Extended Batch Sessions and Three-Phase Debugging: Using DMTCP to Enhance the Batch Environment

Rohan Garg*
College of Computer and Information Science
Northeastern University
rohgarg@ccs.neu.edu

Jiajun Cao*
College of Computer and Information Science
Northeastern University
jiajun@ccs.neu.edu

Kapil Arya
Mesosphere, Inc.
kapil@mesosphere.io

Gene Cooperman*†
College of Computer and Information Science
Northeastern University
gene@ccs.neu.edu

Jérôme Vienne
Texas Advanced Computing Center
The U. of Texas at Austin
viennej@tacc.utexas.edu

ABSTRACT

Batch environments are notoriously unfriendly because it's not easy to interactively diagnose the health of a job. A job may be terminated without warning when it reaches the end of an allotted runtime slot, or it may terminate even sooner due to an unsuspected bug that occurs only at large scale.

Two strategies are proposed that take advantage of DMTCP (Distributed MultiThreaded CheckPointing) for system-level checkpointing. First, we describe a three-phase debugging strategy that permits one to interactively debug long-running MPI applications that were developed for non-interactive batch environments. Second, we review how to use the SLURM resource manager capability to easily implement extended batch sessions that overcome the typical limitation of 24 hours maximum for a single batch job on large HPC resources. We argue for greater use of this lesser known capability, as a means to remove the necessity for the application-specific checkpointing found in many long-running jobs.

CCS Concepts

•Software and its engineering → Checkpoint / restart; Software testing and debugging;

1. INTRODUCTION

*This work was partially supported by the National Science Foundation under Grant ACI-1440788.
†This work was partially supported by the IDEX "Chaire d'attractivité" program of the Université Fédérale Toulouse Midi-Pyrénées under Grant 2014-345.

XSEDE16, July 17 - 21, 2016, Miami, Florida

ISBN 978-1-4503-4755-6/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2949550.2949645

Checkpointing is typically used in high-performance computing for fault tolerance. If a computation fails, whether for reasons of hardware failure or temporary software failure, then the user restarts the computation from a previous checkpoint. Two classic software packages used for this purpose in High-Performance Computing (HPC) are: DMTCP [2, 7] and BLCR [10, 14].

Going beyond fault tolerance, we describe two checkpointing scenarios particularly targeted toward batch environments:

- extended batch sessions for long-running jobs; and
- three-phase debugging for interactively debugging long-running MPI batch jobs.

1.1 Extended batch sessions for long-running jobs

In the first scenario, we consider the use of checkpointing to save the state of a running computation near the expiration of its runtime allotment in a batch system (for example, using [23]). This provides a type of insurance for the user. If a job terminates well in advance of the runtime allotment, then there is no checkpoint and thus, there is no cost.

In this scenario, we review how elements from resource managers and a checkpointing package can be used to handle long-running programs. Hooks for this are already available using SLURM [23], and can be used with any checkpointing package. TORQUE™ 3.0 has a related capability, by invoking a checkpoint during pbs_mom shutdown. However, the TORQUE facility seems to be closely tied to BLCR [10, 14].

While this scenario is not novel in itself, it seems not to be widely known. It is also noted that this technique can be used even when the resource manager does not directly support checkpointing: Given that the duration of a batch time slot is known in advance, it is easy to invoke a timer in a helper function or program that invokes a checkpoint just prior to job termination. It is argued that this system-level checkpointing solution has particular importance in relieving application developers from the need to include a special routine for saving the state of an application. While many large software systems include routines for saving state, this

methodology is often considered error-prone. The issue is that a new feature is added to the software, and a separate routine for saving the state must be correspondingly updated. Thus, the maintainer of code for saving state must also have some knowledge of the data structures throughout the rest of the code.

The importance of checkpointing near the expiration of the runtime allotment of a batch system offers a special opportunity when considered in combination with the potential for resource managers to provide system-generated predictions of runtime duration. As described in [27], users typically over-estimate their runtime requirements in order to avoid having their job killed before completion. Those authors argue that system-generated runtime predictions offer a more accurate assessment, and can then offer additional sources of backfill for a backfill-based queuing algorithm. As described above, system-level checkpointing removes the penalty associated with a wrong system-generated runtime prediction.

1.2 Three-phase debugging

Next, we consider the second of the two scenarios. The second scenario is especially important for application developers, as opposed to end users of an application. This scenario considers interactive debugging of MPI jobs running in a non-interactive batch environment. Once a code base has been developed, there are three important aspects of maintaining and extending that code.

Debugging: To make sure the code runs and yields correct results.

Profiling: To analyze the code to identify performance bottlenecks.

Optimization: To make the code run faster and/or consume fewer resources.

This work makes a particular contribution to debugging. It allows one to checkpoint a long-running job just prior to a software problem, whether that problem is a crash, software "hanging", premature termination, incorrect intermediate values, or any other condition of interest. After that, one can repeatedly restart, attach a debugger, and interactively inspect the causes of the problem. This contributes to the debugging process in four ways.

Reproducibility: To find and freeze a scenario where the software problem is reproducible.

Reduction: To reduce that problem to its essence.

Deduction: For developing hypotheses on what the cause of the problem might be.

Experimentation: To filter out invalid hypotheses.

A three-phase debugging methodology is described that allows one to checkpoint within seconds to minutes prior to a software problem, as desired.

This leaves open the question of how many MPI processes can be restarted on a single "debug node". Even though a typical cluster will provide users with interactive computers for code development and debugging, those computers are often diskless. Hence, they have no swapfile and can only accommodate as many processes as fit in RAM. Even where a swapfile exists, a user will soon experience thrashing as many restarted MPI processes begin to page.

For this reason, a variant of three-phase debugging is also described in which only a single MPI process is restarted. This work represents the first demonstration of restarting just one MPI process from within a widely used checkpointing package. The strengths and weaknesses of this approach are also examined.

When running short MPI applications, a competing approach is to log all MPI messages and events for later replay [5, 6, 13, 18]. However, this runs into several problems when considered for long-running jobs. First, it assumes that the application runs deterministically. This is usually not the case even for multi-threaded software within a single process, and deterministic record-replay continues to be an active area of research [3, 11]. Second, record-replay is especially inefficient for long-running jobs due to the storage requirements and performance overhead of logging many MPI messages, with each message potentially containing large data, for later replay.

1.3 DMTCP

In this work, DMTCP (Distributed MultiThreaded CheckPointing) is used as a vehicle for transparent, system-level checkpointing. System-level checkpointing requires no cooperation from the application. None of the target application, library, and operating system are modified. In particular, system-level checkpointing is always transparent to the application.

1.4 Organization of paper

Section 2 discusses the use of long-running batch sessions and its importance in combination with system-generated runtime predictions. Section 3 describes the three-phase debugging strategy for interactive debugging of long-running batch jobs. Section 4 presents an experimental evaluation. Section 5 presents related work. Finally, a conclusion and discussion of future work appear in Section 6.

2. EXTENDED BATCH SESSIONS

At most HPC sites, the maximum runtime duration of a batch job is 24 hours or less and this is a problem for long-running jobs. We overcome this limit through a system-initiated checkpoint. For example, a batch reservation for 24 hours might include a system-initiated checkpoint near termination — perhaps at the beginning of hour 23. This costs nothing if the job completes before hour 23. Hence, such a policy could be safely adopted for all batch jobs in a queue. As an example, the SLURM strigger command [23] provides a simple mechanism to implement just such a facility. The effectiveness of such an approach depends on questions such as the time to checkpoint. We present a small experimental study in Section 4.2 that shows that checkpointing can generally be achieved in less than a minute — even for the HPCG benchmark [9] stressing both dense and sparse linear algebra. This benchmark was tested with up to 1024 MPI processes, using 1.2 GB per process (1.2 TB total).

The current ideas complement previous work on system-generated prediction of runtime duration [27]. In that work, the authors argue that users are notorious for over-estimating the runtime in order to avoid premature termination. System-generated runtime predictions are shown there to improve

overall performance. By combining this earlier work with system-initiated checkpoints prior to the scheduled termination, one achieves the best of both worlds: even if a system-generated runtime prediction is too short, no work is lost.

Note also that system-generated runtime prediction not only improves the throughput of a batch system, but it also provides benefits for the end user. When a user's job is chosen for backfill, the user's job completes much sooner than otherwise. Hence, fast, system-level checkpointing removes the risk of killing a user's job and often advances the time by which the job completes.

3. THREE-PHASE DEBUGGING FOR LARGE MPI COMPUTATIONS

Debugging software for large MPI computations running in the batch is often difficult and time-consuming — especially if a software bug occurs beyond the first few minutes of a computation. Typically, this requires several cycles in which print statements are added and an additional batch job is run.

Ideally, one would run under an interactive debugger such as GDB, and interactively capture a process that is about to crash or exit, and begin interactive debugging. But maintaining a reservation of a large (many-node) HPC computation in order to debug interactively is an expensive proposition. Many nodes sit idle while the developer interactively probes the state of the computation.

Instead, a three-phase debugging strategy is proposed. It is assumed that the software either crashes or prematurely exits. The proposed strategy also covers the case in which software "hangs" (for example, due to a mismatch of sends and receives in MPI). Hanging can automatically be detected by a user by adding timeouts around the MPI send and receive calls, and then exiting in the event of a timeout. (Timeouts are easy to implement, through either of two mechanisms: MPI_Isend and MPI_Irecv; or the alarm() system call.)

The next subsection describes the basic three-phase debugging strategy, while the later sections discuss important variations in the case that a single "debug" computer node does not have sufficient RAM to prevent thrashing.

3.1 Restarting with all MPI processes

Figure 1 provides an overview of the key steps for three-phase debugging. In phase 1, a batch job is submitted and frequent, periodic checkpoints are taken — perhaps once every hour. When the software crashes or exits, the bug must occur between the last checkpoint and termination. In phase 2, one restarts from the last checkpoint and takes finer-grained (more frequent) checkpoints, such as every minute. Now the bug is manifested within a single minute of parallel runtime. Finally, in phase 3, all of the checkpoint images are copied to a "debug node", and then restarted on that single debug node. A debugger can be attached on restart using standard techniques such as the attach capability of GDB.

3.2 Restarting with a single MPI process

A variation of this strategy is also proposed, in which only one of the MPI processes is restarted on the debug node in phase 3. Such a variation is needed if the swapfile or physical RAM are not large enough to accommodate efficiently

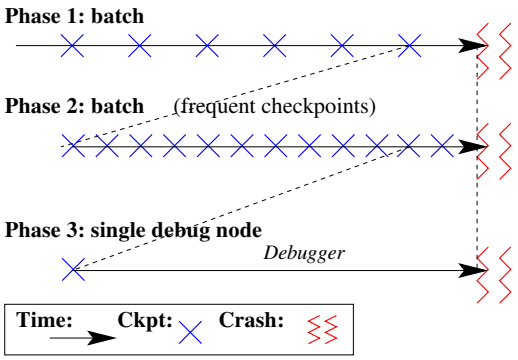


Figure 1: Three-phase debugging: The bug always lies between the last checkpoint and the crash, early termination, or hanging of the process. In phase 3, a debugger is attached after restart.

running many MPI processes on a single node. However, an important caveat for this alternative is that after phase 1, the user must be able to predict which single MPI process will need to be restarted in phase 3 under an interactive debugger.

In this variation, one identifies a single MPI process at the end of phase 1, where the software problem had occurred. In this case, one creates a wrapper function around each of the MPI communication functions. For this exposition, we assume that MPI_Recv was the last MPI library call requiring inter-process communication. Note that DMTCP makes such wrapper functions easy to write through the DMTCP plugin mechanism [8]. Then, in phase 2, one restarts from the last checkpoint of phase 1, but one issues a checkpoint after every MPI receive function in the target process.

In this variation, phase 3 consists of restarting (on a single debug computer), only the single process that will incur the software problem. It is guaranteed that the user code will not make additional calls to MPI, since we took a checkpoint after every invocation of an MPI receive during phase 2. In phase 3, we are restarting from the last checkpoint.

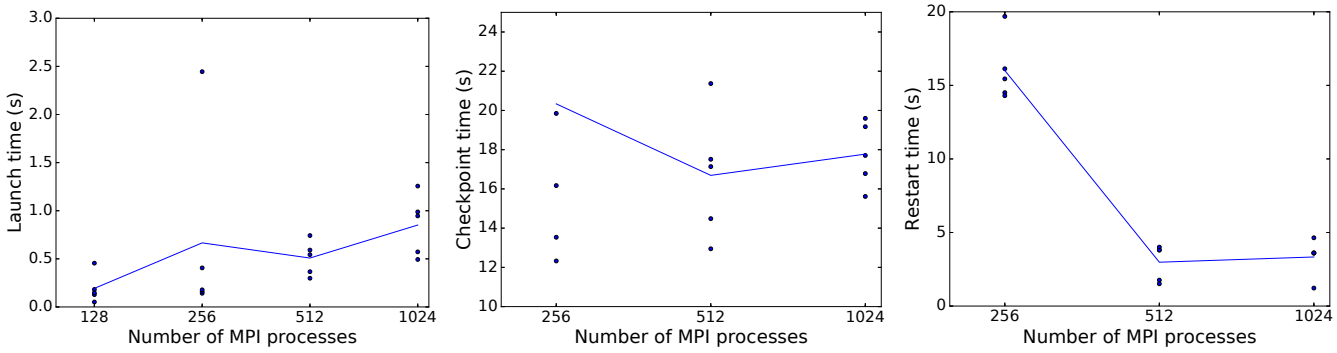
Identifying the faulty MPI process.

Isolating a single MPI process out of potentially thousands of processes is an extremely difficult task and we do not claim to provide a mechanism to predict the faulty process. However, one can use heuristics such as observing an earlier run to see which MPI process exited first.

Note that while this technique is not fully automatic or complete, we propose it as an aid that complements the normal debugging process of the end user. For example, identifying a process only isolates the actual crash (error), whereas the original cause of the bug (fault) may have started with a memory leak, race condition, etc., on a different MPI process. As part of the typical debugging process, one can then approximate the cause of the crash, develop a theory of the original cause, and then execute an additional run with additional asserts or print statements, in order to try to confirm the theory of the original cause.

3.3 Communication-intensive applications

There is a further concern that communication-intensive applications may make frequent calls to MPI. In this case, the overhead of many checkpoints would become excessive.



(a) Launch Overhead (b) Checkpoint Overhead (c) Restart Overhead

Figure 2: Various overheads for NAS Benchmark LU, Class E, when running with DMTCP. Each case was run five times, and the graphed line runs through the average value of each case.

A simple strategy to escape this problem is to include a counter either in the application code or in the wrapper functions around calls to MPI.

In this scheme, in phase 2a, one might choose to checkpoint only when the counter is a multiple of 1,000. A further batch phase 2b would then begin by restarting from the last checkpoint in phase 2a. In phase 2b, one would checkpoint only when the counter is a multiple of 100. Similarly, phase 2c would invoke a checkpoint only when the counter is a multiple of 10. Finally, phase 2d could then follow the original strategy of a checkpoint after each MPI call, while knowing that the total running time would be based on at most ten iterations of the counter.

Finally, an additional concern exists over bugs or software problems that only manifest some of the time. Here, we make the common assumption that if a bug appeared once, then repeating the job enough times will eventually produce the bug. The guiding principle is that we only restart from a given checkpoint if the bug had previously been observed to occur during the original timeline in which the checkpoint was created. Thus, the common assumption implies here that repeating a restart from the checkpoint enough times will eventually produce the bug again. Since phases 2 and 3 are always short compared to phase 1, the expense of possibly requiring multiple restarts is minimal.

4. EXPERIMENTAL EVALUATION

All experiments were conducted on Stampede [24] at TACC (Texas Advanced Computing Center). Stampede is currently the #10 supercomputer on the Top500 list [25]. Each computer node at Stampede has 16 cores, consisting of a dual-CPU Xeon ES-2680 configuration with 32 GB of RAM. Stampede uses the SLURM resource manager for allocation of nodes, but it uses an ibrun or mpirun_rsh command for launching MPI.

4.1 Checkpoint performance

We first demonstrate the performance of DMTCP: launch time, checkpoint time and restart time are studied for different scales.

Figure 2a shows that the overhead to launch all MPI processes under DMTCP is negligible (less than a second in most cases). The non-zero overhead is to establish a connection between an MPI process and the central checkpointing coordinator of DMTCP.

Figure 2b and Figure 3a demonstrate the checkpoint performance for the NAS LU benchmark (class E), and the HPCG benchmark. The main difference between these two benchmarks regarding the checkpoint performance is the checkpoint image size. In the case of HPCG, the image size for each process is constant (1.2 GB) as the number of processes varies, while for LU, the total memory footprint over all nodes is roughly constant as the number of processes varies. For LU, this implies a decreasing trend for the checkpoint image size per process. However, in both cases, we observe no significant change in the time to checkpoint as the number of processes varies. We speculate that this is because the throughput of Lustre is not near capacity, and so the bandwidth to stable storage continues to grow as the total checkpoint image size grows.

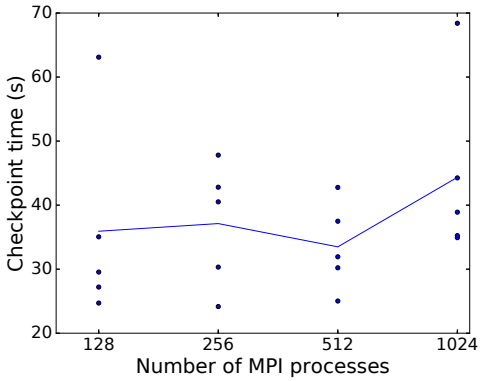
Figure 2c shows that the restart time for the NAS LU benchmark (class E) decreases by a factor of four from 256 processes to 512 processes, but does not change much from 512 to 1024. We believe this is due to the buffering mechanism and parallel nature of the Lustre filesystem. In the case of 256 processes, the checkpoint size per process is much larger, approximately 1 GB, compared to 600 MB for 512 processes, and 425 MB for 1024 processes. As a result, for 256 processes, it takes more time for each individual process to read its checkpoint image from disk and restart.

4.2 Extended batch sessions

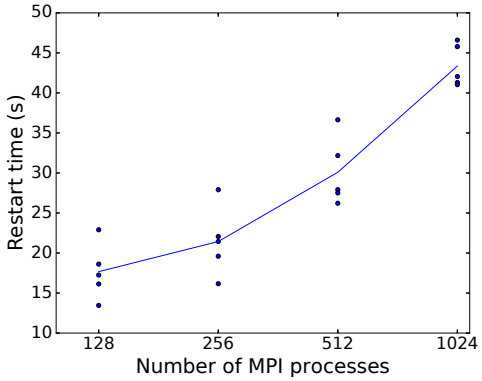
This section asks the question of how close to the end of the allotted time for an allocation one can checkpoint. If one can checkpoint close to the end, many computations will finish in time and never need a checkpoint. Otherwise, one will incur checkpoints that are not required. Section 2 summarized these issues.

In extending an MPI application to run beyond its runtime allocation, it is necessary to checkpoint it prior to the expiration of its allocation. The SLURM environment allows a signal to be sent to the process at a fixed time before the duration of an allocation elapses [23]. It is also easy to create a small utility process running alongside MPI with the same functionality, which sends a signal to trigger checkpointing of an application.

Figure 3a shows the average time to checkpoint in the case of HPCG to be approximately constant as the number



(a) Checkpoint overhead



(b) Restart overhead

Figure 3: Checkpoint and restart overheads for the HPCG benchmark.

of MPI processes increases. We speculate that the near constant time reflects the ability of the Lustre filesystem to split a large write into many small buffers that are scheduled for writing in parallel. The large variance is likely due to the interference of other users' jobs running at the same time.

Figure 3b shows that the time to restart in the case of HPCG increases monotonically with the number of processes. Recall that for HPCG the memory usage per process stays approximately constant across different scales. Consequently, the restart time is dominated by the time to read the checkpoint images. The parallel nature of the Lustre filesystem prevents the performance from degrading exponentially. We speculate that the increasing restart time occurs due to the well-known fact that the competing reads from many simultaneous processes create competing disk seeks within the hard disk [20, Section 25.5].

In order to simulate this scenario, we test the checkpointing performance as the frequency of periodic checkpoints is increased.

4.3 Debugging large MPI computations: Parallel restart case

This section asks two questions. First, how many MPI processes may be run, while ensuring feasibility of a restart on a single development node. The issue is that if the sum of the working sets of each MPI process is more than the size of RAM on the development node, then there may be a large slowdown due to excessive paging (thrashing) of the virtual

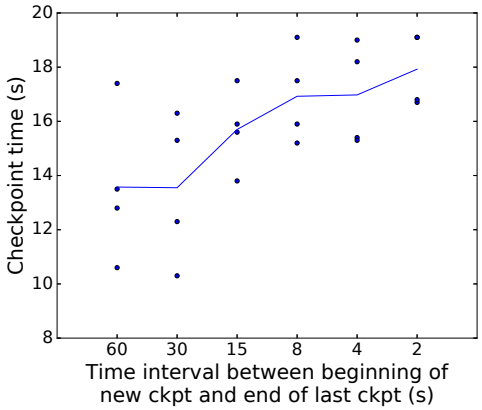


Figure 4: Checkpointing time versus Checkpointing frequency for LU.E.256. Each checkpoint image is 950 MB on average (total 250 GB).

memory subsystem. To answer this question, we measure the slowdown on the development node. We restart the same computation both on the original parallel nodes and with overcommitment on fewer nodes. We report the time to complete the computation for different levels of overcommitment for the LU benchmark (class D) of NAS [21].

Analysis of Phase 2.

The second question addressed here is how close to the software bug can one generate a checkpoint in batch (during phase 2). If one could generate a checkpoint just one second before the software bug, then one could tolerate the delay due to a lot of thrashing on the development node, and still interactively proceed with debugging. Interactively debugging close to the occurrence of the bug is always preferred over a more traditional core dump.

Three-phase debugging requires very frequent debugging to minimize the time between the last checkpoint and the final software failure. Figure 4 demonstrates the feasibility of checkpointing within two seconds of resuming from a previous checkpoint. The checkpoint time is dominated by the time taken by Lustre to write the checkpoint images. The time taken to save the state of the process in memory and its associated kernel state is negligible. This is critical for enabling generation of checkpoint images of an application close to a software bug.

In Figure 4, the effect of invoking a new checkpoint soon after the end of the previous checkpoint is evaluated in the case of the NAS LU (class E) benchmark running on 256 MPI processes. We observe that the checkpoint time stays approximately constant when a new checkpoint is invoked after at least 30 seconds after a previous checkpoint. Although the checkpointing time on average increases by approximately 28 % between an interval of 60 seconds and 2 seconds, it stays below 20 seconds for all cases.

The increase in checkpoint time could be attributed to the increased congestion on the filesystem and the back-end network used by Lustre.

From Figure 4, we note that even at an interval between checkpoints of 2 seconds, the average time for the checkpoint itself is less than 18 seconds. Thus, Phase 2 of Figure 1 could checkpoint at intervals even of two seconds — albeit

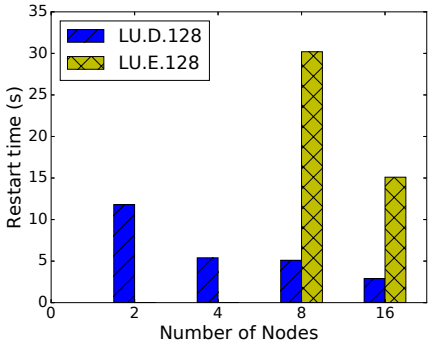


Figure 5: Restart time with overcommitment for LU.D.128 and LU.E.128. Each checkpoint image for LU.D.128 is 210 MB on average. Each checkpoint image for LU.E.128 is 1.5 GB on average.

at a ten-fold slowdown (2 seconds of execution followed by 18 seconds of checkpointing). This is an acceptable trade-off in the shorter Phase 2, since it results in a maximum interval of 2 seconds in which to analyze the program under a debugger in Phase 3.

Analysis of Phase 3.

Figure 5 shows the effect of overcommitment on the total restart time for the NAS LU (classes D and E) benchmark. The size of a checkpoint image in the case of class D is 210 MB. The size of a checkpoint image in the case of class E is 1.5 GB. In each case, the MPI ranks are distributed evenly among the available nodes. The time to restart the application increases with the increasing levels of overcommitment because Lustre optimizes the reads that are issued in parallel from many nodes.

The compute nodes on Stampede are configured with no swapfile, and hence, restarting doesn't work for the cases where the required RAM per node nears or exceeds the available limit on Stampede (32 GB), namely, LU.E.256 over less than 16 nodes, LU.E.128 over less than 8 nodes. In the case of LU.E.256, each MPI rank uses 957 MB of RAM on average, and for LU.E.128, each MPI rank uses 1.5 GB of RAM on average.

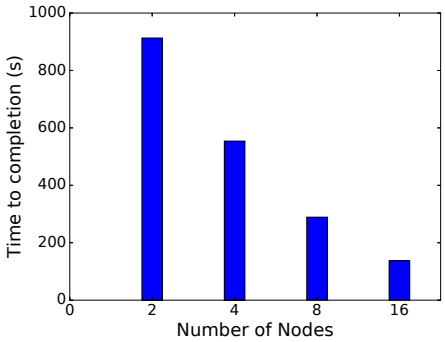


Figure 6: Time to completion for LU.D.128 when restarting with overcommitment.

Figure 6 shows the performance overhead incurred when restarting the NAS LU benchmark (class D) with overcommitment. In each case, the MPI ranks are distributed evenly among the available nodes. The same set of checkpoint images was used in all cases. We observe an exponential slowdown with increasing levels of overcommitment (or decreasing number of nodes).

4.4 Debugging large MPI computations: Single restart case

Section 4.3 demonstrated the issue with debugging of batch jobs when the number of MPI processes is too large, and the virtual memory subsystem is thrashing when running under a single development node. Naturally, one limited solution is to use more than one development node. If the MPI processes can be distributed among two or more development nodes, then the sum of the working sets on a single node is reduced by half or more.

In this section, we present a solution that fully scales, as described in Section 3. During phase 1, some conditions were identified to determine which MPI process will crash. During phase 2, checkpoints were taken after every MPI receive on the MPI process of interest. Thus, in phase 3, we restart on a single development node, but we restart only the process that will experience the software crash. That process can then be run under a debugger, and we are guaranteed that there will be no further MPI receives. In the example shown here, the resulting computation for this MPI process is then deterministic. There is no issue with thrashing, since we are restarting only a single MPI process on the development node.

We extended DMTCP to support restarting of a single process without restarting the entire distributed computation. The DMTCP socket plugin is responsible for checkpointing and restoring the state of socket connections between the computation processes. However, when restarting a single process, there are no socket peers and thus, socket connections cannot be properly restored which results in a failure. We changed the socket plugin to skip restoration of the socket connections when indicated by an environment variable. During the normal, full-computation restarts, the environment variable is not set and thus sockets are restored, whereas, during single-process restarts, the environment variable is set to avoid restoration of any sockets. The DMTCP InfiniBand plugin was similarly modified to skip restoration of InfiniBand connections.

By ensuring that the application gets checkpointed after the last MPI send, receive, or other calls, we avoid the issue of the restart process trying to communicate with its peers. Hence on restart, can attach an interactive debugging session by using a debugger such as GDB.

Just one experiment was executed, for the purpose of demonstrating the feasibility of this methodology. The case chosen was restarting a single process from an LU.E.256 computation. This corresponds to Figure 2c, in which 256 processes were checkpointed. However, only one process is restarted.

As expected, the restart time is much faster in this case. We observed a restart time of two seconds, as compared to the original 15 seconds of Figure 2c to restart all 256 MPI processes.

5. RELATED WORK

Two checkpointing packages are currently widely used in support of MPI: DMTCP [2,7] and BLCR [10,14]. Both are

free and open source. DMTCP (Distributed MultiThreaded CheckPointing) was chosen for this work because it is MPI-agnostic. While the current experiments were performed for MVAPICH2, the same code should also support other MPI implementations.

Extending checkpoint-restart to support the InfiniBand network was particularly difficult. Cao et al. provided direct support for InfiniBand [7], and thus maintained the MPI-agnostic character of DMTCP. In contrast, earlier implementations of transparent distributed checkpointing over InfiniBand relied on a custom checkpoint-restart service from each MPI implementation, in combination with BLCR [14]. Several MPI dialects implemented a checkpoint-restart service to disconnect from the network prior to checkpoint, and re-connect after restart [4, 12, 16, 17, 19, 22].

Hursey et al. [15] discussed creating intermediate checkpoints, so as to facilitate going back to earlier points in time in order to analyze a bug. This is similar to phase 1 of our three-phase debugging scenario, except that we also assume that a bug manifests in a crash, early termination, or a hanging process. Options to detach a debugger on checkpoint and to re-attach on restart are discussed.

In this work, a variation of three-phase debugging proposes to checkpoint after every call to MPI send, receive, or other calls to the MPI library on a process that is known to manifest a bug. This allows one to "replay" after restart without concern for communication with other MPI processes. An alternative approach for phase 3 would be to use message or event logging [5, 6, 13, 18]. However, as discussed at the end of Section 1.2, this is not practical for long-running applications.

Tsafrir et al. argue for system-generated runtime predictions rather than user runtime estimates [27]. In that work, their argument is motivated by the need for small jobs to insert during backfill. Most batch systems today employ a backfill algorithm in which larger jobs are given priority, but shorter jobs are scheduled when it can be shown that some nodes will be available, but only for a shorter duration. The authors argue that users habitually over-estimate the runtime of their jobs because they will not tolerate jobs being killed due to the expiration of their scheduled runtime. By more aggressively predicting runtimes, the authors' system can "discover" additional small jobs to be used for backfilling that would not normally be available. The downside of this approach is that some user jobs will be terminated early if the runtime prediction is too aggressive.

In this work, we argue for a combination of this idea with system-initiated checkpoints prior to termination. Normally, one prefers not to employ checkpointing for short jobs, due to the high relative overhead. But if the resource manager has predicted that a user job will complete early, then the overhead of checkpointing will not occur, except in the infrequent case of an inaccurate prediction. Even in this infrequent case, no valuable work was lost. And in the frequent case, the user is often rewarded by having his or her job complete early if it can be executed within a backfill slot.

Finally, we consider the existing state-of-the-art approaches to debugging MPI applications. The two most common tools are TotalView [26] and Allinea DDT [1]. DDT (and TotalView) are particularly difficult to use in the case of a long-running computation in which the bug does not occur early. Contrary to the checkpointing approach described here, once one reaches the bug, one needs to restart the application from the beginning and redo an analysis. Further, the analysis requires running the application at full scale. This leads to long waits while the job is in the queue or while it is running and has not yet reached a breakpoint. Finally, DDT is also limited to only a few languages, as compared to the debugging approach with DMTCP. In the approach advocated here, DMTCP operates entirely at the binary level, and a generic debugger such as GDB can be used to attach after restart.

Nevertheless, the approach promoted by DDT has a long history and is undoubtedly easier to use for beginners. The approach advocated here requires some familiarity with ideas like the attach mode of GDB, writing new DMTCP plugins for wrappers around MPI functions, and small modifications to DMTCP in the case of restarting a single process.

6. CONCLUSION AND FUTURE WORK

While system-level checkpointing is more often used for fault tolerance, we have shown its applicability and importance in two additional areas.

First, it is important for extended batch sessions: batch jobs that threaten to overrun the allotted time for their reservation. Figure 3a shows that even for large memory-intensive jobs, 1024 MPI processes can be checkpointed in less than 45 seconds.

Second, is novel three-phase debugging strategy was introduced for debugging long-running MPI applications, and where it is not practical to repeatedly restart the application from the beginning during successive debug cycles. In order to avoid the long waits from running batch jobs, only the first phase requires the traditional large resources of a conventional batch job. The second phase consists of a much shorter batch job over the final interval between checkpoints. For every two seconds of execution, 18 seconds is spent in checkpointing (see Figure 4). The overhead of these frequent checkpoints is considered acceptable for the shorter Phase 2. In the third phase, all MPI processes are restarted on a single "debug node", and a traditional debugger such as GDB is attached.

In cases where it is not possible to restart all processes on a single debug node, a variation was demonstrated in which the application developer need only restart one of the MPI processes. Admittedly, this requires the developer to use prior information (e.g., from intermediate printouts or prior debugging runs) in order to guess which process needs to be run under the debugger next. If the developer needs several trials to discover the MPI process that causes the bug or other software problem, he or she can re-run phase 2, while choosing a different MPI process to debug.

In future work, one can explore a variation in which some, but not all MPI processes are restarted as part of phase 3.

7. REFERENCES

[1] Allinea DDT: The Debugger for C, C++ and F90 Threaded and Parallel Code. http://www.allinea.com/, accessed Apr., 2016.

[2] J. Ansel, K. Arya, and G. Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS), pages 1–12. IEEE Press, 2009.

[3] A. Basu, J. Bobba, and M. D. Hill. Karma: Scalable Deterministic Record-Replay. In Proceedings of the

International Conference on Supercomputing, pages 359–368. ACM, 2011.

[4] A. Bouteiler, T. Herault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V project: a Multiprotocol Automatic Fault Tolerant MPI. International Journal of High Performance Computing Applications, 20:319–333, 2006.

[5] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 297–306. Springer, 2007.

[6] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the Message Logging Model for High Performance. Concurrency and Computation: Practice and Experience, 22(16):2196–2211, 2010.

[7] J. Cao, G. Kerr, K. Arya, and G. Cooperman. Transparent Checkpoint-Restart over InfiniBand. In Proc. of the 23rd Int. Symp. on High-performance Parallel and Distributed Computing, pages 13–24. ACM Press, 2014.

[8] DMTCP team. dmtcp/plugin-tutorial.pdf. http://github.com/dmtcp/dmtcp/blob/master/doc/plugin-tutorial.pdf, accessed Apr., 2016.

[9] J. Dongarra, M. Heroux, and P. Lusczek. HPCG Benchmark. http://hpcg-benchmark.org/, HPCG 3.0, 2015.

[10] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart (BLCR). Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2003.

[11] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen. Execution Replay of Multiprocessor Virtual Machines. In Proc. of 4th ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments, VEE '08, pages 121–130. ACM, 2008.

[12] Q. Gao, W. Yu, W. Huang, and D. K. Panda. Application-Transparent Checkpoint/Restart for MPI Programs over InfiniBand. In ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[13] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A Toolkit for Distributed System Performance Analysis. In 8th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2000. Proceedings, pages 267–273. IEEE, 2000.

[14] P. Hargrove and J. Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. Journal of Physics Conference Series, 46:494–499, Sept. 2006.

[15] J. Hursey, C. January, M. O'Connor, P. H. Hargrove, D. Lecomber, J. M. Squyres, and A. Lumsdaine. Checkpoint/Restart-enabled Parallel Debugging. In Recent Advances in the Message Passing Interface, pages 219–228. Springer, 2010.

[16] J. Hursey, T. I. Mattox, and A. Lumsdaine. Interconnect Agnostic Checkpoint/Restart in Open MPI. In Proc. of the 18th ACM int. Symp. on High Performance Distributed Computing, pages 49–58. ACM, 2009.

[17] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine. The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. In Proc. of the 21^{st} IEEE Int. Symp. on Parallel and Distributed Processing (IPDPS) / 12^{th} IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems. IEEE Computer Society, March 2007.

[18] D. B. Johnson and W. Zwaenepoel. Sender-based Message Logging. Rice University, Department of Computer Science, 1987.

[19] J. Liu, J. Wu, and D. K. Panda. High Performance RDMA-based MPI Implementation over InfiniBand. Int. J. Parallel Programming, 32(3):167–198, June 2004.

[20] Lustre Team. Lustre file system: Chapter 25: Lustre tuning, accessed June, 2016. http://wiki.old.lustre.org/manual/LustreManual20_HTML/LustreTuning.html#50438272_45406.

[21] NASA Advanced Supercomputing Division. NAS parallel benchmarks. http://www.nas.nasa.gov/publications/npb.html, accessed Apr., 2016.

[22] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. International Journal of High Performance Computing Applications, 19(4):479–493, 2005.

[23] Slurm: strigger. http://slurm.schedmd.com/strigger.html, accessed June, 2016.

[24] TACC Stampede user guide - TACC user portal. https://portal.tacc.utexas.edu/user-guides/stampede, accessed Apr., 2016.

[25] TOP500 supercomputer sites. http://top500.org/lists/2015/11/, Nov. 2015.

[26] TotalView team. TotalView debugger. http://www.roguewave.com/products/totalview-family/totalview.aspx, accessed Apr., 2016.

[27] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling using System-Generated Predictions Rather than User Runtime Estimates. Parallel and Distributed Systems, IEEE Trans. on, 18(6):789–803, 2007.