Checkpoint/Restart-Enabled Parallel Debugging

Joshua Hursey[1], Chris January[2], Mark O'Connor[2], Paul H. Hargrove[3], David Lecomber[2] Jeffrey M. Squyres[4], and Andrew Lumsdaine[1]

[1] Open Systems Laboratory, Indiana University {jjhursey,lums}@osl.iu.edu *
[2] Allinea Software Ltd. {cjanuary,mark,david}@allinea.com
[3] Lawrence Berkeley National Laboratory PHHargrove@lbl.gov **
[4] Cisco Systems, Inc. jsquyres@cisco.com

Abstract. Debugging is often the most time consuming part of software development. HPC applications prolong the debugging process by adding more processes interacting in dynamic ways for longer periods of time. Checkpoint/restart-enabled parallel debugging returns the developer to an intermediate state closer to the bug. This focuses the debugging process, saving developers considerable amounts of time, but requires parallel debuggers cooperating with MPI implementations and checkpointers. This paper presents a design specification for such a cooperative relationship. Additionally, this paper discusses the application of this design to the GDB and DDT debuggers, Open MPI, and BLCR projects.

1 Introduction

The most time consuming component of the software development life-cycle is application debugging. Long running, large scale High Performance Computing (HPC) parallel applications compound the time complexity of the debugging process by adding more processes interacting in dynamic ways for longer periods of time. Cyclic or iterative debugging, a commonly used debugging technique, involves repeated program executions that assist the developer in gaining an understanding of the causes of the bug. Software developers can save hours or days of time spent debugging by checkpointing and restarting the parallel debugging session at intermediate points in the debugging cycle. For Message Passing Interface (MPI) applications, the parallel debugger must cooperate with the MPI implementation and Checkpoint/Restart Service (CRS) which account for the network state and process image. We present a design specification for this cooperative relationship to provide Checkpoint/Restart (C/R)-enabled parallel debugging.

The C/R-enabled parallel debugging design supports multi-threaded MPI applications without requiring any application modifications. Additionally, all checkpoints, whether generated with or without a debugger attached, are usable

* Supported by grants from the Lilly Endowment; National Science Foundation EIA-0202048; and U.S. Department of Energy DE-FC02-06ER25750ˆA003.
** Supported by the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

within a debugging session or during normal execution. We highlight the *debugger detach* and *debugger reattach* problems that may lead to inconsistent views of the debugging session. This paper presents a solution to these problems that uses a thread suspension technique which provides the user with a consistent view of the debugging session across repeated checkpoint and restart operations of the parallel application.

2 Related Work

For HPC applications, the MPI [1] standard has become the *de facto* standard message passing programming interface. Even though some parallel debuggers support MPI applications, there is no official standard interface for the interaction between the parallel debugger and the MPI implementation. However, the MPI implementation community has informally adopted some consistent interfaces and behaviors for such interactions [2, 3]. The MPI Forum is discussing including these interactions into a future MPI standard. This paper extends these debugging interactions to include support for C/R-enabled parallel debugging.

C/R rollback recovery techniques are well established in HPC [4]. The checkpoint, or snapshot, of the parallel application is defined as the state of the process and all connected communication channels [5]. The state of the communication channels is usually captured by a C/R-enabled MPI implementation, such as Open MPI [6]. Although C/R is not part of the MPI standard, it is often provided as a transparent service by MPI implementations [6–9]. The state of the process is captured by a Checkpoint/Restart Service (CRS), such as Berkeley Lab Checkpoint/Restart (BLCR) [10]. The combination of a C/R-enabled MPI and a CRS provide consistent global snapshots of the MPI application. Often global snapshots are used for fault recovery, but, as this paper demonstrates, can also be used to support reverse execution while debugging the MPI application. For an analysis of the performance implications of integrating C/R into an MPI implementation we refer the reader to previous literature on the subject [9, 11].

Debugging has a long history in software engineering [12]. Reverse execution or back-stepping allows a debugger to either step backwards through the program execution to a previous state, or step forward to the next state. Reverse execution is commonly achieved though the use of checkpointing [13, 14], event/message logging [15, 16], or a combination of the two techniques [17, 18]. When used in combination, the parallel debugger restarts the program from a checkpoint and replays the execution up to the breakpoint. A less common implementation technique is the actual execution of the program code in reverse without the use of checkpoints [19]. This technique is often challenged by complex logical program structures which can interfere with the end user behavior and applicability to certain programs.

Event logging is used to provide a deterministic re-execution of the program while debugging. Often this allows the debugger to reduce the number of processes involved in the debugging operation by simulating their presence through replaying events from the log. This is useful when debugging an application with

a large number of processes. Event logging is also used to allow the user to view a historical trace of program execution without re-execution [20, 21].

C/R is used to return the debugging session to an intermediary point in the program execution without replaying from the beginning of execution. For programs that run for a long period of time before exhibiting a bug, C/R can focus the debugging session on a smaller period of time closer to the bug. C/R is also useful for program validation and verification techniques that may run concurrently with the parallel program on smaller sections of the execution space [22].

3 Design

C/R-enabled parallel debugging of MPI applications requires the cooperation of the parallel debugger, the MPI implementation, and the CRS to provide consistently recoverable application states. The debugger provides the interface to the user and maintains state about the parallel debugging session (e.g., breakpoints, watchpoints).

The C/R-enabled MPI implementation marshals the network channels around C/R operations for the application. Though the network channels are often marshaled in a fully coordinated manner, this design does not require full coordination. Therefore the design is applicable to other checkpoint coordination protocol implementations (e.g., uncoordinated).

The CRS captures the state of a single process in the parallel application. This can be implemented at the user or system level. This paper requires an MPI application transparent CRS, which excludes application level CRSs. If the CRS is not transparent to the application, then taking the checkpoint would alter the state of the program being debugged, potentially confusing the user.

One goal of this design is to create *always usable checkpoints*. This means that regardless of whether the checkpoint was generated with the debugger attached or not, it must be able to be used on restart with or without the debugger. To provide the *always usable checkpoints* condition, the checkpoints generated by the CRS with the debugger attached must be able to exclude the debugger state. To achieve this, the debugger must detach from the process before a checkpoint and reattach, if desired, after the checkpoint has finished, similar to the technique used in [17]. Since we are separating the CRS from the debugger, we must consider the needs of both in our design.

In addition to the *always usable checkpoints* goal, this technique supports multi-threaded MPI applications without requiring any explicit modifications to the target application. Interfaces are prefixed with `MPIR_` to fit the existing naming convention for debugging symbols in MPI implementations.

3.1 Preparing for a Checkpoint

The C/R-enabled MPI implementation may receive a checkpoint request internally or externally from the debugger, user, or system administrator. The MPI implementation communicates the checkpoint request to the specified processes

```
volatile int MPIR_checkpoint_debug_gate = 0;
volatile int MPIR_debug_with_checkpoint = 0;
int MPIR_checkpoint_debugger_detach(void) { return 0; } // Detach Function
void MPIR_checkpoint_debugger_waitpoint(void) { // Thread Wait Function
  // MPI Designated Threads are released early,
  // All other threads enter the breakpoint below
  MPIR_checkpoint_debug_gate = 0;
  MPIR_checkpoint_debugger_breakpoint();
}
void MPIR_checkpoint_debugger_breakpoint(void) { // Debugger Breakpoint Func.
  while( MPIR_checkpoint_debug_gate == 0 ) { sleep(1); }
}
void MPIR_checkpoint_debugger_crs_hook(int state) { // CRS Hook Callback Func.
  if( MPIR_debug_with_checkpoint ) {
    MPIR_checkpoint_debug_gate = 0;
    MPIR_checkpoint_debugger_waitpoint();
  } else { MPIR_checkpoint_debug_gate = 1; }
}
```

Fig. 1. Debugger `MPIR_` function pseudo code

(usually all processes) in the MPI application. The MPI processes typically prepare for the checkpoint by marshaling the network state and flushing caches before requesting a checkpoint from the CRS.

If the MPI process is under debugger control at the time of the checkpoint, then the debugger must allow the MPI process to prepare for the checkpoint uninhibited by the debugger. If the debugger remains attached, it may interfere with the techniques used by the CRS to preserve the application state (e.g., by masking signals). Additionally, by detaching the debugger before the checkpoint, the implementation can provide the *always usable checkpoints* condition by ensuring that it does not inadvertently include debugger state in the CRS generated checkpoint.

The MPI process must inform the debugger of when to detach since the debugger is required to do so before a checkpoint is requested. The MPI process informs the debugger by calling the `MPIR_checkpoint_debugger_detach()` function when it requires the debugger to detach. This is an empty function that the debugger can reference in a breakpoint. It is left to the discretion of the MPI implementation when to call this function while preparing for the checkpoint, but it must be invoked before the checkpoint is requested from the CRS.

The period of time between when the debugger detaches from the MPI process and when the checkpoint is created by the CRS may allow the application to run uninhibited, we call this the *debugger detach problem*. To provide a seamless and consistent view to the user, the debugger must make a best effort attempt at preserving the exact position of the program counter(s) across a checkpoint operation. To address the *debugger detach problem*, the debugger forces all threads

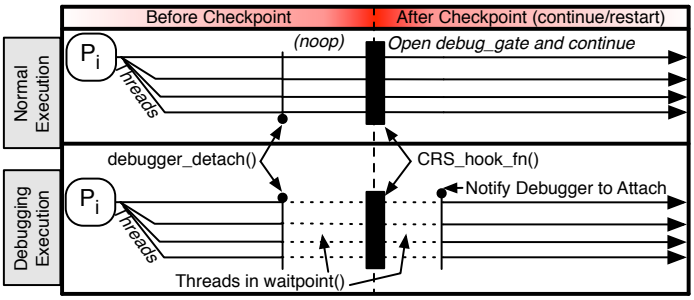

Fig. 2. Illustration of the design for each of the use case scenarios.

into a waiting function (called `MPIR_checkpoint_debugger_waitpoint()`) at the current debugging position before detaching from the MPI process. By forcing all threads into a waiting function the debugger prevents the program from making any progress when it returns from the checkpoint operation.

The waiting function must allow certain designated threads to complete the checkpoint operation. In a single threaded application, this would be the main thread, but in a multi-threaded application this would be the thread(s) designated by the MPI implementation to prepare for and request the checkpoint from the CRS. The MPI implementation must provide an "early release" check for the designated thread(s) in the waiting function. All other threads directly enter the `MPIR_checkpoint_debugger_breakpoint()` function which waits in a loop for release by the debugger. Designated thread(s) are allowed to continue normal operation, but must enter the breakpoint function after the checkpoint has completed to provide a consistent state across all threads to the debugger, if it intends on reattaching. Figure 1 presents a pseudo code implementation of these functions.

The breakpoint function loop is controlled by the `MPIR_checkpoint_debug_gate` variable. When this variable is set to `0` the gate is *closed*, keeping threads waiting for the gate to be *opened* by the debugger. To *open* the gate, the debugger sets the variable to a non-zero value, and steps each thread out of the loop and the breakpoint function. Once all threads pass through the gate, the debugger then *closes* it once again by setting the variable back to `0`.

3.2 Resuming After a Checkpoint

An MPI program either *continues* after a requested checkpoint in the same program, or is *restarted* from a previously established checkpoint saved on stable storage. In both cases the MPI designated thread(s) are responsible for recovering the internal MPI state including reconnecting processes in the network.

If the debugger intends to attach, the designated thread(s) must inform the debugger when it is safe to attach after restoring the MPI state of the process. If the debugger attaches too early, it may compromise the state of the checkpoint

or the restoration of the MPI state. The designated thread(s) notify the debugger that it is safe to attach to the process by printing to `stderr` the hostname and PID of each recovered process. The message is prefixed by "`MPIR_debug_info)`" as to distinguish it from other output. Afterwards, the designated thread(s) enter the breakpoint function.

The period of time between when the MPI process is restarted by the CRS and when the debugger attaches may allow the application to run uninhibited, we call this the *debugger reattach problem*. If the MPI process was being debugged before the checkpoint was requested, then the threads are already being held in the breakpoint function, thus preventing them from running uninhibited. However, if the MPI process was not being debugged before the checkpoint then the user may experience inconsistent behavior due to the race to attach the debugger upon multiple restarts from the same checkpoint.

To address this problem, the CRS must provide a *hook* callback function that is pushed onto the stack of all threads before returning them to the running state. This technique preserves the individual thread's program counter position at the point of the checkpoint providing a best effort attempt at a consistent recovery position upon multiple restarts. The MPI implementation registers a hook callback function that will place all threads into the waiting function if the debugger intends to reattach. The intention of the debugger to reattach is indicated by the `MPIR_debug_with_checkpoint` variable. Since the hook function is the same function used when preparing for a checkpoint, the release of the threads from the waiting function is consistent from the perspective of the debugger.

If the debugger is not going to attach after the checkpoint or on restart, the hook callback does not need to enter the waiting function, again indicated by the `MPIR_debug_with_checkpoint` variable. Since the checkpoint could have been generated with a debugger previously attached, the hook function must release all threads from the breakpoint function by setting the `MPIR_checkpoint_debug_gate` variable to `1`. The structure of the hook callback function allows for checkpoints generated while debugging to be used without debugging, and vice versa.

3.3 Additional `MPIR_` Symbols

In addition to the detach, waiting, and breakpoint functions, this design defines a series of support variables to allow the debugger greater generality when interfacing with a C/R-enabled MPI implementation.

The `MPIR_checkpointable` variable indicates to the debugger that the MPI implementation is C/R-enabled and supports this design when set to `1`. The `MPIR_debug_with_checkpoint` variable indicates to the MPI implementation if the debugger intends to attach. If the debugger wishes to detach from the program, it sets this value to `0` before detaching. This value is set to `1` when the debugger attaches to the job either while running or on restart.

The `MPIR_checkpoint_command` variable specifies the command to be used to initiate a checkpoint of the MPI process. The output of the checkpoint command must be formatted such that the debugger can use it directly as

an argument to the restart command. The output on `stderr` is prefixed with "`MPIR_checkpoint_handle)`" as to distinguish it from other output. The `MPIR_restart_command` variable specifies the restart command to prefix the output of the checkpoint command to restart an MPI application. The `MPIR_controller_hostname` variable specifies the host on which to execute the `MPIR_checkpoint_command` and `MPIR_restart_command` commands. The `MPIR_checkpoint_listing_command` variable specifies the command that lists the available checkpoints on the system.

4 Use Case Scenarios

To better illustrate how the various components cooperate to provide C/R-enabled parallel debugging we present a set of use case scenarios. Figure 2 presents an illustration of the design for each scenario.

Scenario 1: No Debugger Involvement This is the standard C/R scenario in which the debugger is neither involved before a checkpoint nor afterwards. A transition from the upper-left to upper-right quadrants in Figure 2. The MPI processes involved in the checkpoint will prepare the internal MPI state and request a checkpoint from the CRS then continue free execution afterwards.

Scenario 2: Debugger Attaches on Restart In this scenario, the debugger is attaching to a restarting MPI process from a checkpoint that was generated without the debugger. A transition from the upper-left to lower-right quadrants in Figure 2. This scenario is useful when repurposing checkpoints originally generated for fault tolerance purposes instead for debugging. The process of creating the checkpoints is the same as in Scenario 1.

On restart, the hook callback function is called by the CRS in each thread to preserve their program counter positions. Once the MPI designated thread(s) have reconstructed the MPI state, the debugger is notified that it is safe to attach. Once attached, the debugger walks all threads out of the breakpoint function and resumes debugging operations.

Scenario 3: Debugger Attached While Checkpointing In this scenario, the debugger is attached when a checkpoint is requested of the MPI process. A transition from the lower-left to lower-right quadrants in Figure 2. This scenario is useful when creating checkpoints while debugging that can be returned to in later iterations of debugging cycle or to provide backstepping functionality while debugging. The debugger will notice the call to the detach function and call the waiting function in all threads in the MPI process before detaching. The MPI designated checkpoint thread(s) are allowed to continue through this function in order to request the checkpoint from the CRS while all other threads wait there for later release. After the checkpoint or on restart the protocol proceeds as in Scenario 2.

Scenario 4: Debugger Detached on Restart In this scenario, the debugger is attached when the checkpoint is requested of the MPI process, but is not when the MPI process is restarted from the checkpoint. A transition from the lower-left to upper-right quadrants in Figure 2. This scenario is useful when

analyzing the uninhibited behavior of an application, periodically inspecting checkpoints for validation purposes or, possibly, introducing tracing functionality to a running program. The process of creating the checkpoint is the same as in Scenario 3. By inspecting the `MPIR_debug_with_checkpoint` variable, the MPI processes know to let themselves out of the waiting function after the checkpoint and on restart.

5 Implementation

The design described in Section 3 was implemented using GNU's GDB debugger, Allinea's DDT Parallel Debugger, the Open MPI implementation of the MPI standard, and the BLCR CRS. Open MPI implements a fully coordinated C/R protocol [6] so when a checkpoint is requested of one process all processes in the MPI job are also checkpointed. We note again that full coordination is not required by the design, so other techniques can be used at the discretion of the MPI implementation.

5.1 Interlayer Notification Callback Functions

Open MPI uses the Interlayer Notification Callback (INC) functions to coordinate the internal state of the MPI implementation before and after checkpoint operations. After receiving notification of a checkpoint request, Open MPI calls the `INC_checkpoint_prep()` function. This function quiesces the network, and prepares various components for a checkpoint operation [11]. Once the INC is finished it designates a checkpoint thread, calls the debugger detach function, and then requests the checkpoint from the CRS, in this case BLCR.

After the checkpoint is created (called the *continue* state) or when the MPI process is restarted (called the *restart* state), BLCR calls the hook callback function in each thread (See Figure 1). The thread designated by Open MPI (in the `INC_checkpoint_prep()` function) is allowed to exit this function without waiting, while all other threads must wait if the debugger intends on attaching. The designated thread then calls the INC function for either the continue or restart phase depending on if the MPI process is continuing after a checkpoint or restarting from a checkpoint previously saved to stable storage.

If the debugger intends on attaching to the MPI process, then after reconstructing the MPI state, the designated thread notifies the debugger that it is safe to attach by printing the "`MPIR_debug_info)`" message to `stderr` as described in Section 3.2.

5.2 Stack Modification

In Section 3.1, the debugger was required to force all threads to call the waiting function before detaching before a checkpoint in order to preserve the program counter in all threads across a checkpoint operation. We explored two different ways to do this in the GDB debugger. The first required the debugger to force

```
void MPIR_checkpoint_debugger_signal_handler(int num) {
  MPIR_checkpoint_debugger_waitpoint();
}
```

Fig. 3. Open MPI's SIGTSTP signal handler function to support stack modification

the function on the call stack of each thread. In GDB, we used the following command for each thread:
`call MPIR_checkpoint_debugger_waitpoint()`
Unfortunately this became brittle and corrupted the stack in GDB 6.8.

In response to this, we explored an alternative technique based on signals. Open MPI registered a signal callback function (See Figure 3) that calls the `MPIR_checkpoint_debugger_waitpoint()` function. The debugger can then send a designated signal (e.g., `SIGTSTP`) to each thread in the application, and the program will place itself in the waiting function.

Though the signal based technique worked best for GDB, other debuggers may have other techniques at their disposal to achieve this goal.

6 Conclusions

Debugging parallel applications is a time-consuming part of the software development life-cycle. C/R-enabled parallel debugging may be helpful in shortening the time required to debug long-running HPC parallel applications. This paper presented a design specification for the interaction between a parallel debugger, C/R-enabled MPI implementation, and CRS to achieve C/R-enabled parallel debugging for MPI applications. This design focuses on an abstract separation between the parallel debugger and the MPI and CRS implementations to allow for greater generality and flexibility in the design. The separation also enables the design to achieve the *always usable checkpoints* goal.

This design was implemented using GNU's GDB debugger, Allinea's DDT Parallel Debugger, Open MPI, and BLCR. An implementation of this design will be available in the Open MPI v1.5 release series. More information about this design can be found at the link below:
`http://osl.iu.edu/research/ft/crdebug/`

References

1. Message Passing Interface Forum: MPI: A Message Passing Interface. In: Proc. of Supercomputing '93. (1993) 878–883
2. Cownie, J., Gropp, W.: A standard interface for debugger access to message queue information in MPI. In: Proc. of the European PVM/MPI Users' Group Meeting. (1999) 51–58
3. Gottbrath, C.L., Barrett, B., Gropp, B., Lusk, E., Squyres, J.: An interface to support the identification of dynamic MPI 2 processes for scalable parallel debugging. In: Proceedings of the European PVM/MPI Users' Group Meeting. (2006)

4. Elnozahy, E.N.M., Alvisi, L., Wang, Y.M., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Computing Surveys **34**(3) (2002) 375–408
5. Chandy, K.M., Lamport, L.: Distributed snapshots: determining global states of distributed systems. ACM Transactions on Computer Systems **3**(1) (1985) 63–75
6. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: Proceedings of the IEEE International Parallel and Distributed Processing Symposium. (2007)
7. Jung, H., Shin, D., Han, H., Kim, J.W., Yeom, H.Y., Lee, J.: Design and implementation of multiple fault-tolerant MPI over Myrinet (M^3). Proceedings of the ACM/IEEE Supercomputing Conference (2005)
8. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for MPI programs over InfiniBand. International Conference on Parallel Processing (2006) 471–478
9. Bouteiller, A., et al.: MPICH-V project: A multiprotocol automatic fault-tolerant MPI. International Journal of High Performance Computing Applications **20**(3) (2006) 319–333
10. Duell, J., Hargrove, P., Roman, E.: The design and implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory (2002)
11. Hursey, J., Mattox, T.I., Lumsdaine, A.: Interconnect agnostic checkpoint/restart in Open MPI. In: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing. (2009) 49–58
12. Curtis, B.: Fifteen years of psychology in software engineering: Individual differences and cognitive science. In: Proceedings of the International Conference on Software engineering. (1984) 97–106
13. Feldman, S.I., Brown, C.B.: IGOR: A system for program debugging via reversible execution. In: Proceedings of the ACM SIGPLAN/SIGOPS workshop on Parallel and Distributed Debugging. (1988) 112–123
14. Wittie, L.: The Bugnet distributed debugging system. In: Proceedings of the 2nd workshop on Making distributed systems work. (1986) 1–3
15. Bouteiller, A., Bosilca, G., Dongarra, J.: Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. Recent Advances in Parallel Virtual Machine and Message Passing Interface (2007) 297–306
16. Ronsse, M., Bosschere, K.D., de Kergommeaux, J.C.: Execution replay and debugging. In: Proceedings of the Fourth International Workshop on Automated Debugging, Munich, Germany (2000)
17. King, S.T., Dunlap, G.W., Chen, P.M.: Debugging operating systems with time-traveling virtual machines. In: Proceedings of the USENIX Annual Technical Conference. (2005)
18. Pan, D.Z., Linton, M.A.: Supporting reverse execution for parallel programs. In: Proceedings of the ACM SIGPLAN/SIGOPS workshop on Parallel and Distributed Debugging. (1988) 124–129
19. Agrawal, H., DeMillo, R.A., Spafford, E.H.: An execution-backtracking approach to debugging. IEEE Software **8**(3) (1991) 21–26
20. Undo Ltd.: UndoDB - Reversible debugging for Linux (2009)
21. TotalView Technologies: ReplayEngine (2009)
22. Sorin, D.J., Martin, M.M.K., Hill, M.D., Wood, D.A.: SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. SIGARCH Computer Architecture News **30**(2) (2002) 123–134