Adaptive Checkpointing for Master-Worker Style Parallelism (Extended Abstract)

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

1 Introduction

We present a transparent, system-level checkpointing solution for master-worker parallelism that automatically adapts, upon restore, to the number of processor nodes available. We call this *adaptive checkpointing*. This is important, since nodes in a cluster fail. It also allows one to adapt to using mutliple cluster partitions, 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 (0.05 s in our case), and more economical of disk space.

We describe a *system-level* solution. 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 solution has been implemented in TOP-C (Task Oriented Parallel C/C++) [1], an open source software developed over ten years. While there is a common conception that master-worker style parallelism is limited to "embarrassingly trivial" parallel programs, this is not the case. For example, TOP-C supports optimistic concurrency, dataflow diagrams, and other parallel models through a simple master-worker model. (See the overview on the TOP-C home page [1] for further information.)

2 Checkpointing and Restoring 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 taking a snapshot of the master process, only. This is based on that fact that in our system the same state and shared data are uniformly maintained and updated across all processes. Upon restore, this one snapshot serves as a template to restore both master and workers' memory. We create the snapshot by inducing a core dump in a forked copy of the process.

To resume a saved checkpoint, the master is first restored by running the original unmodified binary with a special flag which causes the checkpointed data to be loaded from the checkpoint file. It copies the checkpointed stack onto the current stack, and restores the context of the call frame from which the process was checkpointed. (See Figure 1.) The restored master spawns off the worker processes, which then repeat the same sequence of events.

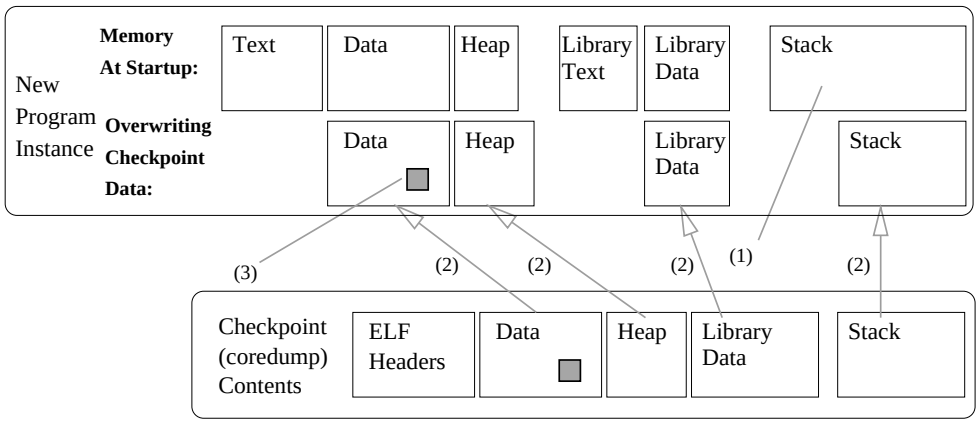
Maintaining State of Open Files To give the illusion that the status of open files has not changed across a checkpoint, we maintain a *file information table* to record such kernel information as open file descriptors. To populate this table, we intercept the library calls: `open`, `close`, `fopen`, `fclose`, `dup` and others. We define our own wrapper functions of the same name. After intercepting a call by the application, the wrapper uses `dlopen` and `dlsym` to call the original libc implementations, and then records the results in the file information table.

At checkpoint time we record the current file offsets for each open file in the file information table. At restore time we recreate the file/stream state described in the file information table.

Checkpointing the program state. Once the system is in a consistent state (no outstanding tasks), we update file offsets in the file information table and call `setjmp` to save stack context information. Then, we simply trigger a core dump (without stopping execution of the current program), the work of interpretting this core dump will be done at restore time.

Restoring from a checkpoint. To restore, we run the original user program with a "restore" flag. Loading the checkpoint file (core dump) is done in four main 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; 3) re-

*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/).



(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 file (overwriting existing memory)
(3) Restore file descriptors/streams and longjmp() into checkpointed stack

Figure 1. Restoring core to a running process

store the kernel file descriptor state from the file information table; and 4) `longjmp` into the original stack and re-initialize TOP-C. (See Figure Figure 1.)

The checkpointed data may contain ELF sections that were not present in the original executable, because, for example, of calls to `mmap`. We must again `mmap` such sections, before copying their data from the checkpoint file. (Note, in GNU libc `malloc` calls above a threshold will in turn call `mmap`.)

We are now effectively back inside a duplicate of the program, as it 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.

Assumptions and Limitations The package has been targeted toward UNIX, using ELF. The current implementation runs in Linux. It could be ported to UNIXes using loader formats other than ELF. The key requirements for a port to another O/S are `dlsym`/`dlopen` and 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, since code will have pointers to data, and ELF is not available for relocation.

3 Fault Tolerance as Worker Processes Fail

MPI standards do not require fault tolerance capability. If one process in an MPI computation fails, or one socket fails, the entire computation is aborted. This is reasonable in MPI, but it is a disaster for a parallel application that has been running for weeks. 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.

4 Timing

The tests were run on a 2.4 GHz Pentium-4 with 512 KB cache and 1 GB of RAM. The operating system was Debian Linux ("Sid") with kernel 2.6.10. The C library used was glibc 2.3.2.

The master always completes its checkpoint in less than 0.05 seconds. This is the time to fork and wait on the child process while it calls `abort()` to trigger a core dump. The core is then renamed as the checkpoint file. The O/S asynchronously writes out the core file while the user process proceeds.

Restoring checkpoints involves the time to restore the memory, plus the time to rerun the initialization code to spawn off all the nodes. The time was close to the time to read the checkpoint file from disk.

The actual cost depends on additional factors, such as: (1) waiting for slaves to complete their current task before a checkpoint; (2) application-dependent reinitialization after a restore; (3) and a time proportional to the number of nodes for the purpose of running ssh and opening sockets to each worker. A future version will not wait for slaves to complete before checkpointing.

References

[1] 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`.