

DMTCP: Scalable User-Level Transparent Checkpointing for Cluster Computations

Jason Ansel
Computer Science and
Artificial Intelligence Laboratory
MIT
Cambridge, MA 02139 / USA
jansel@csail.mit.edu

Kapil Arya Gene Cooperman*
College of Computer
and Information Sciences
Northeastern University
Boston, MA 02115 / USA
{kapil,gene}@ccs.neu.edu

Abstract

As the size of clusters increases, failures are becoming increasingly frequent. Applications must become fault tolerant if they are to run for extended periods of time. We present DMTCP (Distributed MultiThreaded CheckPointing), the first user-level distributed checkpointing package not dependent on a specific message passing library. This contrasts with existing approaches either specific to libraries such as MPI or requiring kernel modification. DMTCP provides fault tolerance through checkpointing. DMTCP transparently checkpoints general cluster computations consisting of many nodes, processes, and threads. DMTCP automatically accounts for TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals), signal handlers, ordinary file descriptors, shared file descriptors, and other operating system artifacts. We demonstrate checkpointing and restart of applications communicating through MPICH2, OpenMPI, and sockets directly. These applications were written with a variety of languages including Fortran, C, C++, and Python. Results show that checkpoint time remains nearly constant as the number of nodes increases.

1 Introduction

Large cluster computations are seeing several node failures each day. This problem will only get worse as distributed computations grow ever larger. As of November, 2007, 429 of the top 500 supercomputers used more than 1,024 processors [1]. This level of scalability is becoming widespread for many scientific applications being run at the national laboratories and in certain industries. Checkpointing can be used to provide fault tolerance with little or no programmer effort.

We present DMTCP (Distributed MultiThreaded CheckPointing). DMTCP provides transparent user-level distributed checkpointing. Checkpointing is the act of saving the state of a computation to disk so that it can later be restarted. DMTCP is able to checkpoint the state of an arbitrary network of multithreaded processes connected by sockets and spread across many nodes. DMTCP can then restart this network of processes on the same, or a different, set of nodes. It is also able to migrate individual processes between nodes. Thus, the total number of nodes at checkpoint time need not be the same at restart time.

DMTCP is the first checkpointing package that is both distributed, user-level, and not specific to a message passing library. This is in contrast with existing work that either: is for a single process only; requires kernel modification; or is specific to a message passing library. Section 2 provides a more detailed analysis of related work. Our user-level approach is particularly important because it allows an application developer to directly bundle our checkpointing package, with the knowledge that an unprivileged user can still run it on any target Linux cluster. This is achieved without loss of generality; an arbitrary cluster application can still be checkpointed without the need to recompile or relink.

The approach of this paper is motivated by the four following usage scenarios:

*This work was partially supported by the National Science Foundation under Grant CNS-06-19616.

1. *improving fault tolerance*: checkpointing a traditional distributed scientific computation, such as a long-running MPI computation;
2. *migration and load balancing*: allowing better cluster management by suspending low-priority tasks and possibly migrating them to other resources;
3. *interactive problem-solving*: checkpointing an interactive scientific problem-solving session for later consideration; and
4. *debugging*: debugging parallel computations where bugs may arise only after hours or days of execution. One can restart from a checkpoint rather than re-running the application from the beginning.

Most large clusters, such as the ones in TeraGrid, are shared resources. Despite the fact that these clusters consist of thousands of nodes, if one wishes to run a task on hundreds or thousands of nodes, one must wait in a queue for days. Using distributed checkpointing, one could schedule a cluster in much the same way one schedules a processor. Tasks running on hundreds of nodes could be moved or suspended in order to more efficiently utilize shared cluster resources. Computations could automatically be scaled down (or up provided there are more processes than nodes) by migrating processes in the computation onto a different number of nodes. Unlike existing process migration systems like Condor [2], no stub process is left on the original node.

Interactive problem solving environments, such as iPython [3] and Star-P [4], are becoming popular in some fields. These systems consist of an interactive terminal backed by a cluster of compute machines. DMTCP can transparently save the state of this terminal, and all backing compute nodes, so that it can be resumed at a later time. This is useful for capturing an interesting state for later inspection. It is also useful so one can release costly compute resources while the researcher is taking a break.

Debugging is a particularly useful application of checkpointing. One can run the application with frequent checkpointing enabled. These checkpoints can be written as a log snapshots of the program’s execution. To diagnose a problem, the programmer can work backwards in time, resuming older and older checkpoints.

A novel twist on the above debugging example occurs when running a large computation in batch on a cluster with expensive accounting. After a program fault, the programmer retrieves the last checkpoint image, which will illustrate the bug. The checkpoint image of the misbehaving computation is then migrated to a developer machine (all processes on a single host), where it can be inspected at leisure, and outside of the restrictions of a batch environment.

Unprivileged checkpointing is also useful for the “ultimate bug report”. When using third party applications, rare bugs can be reported by the end user in the form of a restartable snapshot taken right before the bug occurs. This produces the analog of a core dump, when running an application with multiple processes.

Basing DMTCP directly on sockets has allowed us to gain a broad base of applications coverage. We demonstrate checkpointing of applications running with two of the most widely used dialects of MPI: MPICH2 [5] and OpenMPI [6]. (MPI stands for Message Passing Interface [7].) Additionally, one is no longer restricted to MPI-based applications. We also demonstrate on a non-MPI application, iPython, which, in conjunction with SciPy and NumPy, has enabled large efficient parallel computations using Python [3].

By the nature of the problem, robust checkpointing in user space requires that a plethora of issues be tackled in order to checkpoint general applications successfully. Some issues that one faces are: shared socket descriptors; checkpointing socket buffers when messages are still “in flight”; pipes; stopping all threads of process; checkpointing file locks; restoring signal handlers; pseudo-terminals; and system services backed by daemons.

We demonstrate a two-layer approach to distributed checkpointing. These layers consist of:

1. DMTCP, reported on in this paper, allows checkpointing of a network of processes spread over many nodes. After DMTCP copies all inter-process information to user space, it delegates single-process checkpointing to a separate checkpoint package.
2. We base single-process checkpointing on our previous work, MTCP (Multi-Threaded CheckPointing) [8].

These two layers are separate, with a small interface between them. This two-layer user-level approach has a potential advantage in non-Linux operating systems, where DMTCP can be ported to run over other single-process checkpointing packages that may already exist.

Checkpointing is added to arbitrary applications by injecting a shared library at execution time. This library:

- Launches a checkpoint management thread in every user process which coordinates checkpointing.
- Adds wrappers around a small number of `libc` functions in order to record information about open sockets at their creation time.

System calls and the `proc` filesystem are also used to probe kernel state.

We use a *coordinated checkpointing* method, where all processes and threads cluster-wide are simultaneously suspended during checkpointing. Network data “on the wire” and in kernel buffers is flushed into the recipient process’s memory and saved in its checkpoint image. After a checkpoint or restart, this network data is sent back to the original sender and retransmitted prior to resuming user threads. A more detailed account of our methodology can be found in Section 4

The only global communication primitive used at checkpoint time is a barrier. At restart time, we additionally require a discovery service to discover the new addresses for processes migrated to new hosts. Both barriers and discovery are currently implemented by a single coordinator process for simplicity of implementation. Experiments show that this is not currently a bottleneck. If it become a bottleneck in the future, we could easily switch to well-known scalable distributed algorithms for barriers and discovery.

While not a focus of this paper, it is important to note that usage scenario 1, fault tolerance, requires a robust storage solution for checkpoint images in order to handle non-transient failures. Robust, scalable storage is not a problem unique to checkpointing, thus many clusters will already have systems in place that can be utilized for storing checkpoints. Systems such as Lustre Filesystem[9] have been proven to scale beyond 10,000 nodes. Clusters without robust storage could utilize a method of “buddy backup” where checkpoint images are copied to a neighboring node for redundancy. We present timing results both where checkpoints are written to local disk and where checkpoints are written to centralized RAID storage.

The primary contributions of this paper are that:

- We demonstrate the first general *user-level distributed* checkpointing package. (Previous user-level examples were specific to particular MPI dialects, while this work is directly based on sockets. This makes our approach more general and thus applicable to a larger class of applications.)
- We show experimental results that demonstrate the scalability and coverage of our approach. These results are presented in Section 5.
- The accompanying implementation is provided free and open source at:
<http://sourceforge.net/projects/dmtcp>

1.1 Background

1.1.1 Single process checkpointing: classification

We motivate the choice of user-level checkpointing by reviewing the four most common checkpointing methods. The problem of checkpointing a single process to disk is one of saving the state of:

- user-space memory; and
- kernel state associated with the process.

The major types of checkpointing are:

1. **checkpointing an entire operating system image:** both user-space and kernel-space memory are checkpointed to disk. This is the approach used by virtualization systems such as VMWare.

- 2. kernel-level checkpointing:** modify the kernel or add a kernel module allowing the process-specific state in the kernel to be easily saved and restored with all process memory.
- 3. user-level checkpointing:** use existing operating system APIs (system calls, the `proc` filesystem, etc.) to extract (and later re-create) the process-specific state in the kernel. Additionally program memory is saved and restored with the checkpoint.
- 4. application-level checkpointing:** provide tools for the application writer to save to disk application data that has been created or modified since process startup. Rather than saving all state, only save specific user selected state.

Method 1 above requires extensive disk space, and is not flexible. Method 2 is highly sensitive to upgrades of the kernel. Method 2 also requires root (administrator) privilege for installation. Method 4 is labor-intensive in complex applications.

User-level checkpointing (method 3) was chosen for its transparency. The end user requires no modification of kernel or application. The key to robust user-level checkpointing is a complete list of process-specific kernel state, and the use of existing kernel APIs to save and restore that state.

1.1.2 Distributed checkpointing: classification

The problem of distributed checkpointing is a superset of single process checkpointing. In addition to user-space and kernel-space, one must also save the state of:

- The graph¹ of connections between processes; and
- The data “in flight” between nodes.

Additionally, the snapshot must be taken in a way that it is globally consistent, as if it were taken at a single moment in time. There are two main techniques for doing this:

- 1. Coordinated checkpointing** where computation is globally suspended, and all processes are checkpointed simultaneously.
- 2. Uncoordinated checkpointing** where checkpoints are compiled at different times and consistency is gained through message logging that must be replayed at restart time.

Both techniques are useful in different scenarios. Method 1 is effective on low latency networks, such as clusters, where computation can be globally suspended in a fast, scalable way. Method 2 holds an advantage on high latency networks, where broadcasting commands to all nodes is difficult. We chose coordinated checkpointing (method 1) for its simplicity and out of a desire to never have checkpointing and user computation competing for resources.

2 Related Work

While there is a long history of checkpointing packages (kernel- and user-level, coordinated and uncoordinated, single-threaded vs. multi-threaded, etc.), this represents the first user-level distributed checkpointing package that is not MPI-specific. Given the space limitations, we highlight only the most significant of the many other approaches.

Section 1.1.1 provides a taxonomy of the wealth of checkpointing work, even for the limited case of single process checkpointing. We do not discuss checkpointing of the entire operating system image, or application-level checkpointing (modification of the application source code), as being too far from our goal of transparent checkpointing of a process.

Many packages exist which perform single-process checkpointing [10, 11, 12, 13, 14, 15, 16, 17, 18, 2]. Far fewer packages provide distributed checkpointing. We divide existing work on distributed transparent checkpointing into two categories:

¹ A graph is actually insufficient to represent socket relations. Something closer to a hypermultigraph is needed to represent the facts that a single socket descriptor can be shared between many processes and that nodes can be connected through multiple sockets.

1. *User-level MPI libraries for checkpointing* [19, 20, 21, 22, 23, 24, 25, 26, 27]: works for distributed processes, but only if they communicate exclusively through MPI (Message Passing Interface). Typically restricted to a particular dialect of MPI.
2. *Kernel-level (system-level) checkpointing* [28, 29, 30, 31]: modification of kernel; requirements on matching package version to kernel version.

DMTCP falls into neither of these categories. Unlike 1, DMTCP does not restrict the application to a specific library. Unlike 2, DMTCP does not require kernel modification or even administrator privileges.

Two notable distributed kernel-level solutions based on the Linux kernel module Zap are provided by Laadan and Nieh [29, 30] and Janakiraman et al. [28]. A difficulty of this approach is that the Zap kernel module must be kept in sync with the intended version of the kernel. This approach leads to checkpoints being tightly coupled to kernel versions. In contrast, DMTCP supports migration between machines with different kernels, libraries, and hardware.

Much MPI-specific work has been based on *coordinated checkpointing* and the use of hooks into communication by the MPI library [19, 20]. In contrast, our goal is to support more general distributed scientific software.

Such job management systems as LFS and PBS currently have hooks to support third-party checkpointing software. However, they do not currently checkpoint general, distributed computations. The LAM implementation of MPI (as of version 7.0) provides hooks to single-process checkpointing packages, in order to enable distributed checkpointing of MPI computations [21]. It is not clear how easily LAM supports multithreaded user applications.

Non-coordinated checkpointing approaches also exist, such as the Chandy-Lamport algorithm [32]. However, these approaches add an additional cost to messages, as they compute a globally consistent snapshot.

Process migration is also related to checkpointing, but only a subset of the issues must be overcome [33, 34, 35]. A common process migration strategy is to maintain a stub process in place of the original process, and the system calls of the migrated process will be intercepted and passed on to the stub process, that provides appropriate virtualization services. This was originally the core of the strategy used by the ground-breaking system Condor [2].

3 Usage and Features

Checkpointing is the action of saving process state to persistent storage, so that the process can later be *restarted*. *Distributed checkpointing* is the action of copying to persistent storage the state associated with all processes of a distributed computation in such a way that the entire computation can be resumed at a later time.

DMTCP provides three commands to the user:

```
dmtcp_coordinator [ Coordinates checkpointing across cluster ]
dmtcp_checkpoint <program>
dmtcp_restart_script.sh
```

Each invocation of `dmtcp_checkpoint` by the end user causes the corresponding process to be registered as one of the set of processes that will be checkpointed. The DMTCP environment variables `DMTCP_HOST` and `DMTCP_PORT` are set by the end user to enable `dmtcp_checkpoint` to find the `dmtcp_coordinator` for the cluster. They are inherited by forked or exec'ed processes, and DMTCP propagates their values to remotely spawned processes created via `ssh`.

A user may directly request a checkpoint be generated, or the coordinator may generate checkpoints at regular intervals. In the console of the coordinator, the `c` command initiates a checkpoint immediately, while `t` command sets an interval for automatic checkpoints. Automatic checkpoints may also be enabled through the `DMTCP_CHECKPOINT_INTERVAL` environment variable. Other commands such as `l` and `h`, list participating processes and display help.

Each participating process is simultaneously checkpointed. The coordinator is *not* checkpointed, because it is a stateless server used to implement barriers during checkpointing. Checkpoint files are written with unique names to a user specified location. During each checkpoint, a new copy of the script, `dmtcp_restart_script.sh`, is created on the host of the coordinator.

This script is simply a series of commands of the form:

```
ssh hostA dmtcp_restart ckpt_process1 ckpt_process2 ... &
```

All participating processes on a single host are restarted at the same time, so that shared file descriptors can be restored. At restart time, the new coordinator acts as a discovery service to allow restart programs to discover the new addresses for relocated peers. Process migration can be accomplished by editing the script to specify the new hostnames and, if necessary, copying the checkpoint files. Single processes, that do not share socket descriptors, can be split up and restarted on separate nodes at restart time. Similarly, processes from many nodes can be restarted on a single node.

A configurable signal (*SIGUSR2* by default) is used by the checkpoint coordinator to capture the attention of other processes. A draft POSIX 1003.m had proposed a new signal, *SIGCKPT*, to avoid conflict with application signals.

3.1 Application Coverage

We have verified DMTCP through a wide set of test suites. The first such challenge to support distributed scientific computations is to test on MPI (Message Passing Interface) [7]. At present, we have successfully checkpointed applications running on the two most widely used dialects of MPI: MPICH2 [5] and OpenMPI [6].

Each MPI implementation invokes run-time daemons to manage the jobs. OpenMPI invokes an OpenRTE daemon to manage OpenMPI application processes. MPICH2 invokes MPD (MPI Daemon). MPD is implemented as a Python script. At least one corresponding daemon must run on each computer that is running MPI. DMTCP transparently checkpoints the MPI daemons along with the MPI application processes. In this context, the initial invocation might be, for example:

```
dmtcp_checkpoint mpirun -np 32 <mpi-program>
```

We also checkpoint iPython as an example of a scientific problem-solving environment. This corresponds to usage scenario 3 described in the introduction. This also represents an application that does not communicate through MPI, but rather sockets directly. Interpreted languages, such as python, are checkpointed by checkpointing the entire interpreter that is running them. We used a similar approach when checkpointing Java applications, in our previous work [8]. In that case, we checkpointed the entire Java Virtual Machine running the Java program.

4 Checkpoint-Restart Architecture

4.1 Checkpointing a Single, Multi-Threaded Process

This review of checkpointing a single process is provided as background. A more detailed report of our single-process methodology can be found in previous work, MTCP (MultiThreaded CheckPointing) [8].

MTCP checkpoints the state of both user-space and kernel-space. User-space state is captured directly through memory reads. Kernel-state is discovered indirectly by probing via system calls and the `proc` filesystem. This kernel state can later be restored through a different set of system calls. Restoring user memory involves a more complex bootstrapping process similar to a loader.

Threads are initially discovered through a wrapper around the `_clone` libc function (indirectly called through `pthread_create`). At checkpoint time, a custom signal handler is installed and signals are used to hijack each user thread. Information is stored about each thread, and each thread is suspended temporarily during checkpointing.

4.2 Distributed Checkpointing

4.2.1 Retrofitting single-process checkpointing for distributed processing

We built DMTCP upon our previous work, MTCP (described in 4.1). In order to retrofit MTCP for use by DMTCP, we added 3 hooks into MTCP. These hooks are initialized to function pointers into DMTCP code. This allows the thread of control to be temporarily passed from MTCP to DMTCP. Control is passed to DMTCP:

1. to determine the next time at which to checkpoint;
2. after suspending user threads, before the checkpoint is written; and
3. before resuming user threads (after writing a checkpoint or after a restart).

Additionally, the retrofit disables the checkpointing of open file descriptors by MTCP. Instead, this is re-implemented by DMTCP, which also checkpoints pipes, pseudo-terminals and a variety of socket descriptors. This is necessary, since the wealth of UNIX descriptors in a distributed setting goes far beyond the single process scenario for which MTCP was designed.

By design, MTCP is retrofitted in a limited, relatively non-invasive fashion. This both insulates DMTCP from lower-level concerns, and also provides the flexibility to substitute a different single-process checkpointing package for MTCP.

4.2.2 Initialization of an application process under DMTCP

Our routine `dmtcp_checkpoint` sets `LD_PRELOAD` to `dmtcphiack.so`, the DMTCP library responsible for checkpointing. DMTCP loads `mtcp.so` and calls the MTCP setup routines to enable the hooks described in Section 4.2.1. MTCP creates the checkpoint manager thread in this setup routine. DMTCP also opens a TCP/IP connection to the checkpoint coordinator at this time, as described in Section 3.

The DMTCP library adds wrappers around a small number of `libc` functions. For efficiency reasons, we avoid wrapping any frequently invoked system calls such as `read` and `write`. The wrappers are necessary since DMTCP must be aware of all forked child processes, of all attempts to create remote processes (for example via an `exec` to an `ssh` process), and of the parameters by which all sockets are created. In the case of sockets, DMTCP needs to know whether the sockets are TCP/IP sockets (and whether they are listener or non-listener sockets), UNIX domain sockets, or pseudo-terminals.

UDP domain sockets are currently not handled, since they are rarely used in computation-intensive programs, but they are easy to implement. Similarly, other types of sockets (datagram, raw packets, etc) would be easy to add if required by a scientific application.

DMTCP places wrapper functions around the functions: `socket`, `connect`, `bind`, `listen`, `accept`, `setsockopt`, `fexecve`, `execve`, `execv`, `execvp`, `fork`, `close`, `dup2`, `socketpair`, `openlog`, `syslog`, `closelog`, `ptsname` and `ptsname_r`. The rest of this section describes the purposes for these wrapper.

Wrappers are placed around `dup2` and `close` to prevent the application from closing DMTCP owned sockets and files. This protects us against some applications that blindly close unused file descriptors. (Many applications do this to get a “clean slate” prior to an `exec`.)

We place a wrapper around `fork` in order to initialize the child process in much the same way as the initialization routine of `dmtcphiack.so`. This is needed because the environment variable `LD_PRELOAD` has no effect in the context of a forked child process.

Although the `exec` functions `fexecve`, `execve`, `execv`, `execvp` will cause invocation the initialization routine of `dmtcphiack.so`, this does not suffice. One also needs to transfer information about open file descriptors to the new program. This is done by writing this information to a file in `/tmp/` and storing the path of this file in an environment variable. Exec wrappers also ensure that DMTCP specific environment variables have not been changed or removed by the user program. (Some applications attempt to reset all environment variables before an `exec`.)

A special case occurs when `ssh` is checkpointed. The `ssh` servers themselves are not checkpointed. Instead, the initialization routine detects invocations of `ssh` and instead mutates the command line to

include checkpointing of the remote process. The rewritten command line will be of the form:

```
ssh host dmtcp_checkpoint ...
```

Wrappers around `ptsname` and `ptsname_r` are used to virtualize pseudo-terminals under Linux. The master device of a pseudo-terminal is referred to by a file descriptor, and can be restored directly by the algorithm of Section 4.2.4. The slave device is a filename. Upon restart, there could be a conflict of the slave device filename with an already existing pseudo-terminal. To virtualize this, at the first call to `ptsname` by the application, we create a symbolic link in the `/tmp` directory to the actual slave device, and the wrapper function returns the name of this symbolic link. Upon restart, we create a new pseudo-terminal and adjust the symbol links accordingly.

Finally wrappers are placed around `socket`, `bind`, `listen`, and `setsockopt` in order to locally record within *dmtcphi*.*jack.so* whether a listener or non-listener socket is being created, the status of the listener socket, and similar information that will be needed at the time of restart. (See Section 4.2.4.)

The wrappers around `connect`, `accept` and `socketpair` allow the DMTCP routines in the two application processes of a new connection to identify each other. The connecting process sends a one-way handshake containing a globally unique socket ID to the accepting process. The socket ID is associated with the underlying socket, and not with the particular socket descriptor. At restart time, the accepting process will use this socket ID to discover the original connecting process that sent it, and to therefore reestablish the connection prior to turning control over to the application. (See Section 4.2.4.)

Wrappers around `pipe` allow DMTCP to discover all pipes, and to promote them to socket pairs (as if created by `socketpair`) at restart time. This enhances the ability to migrate single processes to a remote hosts.

4.2.3 Checkpointing under DMTCP

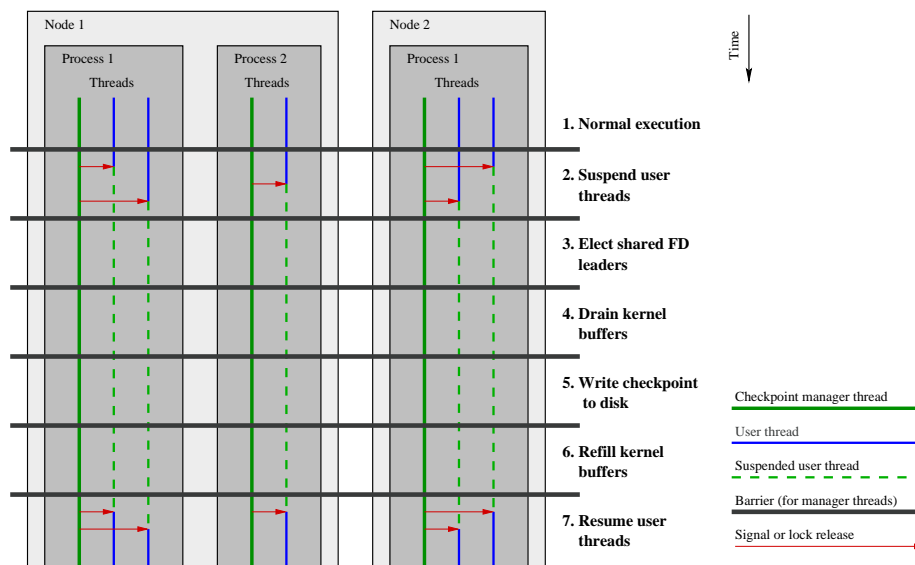


Figure 1: Steps for checkpointing a simple system with 2 nodes, 3 processes, and 5 threads.

The following is the DMTCP distributed algorithm for checkpointing an entire cluster. It is executed asynchronously in each user process. The only communication primitive used is a cluster-wide barrier. These barriers are currently implemented using the DMTCP coordinator. Efficient distributed algorithms for barriers are well known, therefore the coordinator could theoretically be eliminated if it ever became a bottleneck. The following steps are depicted graphically in Figure 1.

1. *Normal execution:*

- The checkpoint manager thread in each process waits until a new checkpoint is requested by the coordinator. This is done by waiting at a special barrier that is not released until checkpoint time.

2. *Suspend user threads:*

- MTCP suspends all user threads.
- DMTCP saves the owner of each file descriptor via the call `fcntl(fd, F_GETOWN)`. In Step 3 the owner flag will be changed for purposes of leader election. In Step 6 the original owner will be restored.
- Wait until all application processes reach Barrier 2: “suspended”. Then release the barrier.

3. *Elect shard FD leaders:*

- DMTCP executes an election of a leader for each potentially shared file descriptor. We trick the operating system into electing a leader for us by misusing the `F_SETOWN` flag of `fcntl`. Each application process on a host calls `fcntl(fd, F_SETOWN, getpid())` for each socket. The last process to make this call for a particular socket shared wins the election for that socket. In Step 4, a process can test if it is the election leader for a socket `fd` by testing if `fcntl(fd, F_GETOWN)==getpid()` returns true.
- Wait until all application processes reach Barrier 3: “election completed”. Then release the barrier.

4. *Drain kernel buffers:*

- For each socket, the corresponding election leader flushes that socket by sending a special token. The election leader then drains that socket by receiving until there is no more available data and the special token is seen.
- The *connection information table* is written to disk. This includes all meta-information associated with each file descriptor including a *globally unique ID* (hostid, pid, timestamp, per-process connection number) that can be used at restart time to detect shared sockets.
- Wait until all application processes reach Barrier 4: “drained”. Then release the barrier. (Note that this barrier is not required but is useful for debugging.)

5. *Write checkpoint to disk:*

- The contents of all socket buffers is now in user space. MTCP writes all of user space memory to a checkpoint file of type `.mtcp`.
- Wait until all application processes reach Barrier 5: “checkpointed”. Then release the barrier.

6. *Refill kernel buffers:*

- DMTCP sends the drained socket buffer data back to the sender. The sender refills the socket buffers.
- Wait until all application processes reach Barrier 6: “refilled”. Then release the barrier.

7. *Resume user threads:*

- MTCP resumes the application threads.
- Return to Step 2

Recall that Section 4.2.1 describes several hook functions implemented by DMTCP and added to MTCP. Barrier 1 is implemented in the DMTCP hook function that determines the next time to checkpoint. Barriers 2, 3 and 4 are implemented in the DMTCP hook function executed prior to checkpoint. Barriers 5 and 6 are called in the DMTCP hook function executed just after a checkpoint.

4.2.4 Restart under DMTCP

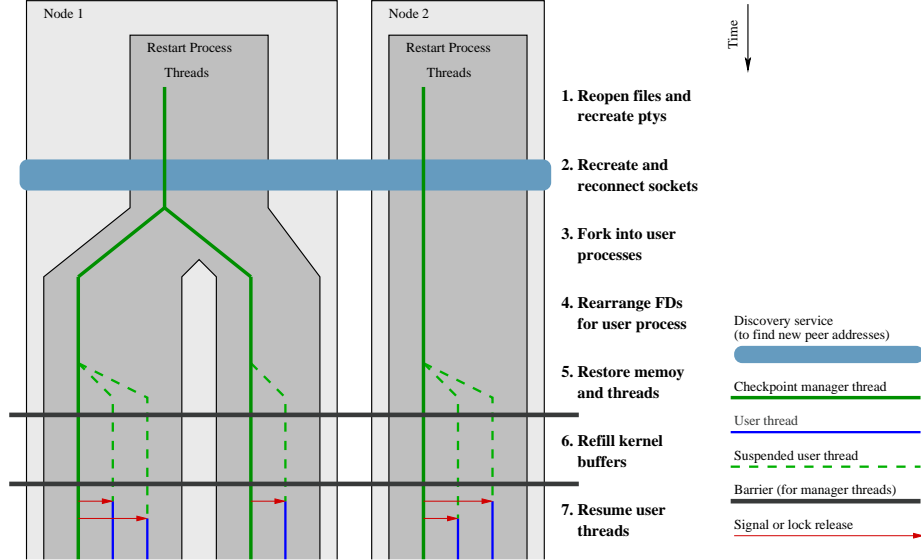


Figure 2: Steps for restarting the system checkpointed in Figure 1. The unified restart process and subsequent fork are required to recreate sockets and pipes shared between processes.

The restart process undergoes some complexity in order to restore shared sockets. Under UNIX semantics multiple processes may share a single socket connection. When a process forks all open file descriptors become shared between the child and parent. To handle this, we refer to sockets by a *globally unique ID* (hostid, pid, timestamp, per-process connection number) and thus can detect duplicates at restart time. These globally unique socket IDs (and other meta information), were recorded at checkpoint time in the *connection information table* for each process. To recreated shared sockets, a *single* DMTCP restart process is created on each host. This single restart process will first restore all sockets, and then fork to create each individual user process on that host.

The following algorithm restarts the checkpointed cluster computation. It is executed asynchronously on each host in the cluster. The steps of this algorithm are depicted graphically in Figure 2.

1. Reopen files and recreate ptys:

- File descriptors, excluding sockets connected to a remote process, are regenerated first. These include files, listen sockets, uninitialized sockets, and pseudo-terminals.

2. Recreate and reconnect sockets:

- For each socket, the restart program uses the cluster-wide discovery service to find the new address of the corresponding restart process. Once the new addresses are found the connections are re-established. The discovery services is needed since processes may be relocated between checkpoint and restore.

3. *Fork into user processes:*

- The DMTCP restart program now forks into N processes, where N is the number of user processes it intends to restore.

4. *Rearrange FDs for user process:*

- Each of these processes uses `dup2` and `close` to re-arrange the file descriptors to reflect the arrangement prior to checkpointing. Unneeded file descriptors belonging to other processes are closed. Shared file descriptors will now exist in multiple processes.

5. *Restore memory and threads:*

- The MTCP restart routine is now called to restore the local process memory and threads. Upon completion the user process will resume at Barrier 5 of the checkpoint algorithm in Section 4.2.3

6. *Refill kernel buffers:*

- The program resumes as if it had just finished writing the checkpoint to disk, in Step 6 of checkpointing.

7. *Resume user threads:*

- The program continues executing Step 7 of checkpointing.

Step 2 above bears further explanation. Recall that prior to checkpointing, whenever a new connection was accepted, wrappers around the system calls `connect` and `accept` had transferred information about the `connector` to the `acceptor`. This information includes a globally unique socket ID that remains constant even if processes are relocated.

At restart time, the `acceptor` for each socket advertises the address and port of its restart listener socket to the discovery service. When the `connector` receives this advertisement, it opens a new connection to the `acceptor` who sent the advertisement. The two sides then perform a handshake and agree on the socket being restored. Finally, `dup2` is used on each side to move the socket descriptor to the correct location. This process continues asynchronously until all sockets are restored. Our methodology supports both sides of a socket migrating. It also supports loopback sockets.

5 Experimental Results

DMTCP is currently implemented for GNU/Linux. The software has been verified to work on recent versions of Ubuntu, Debian, OpenSuse, and Red Hat Linux with Linux kernels ranging from version 2.6.9 through version 2.6.24. DMTCP runs on both x86 and x86_64 architectures. The experimental results were conducted under Red Hat Enterprise Linux AS release 4 (Linux 2.6.9-34.ELsmp, 64-bit). This is installed as part of Rocks release 4.1.1. The cluster was connected with Gigabit Ethernet. Tests are performed on a cluster of 32 dual-processor, dual-core (4 cores per node) Xeon 5130 machines running at 2 GHz. Each node had either 8 or 16 GB of RAM.

In Figure 4b, checkpoints were written to a centralized EMC CX300 SAN storage device over a 4 Gbps Fibre Channel Switch. (SAN stands for storage area network.) On our cluster, only 8 of the 32 nodes were connected to the SAN. The remaining 24 nodes wrote indirectly to the storage device via NFS. In all other tests, checkpoints were written to local disk of each node.

We report checkpoint times, restart times, and checkpoint file sizes for a suite of applications with broad coverage. These are contained in Figures 3a, 3b and 3c, respectively. In each case, we report the timings and file sizes both with and without compression. The following applications are shown:

- **Based on sockets directly:**

- **iPython:** [3] an enhanced Python shell with support for parallel and distributed computations. Used in scientific computations such as SciPy/NumPy [36]. **iPython/Shell:** is the interactive iPython interpreter, idle at time of checkpoint. **iPython/Demo:** is the “parallel computing” demo included with the iPython tutorial.

- **Run using MPICH2:**

- **Baseline** is a “hello world” type application included to show the cost of checkpointing MPICH2 and its resource manger, MPD.
- **ParGeant4:** Geant4 [37] is a million-line C++ toolkit for simulating particle-matter interaction. It is based at CERN, where the largest particle collider in the world has been built. ParGeant4 [38] is a parallelization based on TOP-C, that is distributed with the Geant4 distribution. TOP-C (Task Oriented Parallel C/C++) was in turn built on top of MPICH2 for this demonstration.
- **NAS NPB2.4:** CG (Conjugate Gradient, level C) from the well-known benchmark suite NPB. NPB 2.4-MPI was used.

- **Run using OpenMPI:**

- **Baseline** is a “hello world” type application included to show the cost of checkpointing OpenMPI and its resource manger, OpenRTE.
- **NAS NPB2.4:** a series of well-known MPI benchmarks. NPB 2.4-MPI was used. The benchmarks run under OpenMPI are: BT (Block Tridiagonal, level C: 36 processes since the software requires a square number), SP (Scalar Pentadiagonal, level C: 36 processes since the software requires a square number), EP (Embarassingly Parallel, level C), LU (Lower-Upper Symmetric Gauss-Seidel, level C), MG (Multi Grid, level C), and IS (Integer Sort, Level C).

In Figure 4a we use ParGeant4 as a test case to report on scalability with respect to the number of nodes. When resource management processes are included, we are checkpointing a total of 289 processes in the largest example. Figure 4b repeats this tests with checkpoints written to a centralized storage device.

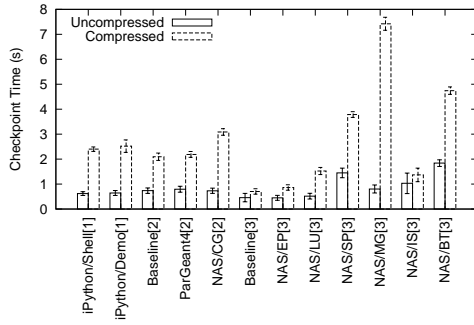
Finally, Figure 5 illustrates the time as memory usage grows, while holding fixed the number of participating nodes at 32. The implied bandwidth is well beyond the typical 100 MB/s of disk, and is presumably indicating the use of secondary storage cache in the Linux kernel. Restart times also indicate the use of cache and page table optimizations in the kernel.

An optional feature in DMTCP is to issue a **sync** after checkpointing to wait for kernel write buffers to empty before resuming the user threads. Results shown do not issue a call to **sync**. This is consistent with timing methodology most prevalent in related work. The cost of issuing a **sync** can be easily estimated based on checkpoint size and disk speed. As an example, if a **sync** is issued for ParGeant4 (compression enabled) a mean additional cost of 0.79 seconds (with a standard deviation of 0.24) is incurred. An alternate strategy is to sync the *previous* checkpoint instead. This has the benefits of still guaranteeing the consistency of all except the last checkpoint without having to wait for disk in most cases.

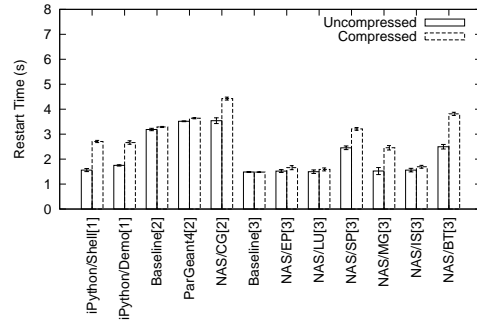
5.1 Experimental Analysis

In principle, the time for checkpointing is dominated by: (i) compression (when enabled); (ii) checkpointing memory to disk; and (iii) flushing network data in transit and re-sending. When compression is enabled, that time dominates. The cost of flushing and re-sending is bounded above by the size of the corresponding kernel buffers and the capacity of the network switches, which tend to be on the order of tens of kilobytes.

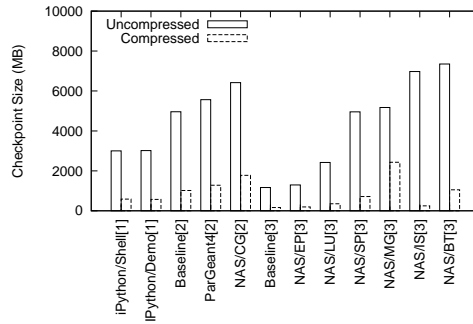
The graphs in Figure 3 show that the time to checkpoint using compression tends to be slowest when the uncompressed checkpoint image is largest. An exception occurs for NAS/IS. NAS/IS is a parallel integer bucket-sort. The bucket sort code has allocated large buckets to guard against overflow. Presumably, the unwritten portion of the bucket is likely to be mostly zeroes, and it compresses both quickly and efficiently.



(a) Checkpoint timings.

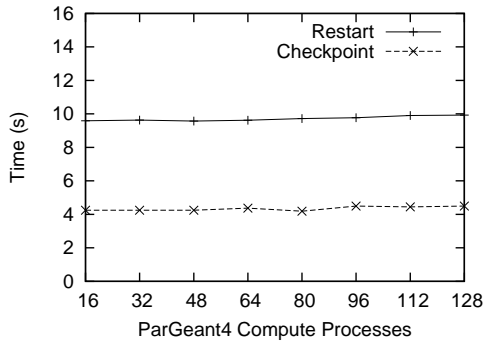


(b) Restart timings.

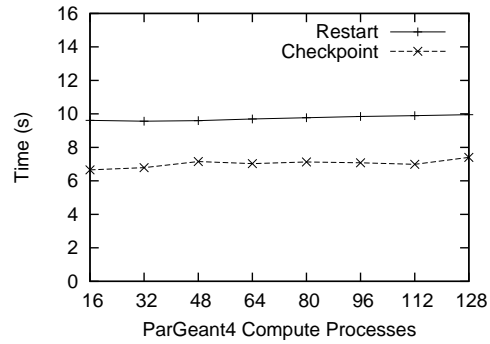


(c) Aggregate (cluster-wide) checkpoint size.

Figure 3: Timings on 32 nodes. Applications marked [1] use sockets directly. Applications marked [2] are run using MPICH2. Applications marked [3] are run using OpenMPI. Timing tests repeated 10 times, mean value is shown. Error bars in timings indicate plus or minus one standard deviation.



(a) Checkpoints stored to local disk of each node.



(b) Checkpoints stored to centralized RAID storage using a SAN and NFS.

Figure 4: Timings as the number of processes and nodes changes. Application is ParGeant4 running under MPICH2. Compression is enabled. Compute processes per core and per node are held constant at 1 and 4, number of nodes is varied. (Note: An additional 21 to 161 MPICH2 resource management processes are also checkpointed.)

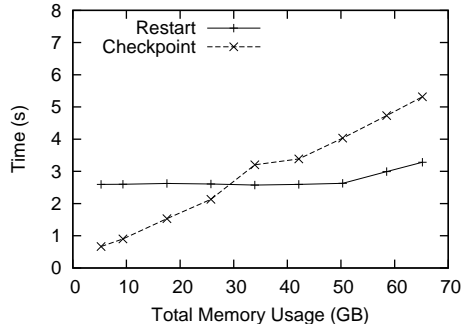


Figure 5: Timings as memory usage grows. A synthetic OpenMPI program allocating random data on 32 nodes. Compression is disabled. Checkpoints written to local disk.

The time to checkpoint without compression is less than 2 seconds in all cases. As one would expect, decompression (gunzip) is relatively quick, and so restart operates in only 1 to 4 seconds in all cases (Figure 3b).

Figure 4 shows the time for checkpoint and restart to be insensitive to the number of nodes being used. This is to be expected since checkpointing on each node occurs asynchronously. It also demonstrates that the single checkpoint coordinator, which implements barriers, is not a bottleneck. In the event that it were a bottleneck, we would replace it by a distributed coordinator in our implementation.

Figure 5 shows that the checkpoint time increases gradually with process image size. Kernel level caching and optimizations results in little increase in restart time until each node utilizes over 1.5 GB of data. A large contiguous read is a relatively easy operation for the kernel to optimize, since it can play tricks with page tables to avoid needing to actually copy the data.

6 Conclusions and Future Work

We have demonstrated a scalable approach to transparent distributed checkpointing that does not depend on a specific message passing library. Nor does it depend on kernel modification. We have demonstrated broad coverage across across a wide array of scientific applications. Experimental results have shown that our approach is scalable and that timings remain nearly constant as nodes are added to a computation. Our centralized checkpoint coordinator, which implements barriers, has not yet been shown to be a bottleneck. As the algorithm is scaled further, the single coordinator can be replaced by well-known algorithms for distributed global barriers and distributed discovery services.

In the future, we hope to support new communication models such multicast and RDMA (remote direct memory access) as used in networks such as InfiniBand. We can additionally add support for less common transport layers such as UDP. Another interesting future direction is to integrate with batch queuing systems for automatic migration and for restarting distributed computations automatically when failures occur.

7 Acknowledgements

We thank Alex Brick for programming a compress/decompress filter for MTCP based on the use of gzip, and for careful reading of a draft manuscript. We also thank Xin Dong for his help in installing and testing ParGeant4 in a variety of operating circumstances. We also acknowledge the gracious help of Michael Rieker in numerous discussions, and in fixing more than one bug in MTCP for us. We gratefully acknowledge the assistance of Chintan Shah in testing several intermediate versions of MTCP and DMTCP. We thank Sriram Venkataramani for tracking down the `-fno-stack-protector` bug in MTCP. We thank William Thies, Marek Olszewski, and David Wentzlaff for providing valuable feedback on draft manuscripts. Finally, we acknowledge helpful discussions with Abyd Al Zain, Kevin Hammond, Jiri Schindler, and Steve Linton.

References

- [1] Top500 supercomputing sites. <http://www.top500.org/>.
- [2] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical report 1346, University of Wisconsin, Madison, Wisconsin, April 1997.
- [3] Fernando Perez and Brian E. Granger. ipython: A system for interactive scientific computing. *Computing in Science and Engineering*, pages 21–29, May/June 2007. (See, also, http://ipython.scipy.org/moin/Parallel_Computing).
- [4] Ron Choy and Alan Edelman. Parallel matlab: Doing it right. In *Proceedings of the IEEE*, pages 331–341, 2005.
- [5] N. Doss, W. Gropp, R. Lusk, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996. software at <http://www-unix.mcs.anl.gov/mpi/mpich/>.
- [6] Richard L. Graham, Timothy S. Woodall, and Jeffrey M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.
- [7] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications*, 8(3/4), 1994. (special issue on MPI; also see <http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html>, and <http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html>).
- [8] Michael Rieker, Jason Ansel, and Gene Cooperman. Transparent user-level checkpointing for the Native POSIX Thread Library for Linux. In *Proc. of Parallel and Distributed Processing Techniques and Applications (PDPTA-06)*, pages 492–498, 2006.
- [9] Philip Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proceedings of the 2003 Linux Symposium*, Ottawa, Canada, 2003.
- [10] William R. Dieter and James E. Lupp Jr. User-level checkpointing for LinuxThreads programs. In *USENIX Annual Technical Conference (FREENIX Track)*, pages 81–92, 2001.
- [11] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Proc. of the USENIX Winter 1995 Technical Conference*, pages 213–323, 1995.
- [12] Eduardo Pinheiro. EPCKPT — a checkpoint utility for the Linux kernel. <http://www.research.rutgers.edu/~edpin/epckpt/>.
- [13] J. S. Plank, J. Xu, and R. H. B. Netzer. Compressed differences: An algorithm for fast incremental checkpointing. Technical Report CS-95-302, University of Tennessee, August 1995.
- [14] Kai Li, Jeffrey F. Naughton, and James S. Plank. Real-time, concurrent checkpoint for parallel programs. In *Proc. of Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, March 1990.
- [15] Kai Li, Jeffrey F. Naughton, and James S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5:874–879, August 1994.
- [16] P. Emerald Chung, Woei-Jyh Lee, Yennun Huang, Deron Liang, and Chung-Yih Wang. Winckp: A transparent checkpointing and rollback recovery tool for Windows NT applications. In *Proc. of 29th Annual International Symposium on Fault-Tolerant Computing*, pages 220–223, 1999.

- [17] Hazim Abdel-Shafi, Evan Speight, and John K. Bennett. Efficient user-level thread migration and checkpointing on Windows NT clusters. In *Usenix 1999 (3rd Windows NT Symposium)*, pages 1–10, 1999.
- [18] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose 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.
- [19] Thomas Herault, Pierre Lemarinier, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi. In *Proceedings of International Symposium on High Performance Computing and Networking (SC2006)*, 2006.
- [20] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdain. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS) / 12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*. IEEE Computer Society, March 2007.
- [21] Youhui Zhang, Dongsheng Wong, and Weimin Zheng. User-level checkpoint and recovery for LAM/MPI. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 72 – 81, 2005.
- [22] Gengbin Zheng, Lixia Shi, and L.V. 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.
- [23] Georg Stellner. Cocheck: Checkpointing and process migration for MPI. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.
- [24] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *ACM/IEEE 2002 Conference on Supercomputing*. IEEE Press, 2002.
- [25] Namyoon Woo, Soonho Choi, hyungsoo Jung, Jungwhan Moon, Heon Y. Yeom, Taesoon Park, and Hyungwoo 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.
- [26] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 77–83, New York, NY, USA, 2002. ACM Press.
- [27] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul 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.
- [28] G.J. Janakiraman, J.R. Santos, D. Subhraveti, and Y. Turner. Application-transparent distributed checkpoint-restart on standard operating systems. In *Dependable Systems and Networks (DSN-05)*, pages 260–269, 2005.
- [29] Oren Laadan, Dan Phung, and Jason Nieh. Transparent networked checkpoint-restart for commodity clusters. In *2005 IEEE International Conference on Cluster Computing*. IEEE Press, 2005.
- [30] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes for commodity clusters. In *2007 USENIX Annual Technical Conference*, pages 323–336, 2007.

- [31] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA '02: Proceedings of the 29th annual International Symposium on Computer Architecture*, pages 123–134, Washington, DC, USA, 2002. IEEE Computer Society.
- [32] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
- [33] K.A. Iskra, F. van der Linden, Z.W. Hendrikse, B.J. Overeinder, G.D. van Albada, and P.M.A. Sloot. The implementation of Dynamite — an environment for migrating PVM tasks. *Operating Systems Review*, 34:40–55, July 2000.
- [34] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast transparent migration for virtual machines. In *Proceedings of USENIX 2005 Annual Technical Conference*, pages 391–394, 2005.
- [35] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. of 5th Symposium on Operating Systems Design and Implementation (OSDI-2002)*, 2002.
- [36] SciPy and NumPy. <http://scipy.org/>.
- [37] Geant4 Web page. <http://wwwinfo.cern.ch/asd/geant4/geant4.html>, 1999–.
- [38] 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.