative to manage multiple provisional delivery of messages is to fork a new child for every message that is provisionally delivered. This is shown in Figure 6(c). Each of the four messages α, β, γ, and δ generates a new process that we will denote C_α, C_β, C_γ, and C_δ. Between each couple 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.



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

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.

VI. 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 I shows the performance for various operations performed by delivering and undelivering messages provisionally.

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.

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

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 II. Given the large variances obtained in the experimental setup, we leave this model in symbolic notation.

	Single process	Multiple processes
First provisional msg	$c + m$	$c + m$
Foll. provisional msgs	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 II
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.

VII. 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. At each step i of the algorithm, processor p sends its current value to processor $p + 2^i$.

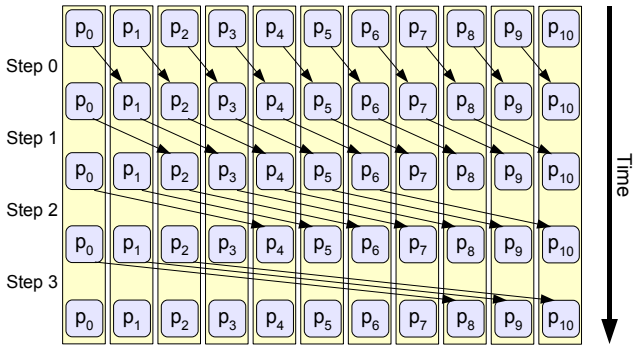


Figure 7. Parallel algorithm for prefix computation.

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 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 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. In this case, a series of messages were racing during a parallel sorting operation, 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.

VIII. 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 [10], 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 [11] and for MPI applications [12]. An interesting extension to the generation of possible orderings of message delivery, and how to explore the generated space without maintaining all possible executions active, is presented in [13].

A tool to allow the user to select messages to be delivered in different order is MAD [14]. 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.

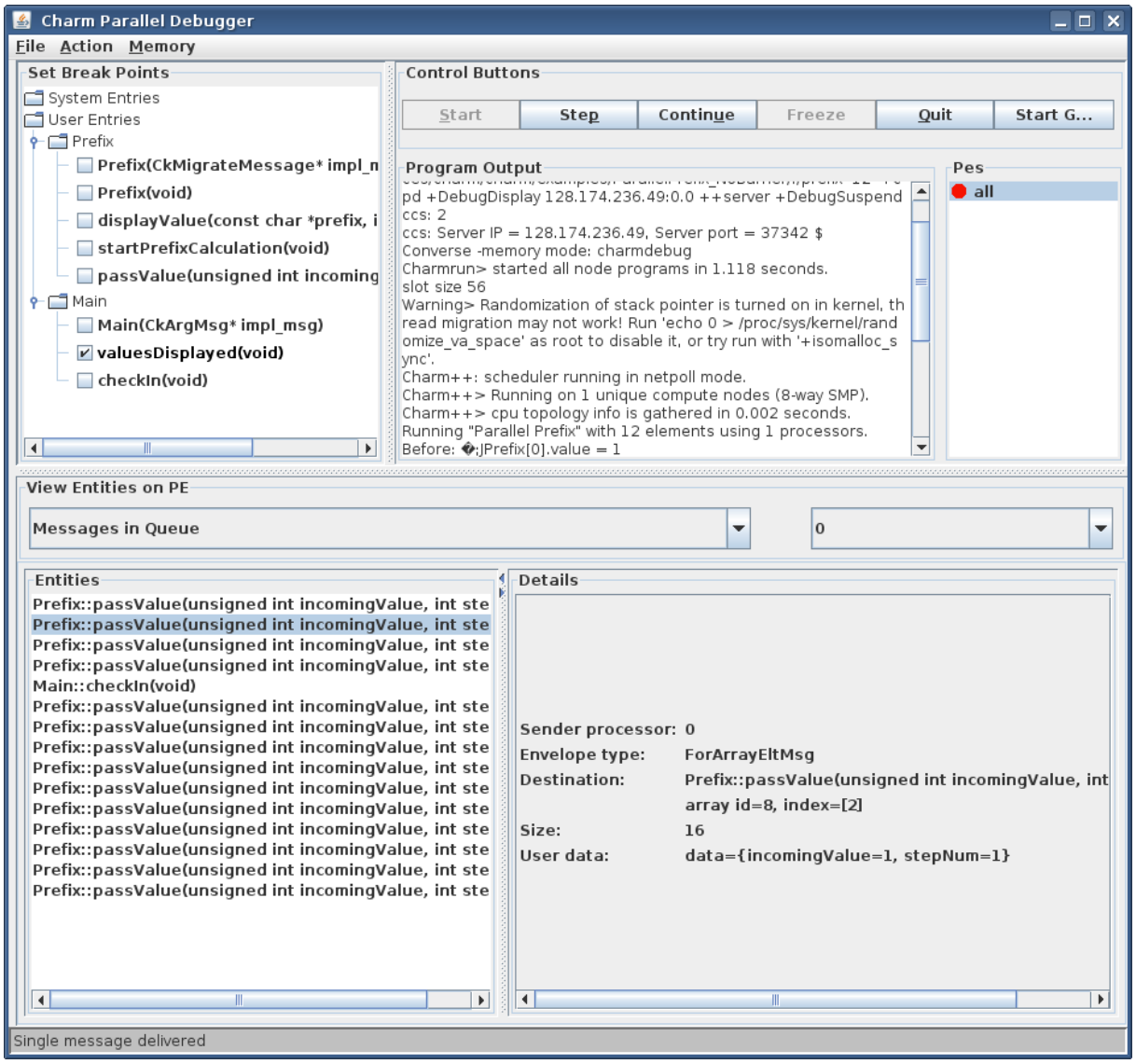


Figure 8. Screenshot of CHARMDEBUG while debugging the parallel prefix application. Multiple messages are enqueued from different steps.

A similar implementation, using the fork system call, is available through the GDB debugger [15] 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.

IX. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a novel approach to enable a programmer to effectively search for race conditions in a parallel program. Given an idea he might have on where the problem is located, he can quickly confirm or refute his idea by testing what happens when messages are processed in a certain order. One major advantage of our approach is that the application is still running while the user makes his decision, and the entire procedure is interactive, with a fast response from the application to the user commands.

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.

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.

ACKNOWLEDGMENT

This work was supported in part by the NSF Grant OCI-0725070 for Blue Waters.

REFERENCES

[1] M. Ronsse and K. De Bosschere, "RecPlay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.

[2] C. Gottbrath, "Quickly Identifying the Cause of Software Bugs with ReplayEngine," TotalView Technologies, Tech. Rep., August 2008.

[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.

[3] F. Gioachin, G. Zheng, and L. V. Kalé, "Robust Record-Replay with Processor Extraction," in *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII)*, Trento, Italy, July 2010.

[4] L. V. Kale and S. Krishnan, "Charm++: Parallel Programming with Message-Driven Objects," in *Parallel Programming using C++*, G. V. Wilson and P. Lu, Eds. MIT Press, 1996, pp. 175–213.

[5] L. 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.

[6] F. Gioachin, "Debugging Large Scale Applications with Virtualization," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, September 2010.

[7] F. Gioachin, C. W. Lee, and L. V. Kalé, "Scalable Interaction with Parallel Applications," in *Proceedings of TeraGrid'09*, Arlington, VA, USA, June 2009.

[8] G. Zheng, L. Shi, and L. V. Kalé, "FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI," in *2004 IEEE International Conference on Cluster Computing*, San Dieago, CA, September 2004, pp. 93–103.

[9] S. 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.

[10] R. Kilgore and C. Chase, "Re-execution of Distributed Programs to Detect Bugs Hidden by Racing Messages," in *In Proceedings of the International Conference on System Sciences*, 1997, p. 423.

[11] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, no. 1, pp. 110–121, 2005.

[12] S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. Kirby, R. Thakur, and W. Gropp, "Implementing efficient dynamic formal verification methods for mpi programs," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2008, vol. 5205, pp. 248–256, 10.1007/978-3-540-87475-1-34. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1-34

[13] B. Schaeli and R. D. Hersch, "Dynamic testing of flow graph based parallel applications," in *PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems*. New York, NY, USA: ACM, 2008, pp. 1–10.

[14] D. Kranzlmüller, C. Schaubschläger, and J. Volkert, "A Brief Overview of the MAD Debugging Activities," in *AADEBUG*, 2000.

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