DMTCP and Condor: a New Checkpointing Mechanism

Gene Cooperman (presenting)
High Performance Computing Laboratory
College of Computer and Information Science
Northeastern University, Boston
gene@ccs.neu.edu

Joint work with:

Kapil Arya, Northeastern University
Peter Keller, Condor Project, U. of Wisconsin-Madison
Artem Y. Polyakov, Siberian State U. of Telecom.
and Informatics, Russia

DMTCP Overview

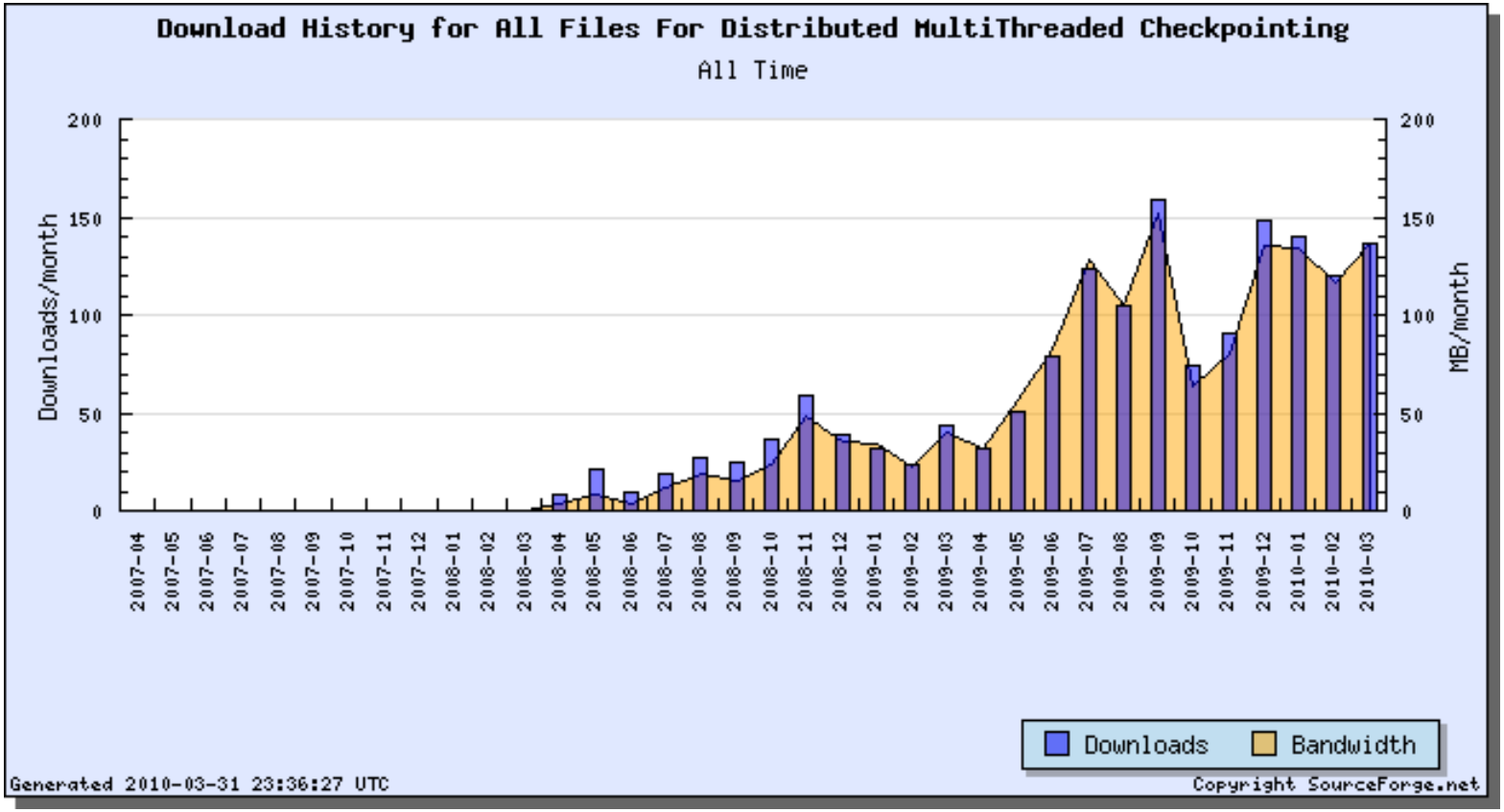
DMTCP (Distributed MultiThreaded CheckPointing):

- *Mature:* five years in development
- *Robust:* current user base in hundreds and growing
- *Non-invasive:* no root privilege needed; no kernel modules; transparently operates on binaries, no application source code needed
- *Fast:* checkpoint/restart in less than a second (dominated by disk time)
- *Versatile:* works on OpenMPI, MATLAB, Python, bash, gdb, X-Windows apps, etc.
- *Open Source:* freely available at `http://dmtcp.sourceforge.net` (LGPL)

STANDALONE USAGE:

```
dmtcp_checkpoint a.out
dmtcp_command --checkpoint
dmtcp_restart ckpt_a.out_*.dmtcp
```

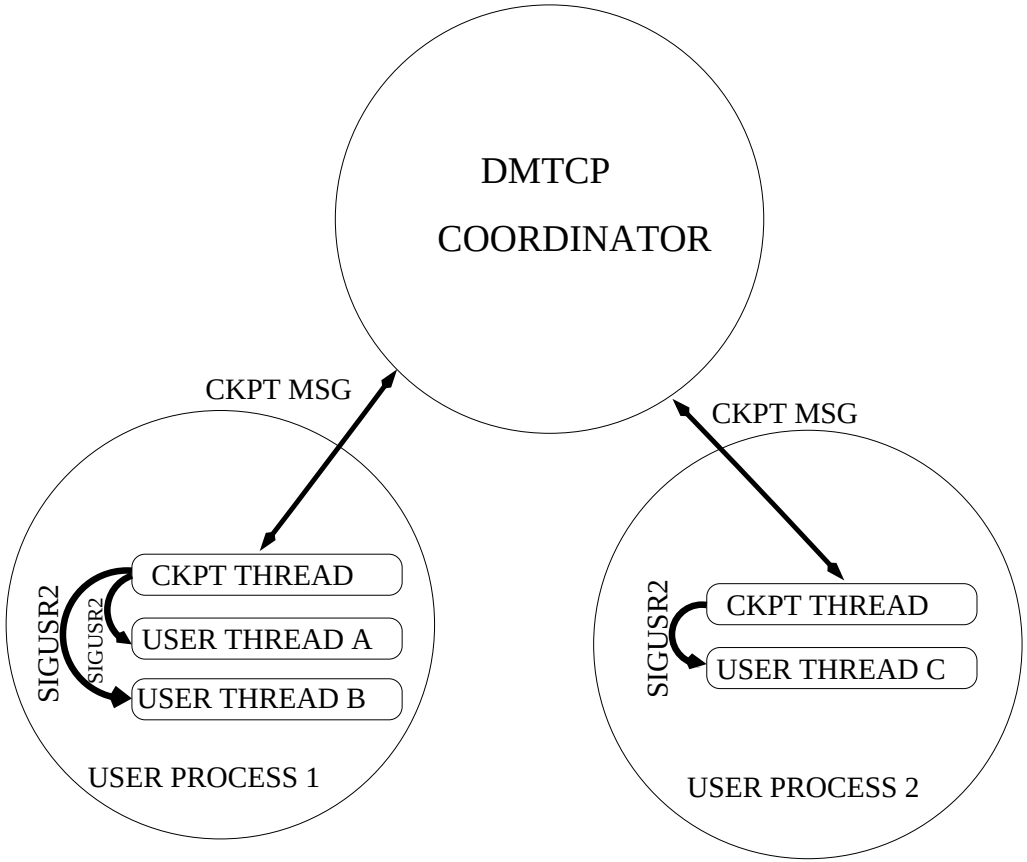
DMTCP Downloads



DMTCP is currently averaging over 100 downloads per month.

DMTCP: How Does It Work?

Provides fast checkpoint-restart (typically less than a second)
10 MB checkpoint typical (based on footprint in RAM)
Dynamic compression of checkpoint images (enabled by default)



DMTCP Features

- Distributed MultiThreaded CheckPointing
- Works with Linux kernel 2.6.9 and later
- Supports sequential and multi-threaded computations across single/multiple hosts
- Entirely in user space (no kernel modules or root privilege)
- Transparent (no recompiling, no re-linking)
- DMTCP Team centered around Northeastern U., with collaborators from MIT and Siberian State U. of Telecom. and Informatics
- LGPL, freely available from sourceforge
- No Remote I/O (except through certain extensions)

Some DMTCP Features Relevant to Condor

- Multiple processes allowed: fork() is supported
- Multiple threads allowed: POSIX Threads is supported
- Calls to mmap() are supported
- No need to re-link : Original binary is supported

DMTCP from the Desktop


```
dmtcp_checkpoint a.out     # starts coordinator too
dmtcp_command --checkpoint # talks to coordinator
dmtcp_restart ckpt_a.out-*.dmtcp
```

- Coordinator is a *stateless* synchronization server for the distributed checkpoint algorithm
- Essentially zero run-time overhead
- Checkpoint/Restart performance depends on size of memory, disk write speed (e.g., 100 MB/s), and network latency

DMTCP Stateless Coordinator

- The DMTCP coordinator is *stateless*. Its primary responsibilities are:
  1. Allow DMTCP checkpoint thread of user process to register with the coordinator.
  2. Relay commands (or timed checkpoints) back to user processes.
  3. Provide a distributed barrier for stages of checkpoint and restart.
  4. Provide a distributed nameserver to connect both ends of sockets at restart time.

What happens if one of my processes dies?
ANSWER: Tell the coordinator to kill the other processes
(e.g., `dmtcp_command --kill`*), and then restart.*

What happens if the coordinator dies?
ANSWER: Kill the rest of the process, and then restart.

My processes died. Do I need to kill the DMTCP coordinator to restart?
ANSWER: It's optional. The dmtcp_restart command will look for an existing coordinator, and if none exists, it will start a new one.

War Stories: Interesting Interactions Between DMTCP and Condor

While simple examples of integration of DMTCP with Condor worked from the beginning, there have been many details in getting DMTCP "ready for prime time".

- Blocking checkpoint — checkpoint request does not return until checkpoint image has been fully written out
- Uniquely named jassert trace files — useful for diagnosis
- `DMTCP_TMPDIR` environment variable added, since jassert files are written to `/tmp`, and Condor needs more flexibility.
- Environment variable `DMTCP_RESTART_DIR` added
- When a process checkpoints and exist (for example, for process migration), what should be the success return value passed back to Condor? (When DMTCP was running standalone, this didn't matter.)
- Removed a `-Wl,-export-dynamic` in build of DMTCP library, mtcp.so (danger of generating name conflict with application code)

War Stories: Interesting Interactions (cont.)

- `-fno-stack-protector` used in DMTCP build for one DMTCP file — Condor turns on `-fstack-protector` by default, but can coexist with it turned off for this DMTCP file.
- Condor uses `-Wl,--hash-style=both` because ELF format changed at one point; DMTCP needs to be aware of the same issue for migration between nodes that use different ELF formats.
- DMTCP needed to work alongside other applications that also use `LD_PRELOAD`

DMTCP Internals: Startup

- dmtcp_checkpoint sets Linux LD_PRELOAD environment variable to load dmtcphijack.so into all newly created process
- dmtcphijack.so creates an additional thread, DMTCP checkpoint thread, for each process
- DMTCP checkpoint thread connects to DMTCP coordinator, and then waits for further instructions from coordinator
- dmtcphijack.so also creates a wrapper around certain Linux system calls; e.g., clone() (pthread_create), fork(), exec(), open(), bind(), connect(), listen(), gettid(), getpid(), getppdi(), getsid(), and others
- POLICY (zero run-time overhead): dmtcphijack.so never creates a wrapper around read(), write(), or other frequently used system calls
- Remote processes supported: Wrapper for exec
  rewrites calls to `exec("ssh HOST ...")`
  as: `exec("ssh HOST dmtcp_checkpoint ...")`

More War Stories: the Fifteen Bugs from the Condor Test Suite

The following stories are related to DMTCP internals.

- Uncleanly exiting due to race condition: The main user thread was forced by `LD_PRELOAD` to execute a DMTCP C++ constructor. If the main user thread exits before the DMTCP checkpoint thread, then the destructor is called on the DMTCP object. Checkpoint thread is deep in a select() call listening to the coordinator and is unresponsive. The DMTCP checkpoint thread wakes up on process exit, but it may refers to the DMTCP object after the user thread destroyed it. (Solution: Upon exiting, the main user thread sets a global variable, and then sends a signal to the checkpoint thread. The checkpoint thread sees the global variable inside the signal handler, and exits appropriately. Solves four of the 15 bugs.)
- Shared file descriptors were no longer shared on restart in certain cases. (Now more careful about preserving shared file descriptors)
- One of the DMTCP internal environment variables becomes visible to user program upon restart (Solution: DMTCP now cleans up better)

More War Stories: the Fifteen Bugs (cont.)

- dmtcpaware bug — during very frequent checkpoints, dmtcpaware would get confused about whether it had finished an operation upon restart.
- If checkpointing too soon, application can die. (Bug in DMTCP coordinator logic — fixed)
- Address space randomization (Linux security measure) causes the kernel to change where it believes the stack to be upon restart. DMTCP takes special measures to grow the kernel stack on restart to include the old stack. A one megabyte stack in main() caused the stack to sometimes grow beyond what DMTCP was expecting. Currently, DMTCP increased its safety zone to always accomodate one megabyte stacks. Additional measures are planned for the long term.

More War Stories: the Fifteen Bugs (cont. again)

- Bug in wrapper around signals was exposed exposed potential race conditions. Signal wrapper code re-written more carefully.
- Additional race condition: User thread asks for checkpoint before checkpoint thread acknowledges to coordinator that it is now running. (Currently added timeout if coordinator doesn't reply soon enough — longer term measures being considered)

DMTCP Internals: Process Virtualization

- *Process Virtualization:* Persistent operating system constructs, such as process id (pid), must be maintained between checkpoint and restart. DMTCP translation tables between the virtual and real id are used.

I called `getpid()` *before checkpoint, and my process id was 3603. If I stored it in a program variable, will my process id still be 3603 when I use the program variable after restart?*
ANSWER: Yes.

My thread was waiting on a mutex lock before checkpoint. Will it still be waiting on the mutex lock after restart? Will `pthread_mutex_unlock()` *still unlock it using the original pointer to the lock?*
ANSWER: Yes and Yes.

DMTCP Internals: Process Virtualization (cont.)

- Among the Linux constructs supported by DMTCP are: fork, exec, ssh, mutexes/semaphores, TCP/IP sockets, UNIX domain sockets, pipes, ptys (pseudo-terminals), fifos (named pipes), terminal modes, ownership of controlling terminals, signal handlers, open file descriptors, shared open file descriptors, I/O (including the readline library), shared memory (via mmap), parent-child process relationships, pid and thread id virtualization
- Persistent Linux identifiers supported include: process ids (pids), thread ids (tids), process group ids (pgids), session ids (sids), parent process id (ppid, parent-child relationship), pseudo terminals (ptys, such as /dev/ptmx, /dev/pts/1, etc.; or the older BSD-style ptys), named pipes (fifos), listener ports of servers, the socket address of a peer on the network, shared memory buffer pools (mpool, non-POSIX, but present in Linux and BSD), System V IPC objects (shared memory, message queues, and semaphores), POSIX shared memory (shm), and mmap (mapping between virtual memory and physical backing file at a well-known address).

DMTCP Odds and Ends

- At checkpoint time, a checkpoint image, ckpt_⟨filename⟩_⟨global_id⟩.dmtcp, for each process is created in the current working directory on the host of the process: e.g., `ckpt_a.out_5f2439cb-11756-4bc1ce3e.dmtcp`
- global_id is globally unique to support process migration.
- The coordinator generates `dmtcp_restart_script.sh` as a convenient way of restarting all checkpoint images on all hosts. It is easily modified to restart on different hoss, etc.
- Checkpointing under program control is supported: dmtcpaware interface
- Currently, only supports dynamic linking to libc.so. Support for static libc.a is feasible, but not implemented.
- Runtime libraries (including libc.so) are saved as part of the checkpoint image.
- Restarting on different Linux distributions and different Linux kernels usually works, but further work is needed to support robust heterogeneity among Linux distributions.

Condor/DMTCP Integration

- Undergoing validation with Condor checkpointing test suite. Passes most tests now; anticipated to pass all tests soon (within a month??).
- DMTCP is completely outside of Condor source code.
  - A vanilla job called "shim_dmtcp" that wraps the use's job and stdfiles with DMTCP.
  - A submit description file which transfers needed dmtcp files over to the remote side and saves intermediate checkpoints.
  - No remote I/O!
- `condor_starter` calls `shim_dmtcp` which then starts the `dmtcp_coordinator` and user job under control of DMTCP. Additional processes and threads created by the user remain under DMTCP checkpoint control.

Submit File Example

```
universe = vanilla
executable = shim_dmtcp
arguments = logfile stdinf stdoutf stderrf a.out arg0 ...

should_transfer_files = YES
when_to_transfer_output = ON_EVICT_OR_EXIT
transfer_input_files = <dmtcp libraries and programs>,\
           a.out, stdinf, stdoutf, stderrf

environment = DMTCP_TMPDIR=./;JALIB_STDERR_PATH=/dev/null
kill_sig = 2
output = shim.(Cluster).(Process).out
error = shim.(Cluster).(Process).err
log = shim.log
queue
```

Future Condor Integration

- Add `WantCheckpoint = True` and `CheckpointMethod = DMTCP` for a vanilla universe job.
- Condor takes care of the wrapping of the job with DMTCP and transfer of needed DMTCP files — no shim script voodoo.
- Condor should honor `CheckpointPlatform` for Vanilla universe jobs in case of pool segmentation.
- Parallel universe support with single coordinator.
- Doug Thain's Parrot for remote I/O.

Plans for Testing of DMTCP

Because DMTCP began life checkpointing on the desktop with a simple plan: *Use* `make check` *to catch simple bugs, and rely on a growing user base to help find further bugs.*
Good News: The process virtualization approach provides flexibility.
Bad News: Process virtualization means that bugs can come from anywhere (interaction with kernel, compiler, linker, libc, etc.) It's similar to the issue of test suite for a compiler, an operating system kernel, or a virtual machine.

- Use NMI build and test lab for testing on many Linux distributions.
- Add "real-world" programs for standard testing. (Contributions, anyone?)
- Continue using user base: *Please keep those bug reports coming.* (see next slide)
- Wiki of Known Problems and Solutions (to replace the overgrown DMTCP QUICK-START file)

The DMTCP User Base: the Ultimate Test Lab

Please keep those bug reports coming. A warm and friendly welcome is promised. Places to report bugs:

- `dmtcp-forum@lists.sourceforge.net`
- IRC channel: `dmtcp`
- e-mail directly to developers at dmtcp.sourceforge.net (current team of six active developers, each with his or her own specialty)

Many methodologies used for handling bugs (according to user preference):

1. Can you generate DMTCP jassert logs, and send us the logs?
2. Can we test your code on our machines? (First, we try to diagnose bug with binary; if no success then with source.)
3. Can we talk on the IRC channel?
4. Can we talk on the phone while watching you demonstrate the bug using VNC (virtual network client) in read-only mode?

The DMTCP User Base: Do It Yourself Debugging

If you prefer to take a quick look for the bug by yourself before calling us in on it, some strategies are:

1. In DMTCP: `./configure --enable-debug ; make clean ; make`
   Re-execute your code, and look for the `/tmp/jassert.log.XXXX` files. There will be one file for each process.
2. In DMTCP: `./configure --disable-pid-virtualization; make clean; make`
   Re-execute your code. If you code does not depend on remembering the process id (pid), thread id (tid), etc., then you don't need the DMTCP pid virtualization. See if this module is the cause of your bug.
3. In DMTCP: `./configure --enable-allocator; make clean; make`
   Use a simple (inefficient) DMTCP allocator which is guaranteed not to call mmap(), malloc(), and free(), to see if it is a memory allocation issue.
4. For the adventurous: examine `/proc/⟨PID⟩/maps` for your process id (PID) and see if the same memory segments appear on restart as had existed originally.
5. Try: `.../dmtcp/mtcp/readmtcp YOUR_CKPT_IMAGE`
   to see if the memory segments in the checkpoint image agree with `/proc/⟨PID⟩/maps`

Further Reading

- "DMTCP: Transparent Checkpointing for Cluster Computation and the Desktop", Jason Ansel, Kapil Arya, and Gene Cooperman
  - 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09), 2009
  - Technical Report: `http://arxiv.org/abs/cs/0701037`
- Source code and further information:
  - `http://dmtcp.sourceforge.net`

Questions

- DMTCP
  - `http://dmtcp.sourceforge.net` : `dmtcp-forum@lists.sourceforge.net`
  - Gene Cooperman : `gene@ccs.neu.edu`
- Condor/DMTCP Integration
  - Pete Keller: `psilord@cs.wisc.edu`
  - Ask me if you want to try the Alpha Version out!