Transparent Adaptive Library-Based Checkpointing for Master-Worker Style Parallelism

Gene Cooperman* Jason Ansel Xiaoqin Ma
College of Computer and Information Science
Northeastern University
Boston, MA 02115
{gene,jansel,xqma}@ccs.neu.edu

Abstract

We present a transparent, system-level checkpointing solution for master-worker parallelism that automatically adapts, upon restart, to the number of processor nodes available. This is important, since nodes in a cluster fail. It also allows one to adapt to using multiple cluster partitions and multiple resources from the Computational Grid, as they become available. Checkpointing a master-worker computation has the additional advantage of needing to checkpoint only the master process. This is both fast and more economical of disk space. This has been demonstrated by checkpointing Geant4, a million line C++ program. Our solution has been implemented in the context of TOP-C (Task Oriented Parallel C/C++), a free, open-source parallel package, although it can easily be ported to additional master-worker packages.

1 Introduction

There is now a rich literature on checkpointing techniques for parallel computation on a cluster [1, 7, 25, 26, 28, 34, 35]. Nevertheless, a thorny issue remains. How does one checkpoint and restart a parallel computation, if the number of processor nodes currently available is different from the original number of nodes? This issue may occur, for example, if half of a cluster (perhaps the network hub for a rack) fails. It may happen if a shared cluster facility becomes more heavily used, and so fewer idle nodes are available. In Grid computations, this situation occurs as resource availability changes during the weekend or overnight. Conversely, if even more nodes become available (perhaps on the Grid or on a cluster), one wishes to checkpoint a computation and then restart it to employ the additional nodes.

We define *adaptive checkpointing* as the ability to checkpoint a parallel computation on some number of nodes, and then efficiently restart the computation on a different number of nodes. Our solution is a *system-level* one. The application writer does not declare what data structures to checkpoint. Furthermore, the solution is *transparent*. The application writer need not add code to request a checkpoint at appropriate locations. The system-level strategy avoids the labor-intensive and error-prone work of explicitly checkpointing the many data structures of a large program.

The master-worker style of parallelism is especially appropriate for adapting a computation to the number of nodes available. This has been the basis of most meta-computing packages [6, 19, 31, 32], which allow a master or controlling process to dynamically spawn additional processes on newly available worker nodes. Meta-computing also is tolerant of worker process failures.

Although meta-computing solutions would be attractive for adaptive checkpointing, it has one fatal flaw. It is not tolerant of failure by the master process.

> *Hence, the challenge of adaptive checkpointing for master-worker parallelism is to checkpoint the state of the master, along with sufficient additional state information, so as to either restart the worker processes or spawn new ones, while not losing any ongoing computational tasks on the worker processes.*

Adaptive checkpointing has the important advantage of needing to checkpoint only one process: the master process. This makes checkpointing fast and economical of disk space. In experiments, we checkpoint and restart the master process in seconds. (For an image up to 200 MB, we checkpoint in less than 1 second.) Upon restart, any interrupted worker tasks will be restarted from the beginning.

While there is a common perception that master-worker style parallelism is limited to "embarrassingly trivial" paral-

*This work was partially supported by the National Science Foundation under Grants CCR-0204113 and ACIR-0342555, and by the Institute for Complex Scientific Software (ICSS, http://www.icss.neu.edu/).

Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)
0-7695-2585-7/06 $20.00 © 2006 IEEE

lel programs, this is not the case. TOP-C has been designed as a master-worker paradigm able to handle very general models of parallelism. It allows TOP-C application writers to easily code parallel strategies based, for example, on optimistic concurrency, dataflow diagrams, and other parallel models. The same principles apply to master-worker parallelism in any architecture (for example, MPI), with some additional coding efforts.

Our implementation on top of TOP-C need only checkpoint the master process and not the worker processes. This is because TOP-C ensures that although the master may, for example, modify global variables, worker processes will mirror such changes. So, upon restarting from a checkpoint, new worker processes can be created based on the same checkpoint file as was used for the master process.

Implementation. Our solution is *library-based*. It does not require any new kernel module or kernel modification. We write wrappers around those system functions for which we wish to detect information sent to and returned from the kernel. The wrappers preserve the original behavior through `dlopen/dlsym` in the case of system library calls, and through `syscall` in the case of system calls to the kernel. The use of `syscall` was pioneered by Condor [30]. See Section 4.4 for further discussion.

Section 2 provides a brief summary of the TOP-C parallel model, and now non-trivial parallelism is introduced into a master-worker parallel style. Section 3 describes the Geant4 application. Section 4 describes the higher level issues of bringing the TOP-C master process into a consistent state for checkpointing. Section 5 describes how fault tolerance is implemented for the case when worker processes that fail. Section 6 presents experiments showing that the master process can be checkpointed and restarted in less than a second. Finally, Section 7 presents related work on checkpointing.

2 Brief Overview of TOP-C: Non-Trivial Master-Worker Parallelism

Task Oriented Parallel C/C++ (TOP-C) is a parallelization library developed over a decade [8, 10, 11, 13, 14]. It is designed to make it easy to convert a sequential program to a parallel one. It is latency-tolerant, allowing it to easily take advantage of multiple clusters, such as on a Grid [17]. One applies minimal modifications to a sequential program in order to the invoke the API for the TOP-C parallel library.

TOP-C has been used for many parallelizations on a wide variety of computations, several of them being the largest computations of their kind at the time. Some of its applications include linear algebra [12],construction of large permutation representations from matrix representations [18, 23, 33], coset enumeration [21, 20], and condensation of matrix representations [24, 22]. The TOP-C architecture has also been implemented in GNU Common LISP [8]and GAP (Group, Algorithms and Programming) [14]. Especially notable is the parallelization of Geant4 (see 3).

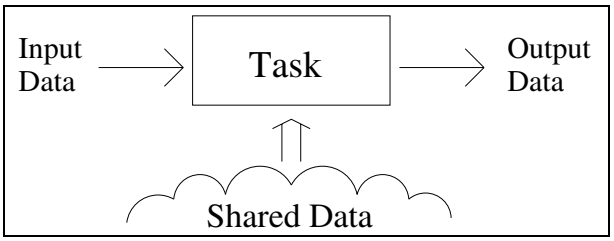


Figure 1. TOP-C Concept: The Task

2.1 TOP-C and the Task

TOP-C is built around the concept of a task. A sequential program usually has a small number of functions and inner loops where most of the time is spent. For simplicity, we assume only one such function. We identify the body of this function with the *task*. The arguments to the function are then identified with the *task input*. The return value is identified with the *task output*. Any global variables or other data outside the local scope are identified as *shared data*. A TOP-C application should not modify the shared data except through a TOP-C `UPDATE` action (see Section 2.2).

The callback functions can best be understood by considering three TOP-C concepts:

1. The Task (executed on a worker, see Figure 1)
2. The Action (directing the parallel strategy for TOP-C)
3. The Shared Data (common global data across all processes

An application writer writes a TOP-C application by defining the four TOP-C callback functions denoted in Figure 2. The four callbacks are registered with the TOP-C library through a library function `TOPC_master_slave`.

```
1. Master:  GenerateTaskInput() ⇒
   task_input
2. Worker:  DoTask(task_input) ⇒
   task_output
3. Master:  CheckTaskResult(task_input,
   task_output) ⇒ TOPC_action
4. Everywhere:  UpdateSharedData(task_input,
   task_output)
```

Figure 2. User-Defined Callback Functions

2.2 The TOP-C Actions and Shared Data

TOP-C achieves its non-trivial parallelism through the concept of *shared data*. Shared data is replicated on all processes. Before describing how non-trivial parallelism is implemented using a master-worker style, we must first review the relation of TOP-C actions and shared data.

When a TOP-C application executes, the same binary executes on the master and on all worker processes, as in SPMD (single program, multiple data) style. As an idle worker process becomes available, the master sends out a task input (the result of `GenerateTaskInput()`), and receives back from that worker a task output (the result of `DoTask(taskInput)`). When a task completes, the master process chooses one of the three primary *actions*, `NO_ACTION`, `UPDATE` and `REDO`.

Upon `NO_ACTION`, nothing further is done. Upon `UPDATE`, the shared data is modified uniformly across the master and all workers. (Recall that the *shared data* is any persistent data, such as global variables.) This uniformity is enforced by executing `UpdateSharedData(taskInput, taskOutput)` on each process.

TOP-C requires that the shared data be modified only through this `UPDATE` action, in order to maintain its uniformity. Hence, the TOP-C contract states that an application writer may directly access the shared data, but he or she may not modify the shared data, except through an `UPDATE` action. This contract ensures that if the shared data is initially uniform across all processes, it will remain so.

Finally, upon `REDO`, the same task is repeated on the original worker process. Note that the `REDO` action is useful only if TOP-C has previously modified the shared data due to an `UPDATE` action. In this case, the original task will be recomputed, but using the latest value of the shared data. This will produce a different task output.

2.3 Implementing Parallel Strategies in TOP-C

Given a working sequential application, and a parallel strategy, a working TOP-C application can usually be quickly implemented. We demonstrate this by considering three scenarios for parallelization:

1. trivial parallelism
2. optimistic concurrency
3. data flow

2.3.1 Implementation of Trivial Parallelism

For trivial parallelism, we identify the task with the function to be executed on each worker process. Upon completion of each task, the master process chooses `NO_ACTION`.

2.3.2 Implementation of Optimistic Concurrency for Parallelism

A second strategy easily implemented in TOP-C is one of *optimistic concurrency*. In databases, a policy of optimistic concurrency means that a node executes a transaction without committing the result. A check is then made whether the two concurrent transactions were legitimately executed in parallel. If the two transactions could not be legitimately executed in parallel, then the first transaction is committed, while the second transaction is rolled back, and then redone. The TOP-C master process rolls back and re-executes the second transaction by choosing a `REDO` action.

2.3.3 Implementation of Dataflow Strategy

The diagram in Figure 3 visually depicts this strategy. Imagine the computation of `DoTask()` as a data dependency computation. The top row represents the initial data and the interior nodes represent values of temporary variables prior to the final task output. The data at each node is computed from previous nodes. After an `UPDATE`, some of the shared data is modified (indicated by double circles in the top row). This dependency on modified data propagates to intermediate and final data (indicated by the double circles below the top row).

Suppose a worker process then receives a `REDO` request after the `UPDATE` described above. Then `DoTask()` need only compute the data nodes indicated by the double circles below the top row of Figure 3.

Hence, in this paradigm, upon an initial `DoTask()`, the application writer should save in a private global variable all data nodes corresponding to its computation. (This private global variable is considered to be outside the TOP-C shared data.) Upon receiving an `UPDATE` request, the values of all modified nodes in the shared data (the double circles of the first row) are also noted in the private global variable. Finally, upon a `REDO` request, only the computations of the modified intermediate and final nodes (the double circles below the first row) must be recomputed.

Further Parallel Strategies For another strategy, appropriate for data parallel computation, see the application of TOP-C to Gaussian elimination [12]. For a fuller description of the power and generality of TOP-C, see [9].

3 Geant4

Geant4 [2, 3, 15, 27], is a million line C++ program to simulate particle-matter interaction. Among other uses, it is being employed for the design of experiments at CERN, where the largest collider in the world is being built. Geant4, itself, is being developed with collaboration from

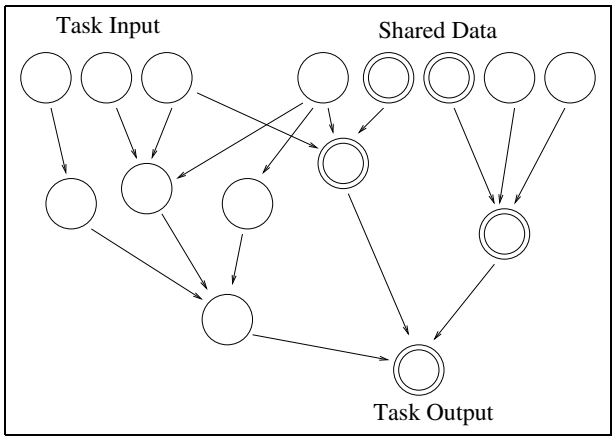


Figure 3. `DoTask()` **as a Data Dependency computation (double circles = modified data)**

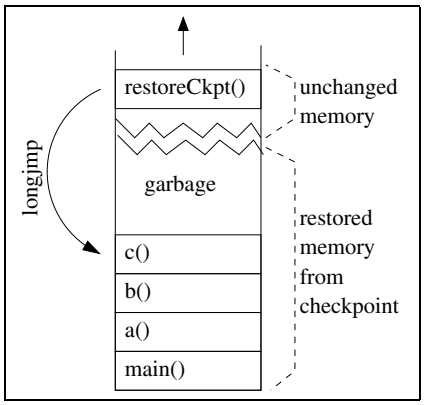


Figure 6. Transitioning into stack of checkpointed program

ten national high energy physics laboratories around the world.

The TOP-C parallelization of Geant4 [3, 4, 16, 17] is distributed with Geant4. In particular, one version of that parallelization is Grid-aware [16, 17].

4 Checkpointing and Restarting in Master-Worker Parallelism

Because TOP-C uses a master-worker style of parallelism and the master process already contains a copy of the TOP-C shared data, the task of checkpointing can be reduced to simply taking a snapshot of the master process. This is based on the fact that in our system the same state and shared data are uniformly maintained and updated across all processes. Upon restart, this one snapshot serves as a template to restore both master and workers' states. We create the snapshot by inducing a core dump in a forked copy of the process.

Checkpointing takes place without waiting for workers to complete the current outstanding tasks. The inputs for the outstanding tasks are saved as part of the checkpoint file and will be sent to workers again after restarting.

4.1 Checkpointing the Program State

Once the checkpoint routine is triggered, either by a timer or by an explicit request from the application, we flush write streams to disk. This "synchronizes" the stream with the underlying file descriptor. The current file offsets for each open file descriptor are next recorded in the file information table. We then call `setjmp` to save stack context information.

Next, we trigger a core dump by forking a child process, and calling `abort` in the child process. This is efficient, because the implementation of `fork` employs a copy-on-write policy. (Before calling `abort`, we call `setrlimit` to temporarily allow a sufficient coredump size.) The parent process then waits until the child process has exited. The parent process renames the `core` file to `checkpoint_topc.`*PID*.

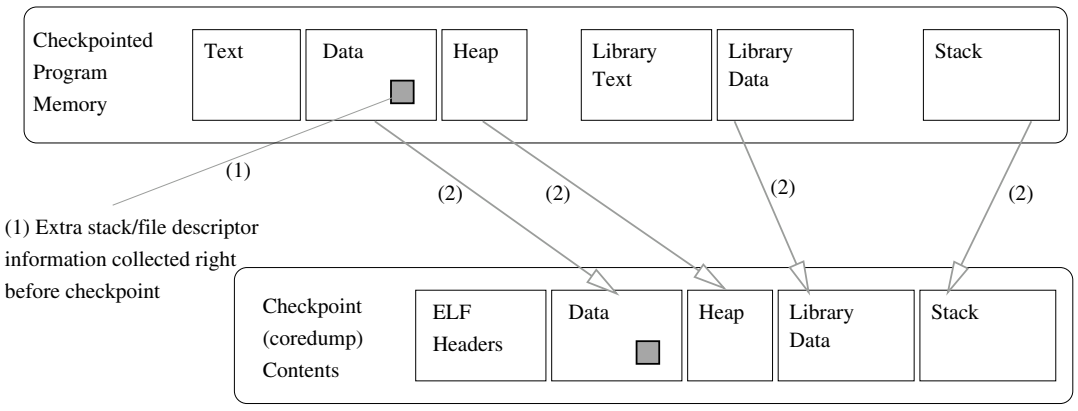
A standard operating system optimization allows the core dump to complete asynchronously. The pages of the child process may still be in a kernel buffer when the child process exits. This interacts well with the previous invocation of copy-on-write for the `fork`.

The file `checkpoint_topc.`*PID* now contains our file information table, and a newly created call frame context suitable for later use by `longjmp`. (See item (1) of Figure 4.)

4.2 Restarting after a Checkpoint

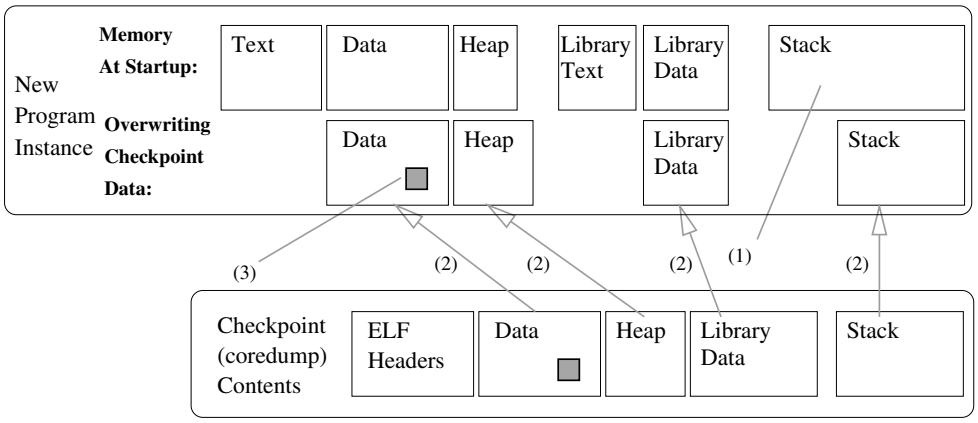
To restart a saved checkpoint, the original user program is run in the presence of an environment variable `TOPC_RESTART`, whose value is the path of the checkpoint file. The user program loads the checkpointed file (core dump) in four steps: 1) grow the new program stack past the old program stack; 2) load all segments of the core dump with the writable flag into memory, overwriting our current program; 3a) `longjmp` into the original stack and re-initialize TOP-C; and 3b) restore the kernel file descriptor state from the file information table by opening the listed file descriptors and seeking to their listed file offsets. (See Figure 5.)

One of the challenges is how to overwrite all of the current process's data and stack memory with that of the checkpointed data at the same time that the current process is exe-



(2) Standard system coredump triggered to write a snapshot of memory to file.

Figure 4. Saving program state to core



(1) Grow stack past the old stack, so top will be safe to use without destroying data
(2) Load data (incl. stack) from checkpoint file to location it was in in old process (overwriting existing memory)
(3) Restore file descriptors/streams and longjmp() into checkpointed stack

Figure 5. Restoring core to a running process

cuting. We overcome the problem by first growing the original stack until it is larger than the stack of the checkpointed program. The old stack is copied to the bottom of the current program's stack.

Now, the stack is a hybrid of the current and old stacks. Too many *return*s would be unsafe, as we would hit the boundary between the two stacks. We escape from this dilemma by calling `longjmp`. This brings us back into the old stack, while also shrinking the stack to the size of the old program. (See Figure 6.)

The process is now effectively a duplicate of the original, as it had existed before we checkpointed. Next, we restore file descriptors from the file information table, and call `freopen` on open streams. We then restart the worker processes. If the user defined an optional restore function, we call it. We then continue where the checkpoint left off.

Two other issues must also be addressed by the restart routine. First, upon restart, the loader may not have mapped all memory regions from our checkpointed process. For example, in GNU libc `malloc` calls above a threshold will in turn call `mmap`. We need to call `mmap` to create missing memory segments at their original address, prior to copying from the checkpointed file.

We do this by calling `mmap` with the suggested address. If the suggested address is not used by `mmap`, then we assume that the segment was previously allocated by the loader, and we unmap our copy of that segment. We do not use the `mmap` parameter `MAP_FIXED`, since its behavior upon collision with an existing segment is not standardized. For example, the GNU version unmaps the pre-existing segment.

The second issue for restart is that certain library functions such as `gethostbyname` fail after restart. This is because such functions communicate with a separate nameserver process. We suspect that our original process has cached the original socket address of the name-

server in the data segment of a system library. Our solution is to write a wrapper for `gethostbyname` that creates a new child process. The child process freshly invokes `gethostbyname`, and returns the desired information to the parent process.

4.3 Maintaining State of Open Files

To give the illusion that the status of open files has not changed across a checkpoint, we maintain a *file information table* to record the open file descriptors, which would otherwise be known only to the kernel. To populate this table, we intercept the following library calls: `open`, `fopen`, `fdopen`, `freopen`, `creat`, `close`, `fclose`, `dup` and `dup2`. We define our own wrapper functions of the same name. After intercepting a call by the application, the wrapper uses `dlopen` and `dlsym` (or in the case of kernel system calls such as `open`, it uses syscall) to call the original libc implementations. This allows us to transparently know what files are currently opened by the application. The list of open file descriptors and open streams is then recorded in the file information table. At restart time we will then recreate the file descriptor and stream states as described in the file information table.

Streams represent a special case. The file descriptor is maintained in kernel space, while the stream is maintained in user space. Hence, after the underlying file descriptor is reopened with the previous file offset, we use `freopen` to reassociate the existing stream with the required file descriptor.

4.4 Assumptions and Limitations

The package has been targeted toward UNIX and the ELF binary format. The current implementation runs in Linux. It could be ported to UNIXes using loader formats other than ELF. One key requirement to port to another operating system is the ability to copy data sections from a core file to their original location in memory. Additionally, it is assumed that if we run a program twice, its memory will be laid out at the same absolute addresses in virtual memory each time. This is important since user code will have pointers to global addresses in data. Relocating the data will invalidate those global addresses. We also use `dlsym`/`dlopen` and `syscall` to implement wrappers around system library calls and system calls.

Following are some important requirements to successfully restart the program:

- The dynamic libraries used by the application must not change between checkpoint and restart.
- Certain environment variables, $LD\_PRELOAD$, $LD\_LIBRARY\_PATH$, and LD_BIND_NOW, must not change between checkpoint and restart.
- The location of memory segments must not change between checkpoint and restart. (Also, see the following paragraph.)

As of Linux kernel 2.6.12, address randomization was implemented for security purposes. Hence, upon restart, a variable may have a different address than originally. Currently, we turn this off via `echo 0 > /proc/sys/kernel/randomize_va_space`. The Linux implementers also plan a per process mechanism, `PF_RANDOMIZE`, to turn off address randomization.

There is also a potential limitation to using wrappers around calls to system libraries. This strategy assumes that we can define our own function, e.g. `open`, that will shadow all calls to the standard system routine, `open`. The strategy fails under the following circumstances.

- A shared library may contain its own statically linked version of `open`, internal to that system library. In such cases, internal calls from the system library may not be bound to our own wrapper function, `open`.
- A library may call a versioned symbol, such as `open@@GLIBC_2.1`. For example, the GNU C++ standard library uses this trick to guarantee a fixed, stable implementation when it calls symbols from @libc.so@. This requires our checkpointing library to track versioned symbols and intercept them, although version numbers may change in future versions of the C++ standard library.

While we can currently work around the above issues, we are investigating the use of the `proc` kernel interface instead of wrappers where such issues may arise. For example, our calls to `open` and friends are used to detect open file descriptors, while the kernel directly exports that information through `/proc/`*PID*`/fd/`.

Currently, we do not support heterogeneous computing in our implementation, since that would require us to checkpoint an example worker process for each architecture present (aside from the architecture of the master).

5 Fault Tolerance as Worker Processes Fail

TOP-C tries to detect two failure modes: *slow worker nodes* and *dead worker nodes*. A worker node is considered *dead* when the socket to that node is no longer alive. This occurs and is detected when the worker process has died (POSIX `ECONNRESET`), or when the network socket connection has died (POSIX `EPIPE`). A worker node is considered *slow* if the corresponding processor is heavily loaded, lacks sufficient resources, or when a network connection is experiencing intermittent network failures. A slow node is detected if a worker fails to return from a task in a timely manner, and if a replicate of the task then finishes earlier.

6 Performance

6.1 Checkpoint Timing

We achieve fast checkpointing by pushing as much work as possible to be performed on restart. The logic is that the checkpointing will occur more often than restart.

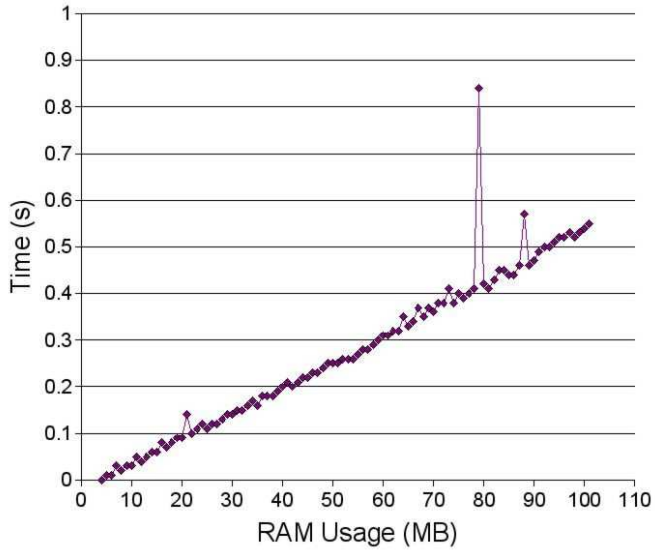


Figure 7. Time to checkpoint, excluding asynchronous flush of the checkpoint file to disk. (The master process is delayed by the time shown.) Spikes (e.g. at 80 MB) are artifacts that vary between runs.

As seen in Figure 7, the checkpoint timing scales linearly with respect to memory usage. This is dominated by the time to fork and wait on the child process while it calls `abort()` to trigger a core dump. The fact that the times to checkpoint are faster than disk bandwidth demonstrate that the operating system is writing the core dump to disk asynchronously. After the process resumes computation, there is still a delay before the full core dump is flushed to disk.

In a second experiment, we measure that delay. We insert a call to `sync` immediately after invoking the checkpoint. The time for `sync` to execute represents the additional delay and is shown in Figure 8,

Our checkpointing solution only involves the master node. This means that none of the worker nodes are interrupted during a checkpoint. Timings should theoretically be completely independent of the number of nodes used. We confirm this with the test described in Figure 9.

Tests were run on a Mobile Athlon64 3000+ processor with 512 KB cache and 1 GB of RAM. The operating system was Debian Linux ("Sid") with kernel 2.6.14. The C library used was glibc 2.3.5.

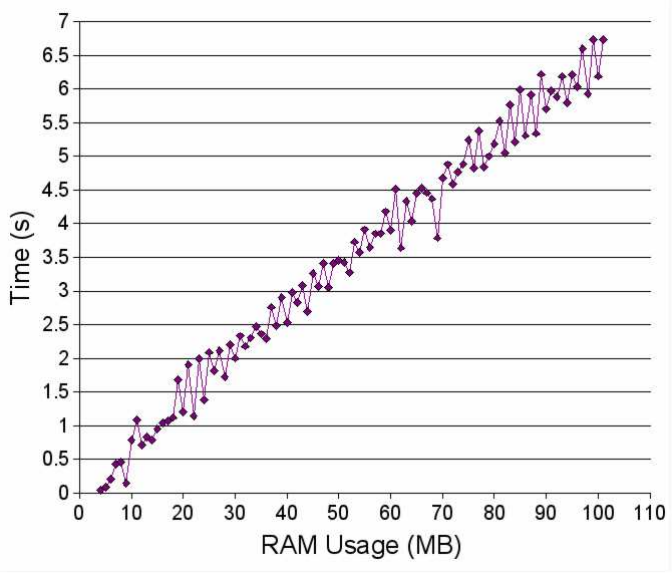


Figure 8. Time for sync to complete when called immediately after a checkpoint. This time is normally concurrent with the ongoing computation.

6.2 Restart Timing

The cost of restarting a checkpoint can be broken into three parts:

1. The cost to restore memory from checkpoint. This is similar to the time to read the entire checkpoint file from disk. It grows as RAM usage on the master grows. In our informal experiments, this time was always bounded above by the times of Figure 8.

2. The cost to reinitialize the cluster. This grows as the number of nodes in new cluster grows. This is approximately the same as the time to execute `MPI_Init`, and is usually very reasonable.

3. The cost to redo lost progress. Since we only checkpoint on master, all progress on outstanding tasks is lost. This work must be redone before we have reached the point at which the checkpoint was taken. The time to reach pre-checkpoint progress could be approximated by (AVG-TASK-LENGTH / 2) * (NUM-SLAVES-BEFORE-CKPT / NUM-SLAVES-AFTER-CKPT). The latter of the two terms represents the possibility of using fewer or more processors upon restart.

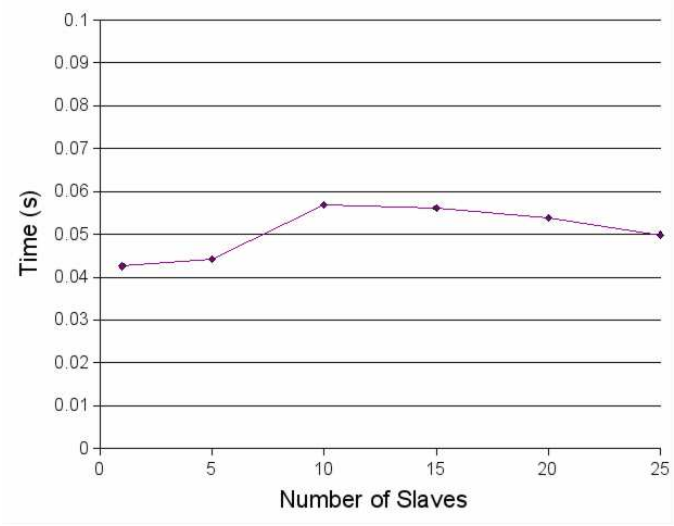


Figure 9. Checkpoint timings for Geant4 (parallel application from `geant4/example/extended/parallel/ParN02`).

Most TOP-C applications will have short task times. If an application were to have very long tasks, this would be an indication that we are in a scenario equivalent to running multiple long-running independent applications.

Experimental results for restarts are omitted because restarts happen rarely and the timings for Part 3 vary greatly among different user applications.

7 Related Work

Checkpointing packages are sometimes classified as *system-level* (system checkpoints all data) versus *application-level* (application directs which data to checkpoint). Additionally, a system-level package is transparent if the application writer need not specify where a checkpoint is allowed. Additionally, a checkpointing package may or may not require kernel modifications. Checkpointing packages for parallel computation may also be classified as *blocking* (requiring all processes to stop at a barrier) or *non-blocking*.

One of the first checkpointing packages was Condor [30], which provided system-level checkpointing without modifications to the kernel. Konuru et al. [28] provide an early example of application-level checkpointing — specifically for PVM. These implementations were intended primarily to support process migration for single processes.

Since then, system-level checkpointing of the multiple processes of a parallel computation has become available [26, 34, 29]. Some more recent checkpointing innovations are in-memory checkpointing [35] and incremental checkpointing of modified data only [1]. Each of the above system-level packages must checkpoint all participating processes in a parallel computation.

The work of this paper falls into the category of transparent, system-level checkpointing that does not modify the kernel. By restricting to master-worker parallelism, it gains efficiency by checkpointing only the master process. For a comparison of general checkpointing techniques for parallel computation that do not require a barrier, see the work of Bouteiller et al. [5].

8 Acknowledgements

We gratefully acknowledge discussions with Mike Rieker about implementation techniques.

References

[1] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM Press.

[2] S. Agostinelli et al. Geant4: a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A*, 506(3):250–303, July 2003. (over 100 authors, including G. Cooperman).

[3] J. Allison et al. Geant4 developments and applications. *IEEE Transactions on Nuclear Science*. to appear (73 authors, incl. G. Cooperman).

[4] G. Alverson, L. Anchordoqui, G. Cooperman, V. Grinberg, T. McCauley, S. Reucroft, and J. Swain. Using TOP-C for commodity parallel computing in cosmic ray physics simulations. *Nuclear Physics B (Proc. Suppl.)*, 97:193–195, 2001.

[5] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. In *2003 IEEE International Conference on Cluster Computing (Novel Computing Session)*, 2004.

[6] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards world-wide supercomputing. In *Proceedings of the Seventh ACM SIGOPS European Workshop on System Support for Worldwide Applications*, 1996.

[7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[8] G. Cooperman. STAR/MPI: Binding a parallel library to interactive symbolic algebra systems. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, volume 249 of *Lecture Notes in Control and Information Sciences*, pages 126–132. ACM Press, 1995.

software at URL: http://www.ccs.neu.edu/home/gene/software.html\#starmpi.

[9] G. Cooperman. *TOP-C: Task Oriented Parallel C/C++*. 1996–. http://www.ccs.neu.edu/home/gene/topc.html, includes 40-page manual.

[10] G. Cooperman. TOP-C: A Task-Oriented Parallel C interface. In 5^{th} *International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 141–150. IEEE Press, 1996. software at http://www.ccs.neu.edu/home/gene/topc.html.

[11] G. Cooperman. GAP/MPI: Facilitating parallelism. In *Proc. of DIMACS Workshop on Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 69–84. AMS, 1997.

[12] G. Cooperman. Practical task-oriented parallelism for Gaussian elimination in distributed memory. *Linear Algebra and its Applications*, 275–276:107–120, 1998.

[13] G. Cooperman. TOP-C: Task-Oriented Parallel C for distributed and shared memory. In *Workshop on Wide Area Networks and High Performance Computing*, volume 249 of *Lecture Notes in Control and Information Sciences*, pages 109–118. Springer Verlag, 1999. http://www.ccs.neu.edu/home/gene/topc.html.

[14] G. Cooperman. Parallel GAP: Mature interactive parallel computing. In *Groups and Computation III*, pages 123–138. DeGruyter Publishers, 2001. software at URL: http://www.ccs.neu.edu/home/gene/pargap.html.

[15] G. Cooperman. Parallelism in Geant4. In *Geant4 2003 Workshop, TRIUMF, Vancouver*, 2003. http://www.triumf.ca/geant4-03/talks/05-Friday-PM-1/05-G.Cooperman/.

[16] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel. Using TOP-C and AMPIC to port large parallel applications to the computational grid. In *Proc. of* 2^{nd} *IEEC/ACM Interantional Symposium on Cluster Computing and the Grid (CCGrid)*, pages 120–127. IEEE Press, 2002.

[17] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel. Using TOP-C and AMPIC to port large parallel applications to the computational grid. *Future Generation Computer Systems (FGCS)*, 19:587–596, 2003.

[18] G. Cooperman, L. Finkelstein, M. Tselman, and B. York. Constructing permutation representations for matrix groups. *J. of Symbolic Computation*, 24:471–488, 1997.

[19] G. Cooperman and V. Grinberg. TOP-WEB: Task-oriented metacomputing on the web. *International Journal of Parallel and Distributed Systems and Networks*, 1:184–192, 1998.

[20] G. Cooperman and V. Grinberg. Scalable parallel coset enumeration: Bulk defi nition and the memory wall. *J. Symbolic Computation*, 33:563–585, 2002.

[21] G. Cooperman and G. Havas. Practical parallel coset enumeration. In *Proc. of Workshop on High Performance Computation and Gigabit Local Area Networks*, volume 226 of *Lecture notes in control and information sciences*, pages 15–27. Springer Verlag, 1997.

[22] G. Cooperman, G. Hiss, K. Lux, and J. Müller. The Brauer tree of the principal 19-block of the sporadic simple Thompson group. *J. of Experimental Mathematics*, 6(4):293–300, 1997.

[23] G. Cooperman, W. Lempken, G. Michler, and M. Weller. A new existence proof of Janko's simple group j_4. In *Progress In Mathematics*, volume 173, pages 161–175. Birkhauser, 1999.

[24] G. Cooperman and M. Tselman. New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. In *Proc. of International Symposium on Symbolic and Algebraic Computation (ISSAC '96)*, pages 155–160. ACM Press, 1996.

[25] R. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the InteGrade Grid middleware. In *Proc. of 2nd Workshop on Middleware for Grid Computing*, pages 35–40. ACM Press, 2004.

[26] M. D. Dikaiakos, editor. *Grid Computing, Second European Across Grids Conference, AxGrids 2004, Nicosia, Cyprus, January 28-30, 2004, Revised Papers*, volume 3165 of *Lecture Notes in Computer Science*. Springer, 2004.

[27] Geant4 webpage. Geant4, 1999–. http://wwwinfo.cern.ch/asd/geant4/geant4.html.

[28] R. B. Konuru, S. W. Otto, and J. Walpole. A migratable user-level process package for pvm. *J. Parallel Distrib. Comput.*, 40(1):81–102, 1997.

[29] O. Laadan, D. Phung, and J. Nieh. Transparent networked checkpoint-restart for commodity clusters. In *2005 IEEE International Conference on Cluster Computing*. IEEE Press, 2005.

[30] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical report 1346, University of Wisconsin, Madison, Wisconsin, April 1997.

[31] C. Pinchak, P. Lu, and M. Goldenberg. Practical heterogeneous placeholder scheduling in overlay metacomputers: Early experiences. In *8th International Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *Lecture Notes in Computer Science*, pages 205–228. Springer, 2002.

[32] SETI@home. http://setiweb.ssl.berkeley.edu, 1996–.

[33] M. Weller. Construction of large permutation representations for matrix groups II. *Applicable Algebra in Engineering, Communication and Computing*, 11:463–488, 2001.

[34] N. Woo, S. Choi, hyungsoo Jung, J. Moon, H. Y. Yeom, T. Park, and H. Park. MPICH-GF: Providing fault tolerance on grid environments. The 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003), the poster and research demo session May, 2003, Tokyo, Japan.

[35] G. Zheng, L. Shi, and L. Kale. FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *2004 IEEE International Conference on Cluster Computing (Fault-Tolerant Session)*, pages 93–103, 2004.