

# Symbolic Computation using Disk-Based Parallel Computing

Gene Cooperman  
gene@ccs.neu.edu

*High Performance Computing Laboratory*

Kapil Arya  
Xin Dong  
Dan Kunkle  
Vlad Slavici  
Ana Visan

Northeastern University, Boston



## First, a Brief Discussion of Checkpoint-Restart

---

*Tech. Report:* <http://arxiv.org/abs/cs.DC/0701037>

by J. Ansel, K. Arya and G. Cooperman (version to appear at IPDPS-09)

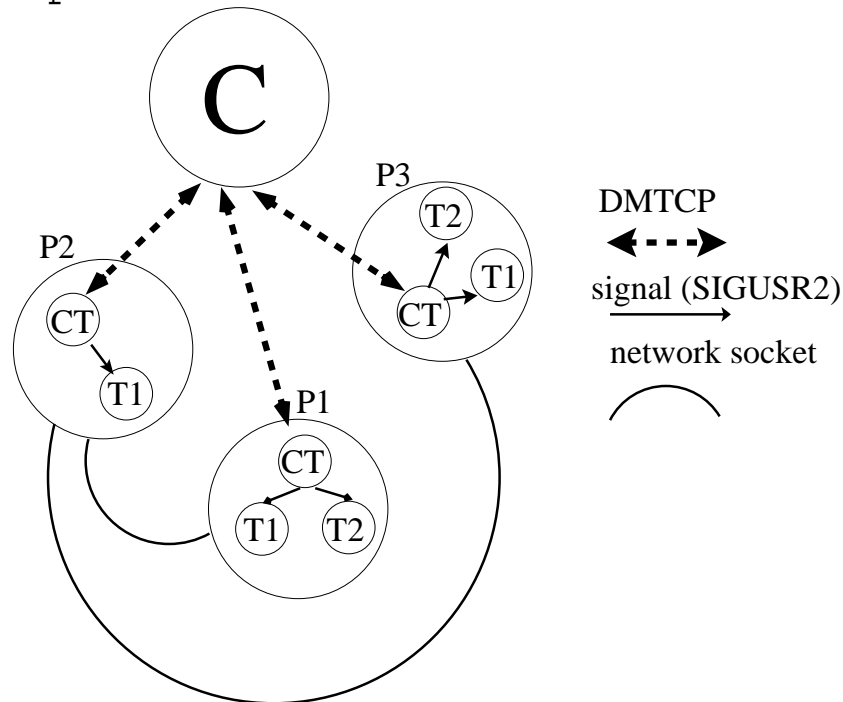
*Open Source:* <http://dmtcp.sourceforge.net>

*SVN:* `svn co https://dmtcp.svn.sourceforge.net/svnroot/dmtcp dmtcp`

# How Does it Work?

```
./dmtcp_checkpoint a.out
./dmtcp_command -c      # -c is checkpoint command
./dmtcp_restart ckpt_a.out_*.dmtcp
```

**C** = DMTCP coordinator process  
**P** = user process  
**CT** = DMTCP checkpoint thread  
**T** = user thread



- Checkpoint time:  $\approx 2$  seconds; Restart time:  $\approx 1$  second
- Run-time overhead: negligible
- Coordinator is *stateless*. (If it dies, begin new coordinator and restart.)



## Checkpoint-Restart on the *Desktop*

---

- A Finer Grained Form of Virtualization (process-level, instead of operating system level)
  - Typical checkpoint or restart time: 2 seconds
  - Space compression allow for small checkpoint images
  - No worries about memory overcommitment, as with a guest O/S
- POTENTIAL USE CASES (not all currently implemented):
  - *Long-Running Computations*
  - *CPU-Intensive Precomputation*: An interactive software package (e.g Matlab) can generate the data on a powerful computer, checkpoint itself, and restart on a laptop computer.
  - *Robustness*: Upon catching a fault (e.g. segmentation fault), restart from an earlier checkpoint.
  - *Reversible Debuggers*: At each breakpoint, generate an incremental checkpoint (minimal storage required)
  - (cont.)

## **NEXT: The End of the World (as we know it) is Coming**

---

- **The world is changing, as we near the end of Moore's Law.**
  - Memory chips are no longer twice as dense every 18 months.
  - Large RAM is still available on server-class motherboards.
  - But the commodity market doesn't want to pay that premium.
  - So, those of us doing large search and scientific computations are being left out in the cold. We still need those ever larger memories
    - especially as the trend toward multi-core CPUs places ever more pressure on RAM.
  - Heterogeneous computing (cell processors, NVIDIA G8, etc.) only make it worse: The newfound power for regular computations (e.g. SIMD) are causing us to run out of RAM faster than ever before!!!



## What To Do When You Run out of RAM?

---

1. Shared-memory many-CPU computers usually have more RAM (expensive, limited to 100 GB or 1 TB of RAM)
2. Use aggregate RAM of a computer cluster (needs parallel programming)
3. Use the disk on a single computer (much slower than RAM)
4. **Use the many parallel local disks (or SAN) of a cluster (ALMOST FREE!!)**

### A Fourth Major Use for Disks:

1. File System
2. Database (relational or other)
3. Virtual Memory
4. **Parallel Disks as extension of RAM** (similar goal to virtual memory: minimally invasive modification of sequential program, but good performance)



## Our Solution

---

- **Disk is the New RAM**
- Bandwidth of Disk:  $\sim 100$  MB/s
- Bandwidth of 50 Disks:  $50 \times 100$  MB/s = 5 GB/s
- Bandwidth of RAM: approximately 5 GB/s
  
- **Conclusion:**
  1. **CLAIM:** A computer cluster of 50 quad-core nodes, each with 500 GB of mostly idle disk space, is a good approximation to a shared memory computer with 200 CPU cores and a *single* subsystem with 25 TB of shared memory.  
*(The arguments also work for a SAN with multiple access nodes, but we consider local disks for simplicity.)*
  2. The disks of a cluster can serve as if they were RAM.
  3. The traditional RAM can then serve as if it were cache.



## What About Disk Latency?

---

- Unfortunately, putting 50 disks on it, doesn't speed up the latency.
- So, re-organize the data structures and low-level algorithms.
- Our group has five years of case histories applying this to computational group theory — but each case requires months of development and debugging.
- We're now developing both higher level abstractions for run-time libraries, and a language extension that will make future development much faster.
- But first, a popular case history: Rubik's Cube.



## History of Rubik's Cube

---

- Invented in late 1970s in Hungary.
- 1982: “God’s Number” (number of moves needed) was known by authors of conjecture to be between 17 and 52.
- 1990: C., Finkelstein, and Sarawagi showed 11 moves suffice for Rubik’s  $2 \times 2 \times 2$  cube (corner cubies only)
- 1995: Reid showed 29 moves suffice (lower bound of 20 already known)
- 2006: Radu showed 27 moves suffice
- 2007 Kunkle and C. showed 26 moves suffice (and computation is still proceeding)
- D. Kunkle and G. Cooperman, “Twenty-Size Moves Suffice for Rubik’s Cube”, *International Symposium on Symbolic and Algebraic Computation (ISSAC-07)*, 2007, ACM Press, pp. 235–242.
- 2008 Rokicki, 22 moves suffice (50 core-years, contributed by John Welborn and Sony Pictures Imageworks) <http://cubezzz.homelinux.org/drupal/?q=node/view/121>



## Applications from Computational Group Theory (2003–2007)

---

Group	Space Size	State Size	Total Storage
Fischer $Fi_{23}$	$1.17 \times 10^{10}$	100 bytes	1 TB
“Baby Monster”	$1.35 \times 10^{10}$	548 bytes	7 TB
Janko $J_4$	$1.31 \times 10^{11}$	64 bytes	8 TB
Rubik (sym. cosets)	$1.4 \times 10^{12}$	6 bytes	8 TB

*(joint with Eric Robinson, Dan Kunkle)*



## LONGER-TERM GOALS

---

- Why do it?
  1. State space search occurs across a huge number of scientific disciplines.
  2. **Because the world is running out of RAM!**
    - A commodity motherboard holds only 4 GB RAM.
    - We now have 4- and 8-core motherboards, but no one will be putting eight times as much RAM on a *commodity* motherboard.



# Applications Benefiting from Disk-Based Parallel Computation

## Symbolic Algebra:

Applications occur wherever there is intermediate swell!

	Application
1.	Gröbner bases
2.	Term Rewriting (Knuth-Bendix)
3.	Term Manipulations (differential operators, polynomial multiplications, ...)
4.	Large Search Applications (widespread)
5.	Numerous examples from computational group theory
6.	Others



## Applications Benefiting from Disk-Based Parallel Computation

	Discipline	Example Application
1.	Verification	Symbolic Computation using BDDs
2.	Verification	Explicit State Verification
3.	Comp. Group Theory	Search and Enumeration in Mathematical Structures
4.	Coding Theory	Search for New Codes
5.	Security	Exhaustive Search for Passwords
6.	Semantic Web	RDF query language; OWL Web Ontology Language
7.	Artificial Intelligence	Planning
8.	Proteomics	Protein folding via a kinetic network model
9.	Operations Research	Branch and Bound
10.	Operations Research	Integer Programming (applic. of Branch-and-Bound)
11.	Economics	Dynamic Programming
12.	Numerical Analysis	ATLAS, PHiPAC, FFTW, and other adaptive software
13.	Engineering	Sensor Data
14.	A.I. Search	Rubik's Cube



## LONGER-TERM GOALS (cont.)

---

- The world is changing, as we near the end of Moore's Law.
  - Memory chips are no longer twice as dense every 18 months.
  - Large RAM is still available on server-class motherboards.
  - But the commodity market doesn't want to pay that premium.
  - So, those of us doing large search and scientific computations are being left out in the cold. We still need those ever larger memories
    - especially as the trend toward multi-core CPUs places ever more pressure on RAM.
- Our solution is to use disk as the new RAM! (See next slide.)



## Central Claim

---

Suppose one had a single computer with 25 terabytes of RAM and 200 CPU cores. Does that satisfy your need for computers with more RAM?

**CLAIM:** A computer cluster of 50 quad-core nodes, each with a 500 GB local disk, is a good approximation of the above computer. (The arguments also work for a SAN with multiple access nodes, but we discuss local disks for simplicity.)

## When is a cluster like a 25 TB shared memory computer?

---

- Assume 500 GB/node of free disk space
- Assume 50 nodes,
- The bandwidth of 50 disks is  $50 \times 100\text{MB}/s = 5\text{GB}/s$ .
- The bandwidth of a *single* RAM subsystem is about  $5\text{GB}/s$ .

**CLAIM:** *You probably have the 25 TB of temporary disk space lying idle on your own recent-model computer cluster. You just didn't know it.*

*(Or were you just not telling other people about the space, so you could use it for yourself?)*

The economics of disks are such that one saves very little by buying less than 500 GB disk per node. It's common to buy the 500 GB disk, and reserve the extra space for expansion.



## **When is a cluster NOT like a 25 TB shared memory computer?**

---

1. We require a *parallel* program. (We must access the local disks of many cluster nodes in parallel.)
2. The latency problem of disk.
3. Can the network keep up with the disk?



# When is a cluster NOT like a 25 TB shared memory computer?

---

... and why doesn't it matter for our purposes?

- ANSWER 1: We've used this architecture, and it works for us.
- We've developed solutions for a series of algorithmically simple computational kernels from computational algebra — especially mathematical group theory. All of the following computations completed in less than one cluster-week on a cluster of 60 nodes or less.
  - Construction of Thompson Sporadic Simple Group (2003)  
2 gigabytes (temporary space),  $1.4 \times 10^8$  states, 4 bytes per state
  - Construction of Baby Monster Sporadic Simple Group (2006)  
6 terabytes (temporary space),  $1.4 \times 10^{10}$  states, 12 bytes per state
  - Condensation of  $Fi_{23}$  Sporadic Simple Group (2007)  
400 GB (temporary space)  $1.2 \times 10^{10}$  states, 30 bytes per state  
(larger condensation for  $J_4$  now in progress)
  - Rubik's Cube: 26 Moves Suffice to Solve Rubik's Cube (2007)  
7 terabytes (temporary space),  $10^{12}$  states, 6 bytes per state
  - *In progress*: coset enumeration (pointer-chasing: similar to algorithm for converting NFA to DFA (finite automata)).



## When is a cluster NOT like a 10 TB shared memory computer?

1. We require a *parallel* program.
2. The latency problem of disk.
3. Can the network keep up with the disk?



# When is a cluster NOT like a 10 TB shared memory computer?

... and why doesn't it matter for our purposes?

1. We require a *parallel* program. (We must access the local disks of many nodes in parallel.)
  - Our bet (still to be proved): Any sequential algorithm that already creates gigabytes of RAM-based data should have a way to create that data in parallel.
2. The latency problem of disk. **Solutions exist:**
  - (a) For duplicates on frontier in state space search: *Delayed Duplicate Detection* implies waiting until many nodes of the next frontier (and duplicates from previous iterations) have been discovered. Then remove duplicates.
  - (b) For hash tables, wait until there are millions of hash queries. Then sort on the hash index, and scan the disk to resolve queries.
  - (c) For pointer-chasing, wait until millions of pointers are available for chasing. Then sort and scan the disk to dereference pointers.
  - (d) For tracing strings, with each string being a lookup, wait until millions of strings are available. Then ....
3. Can the network keep up with the disk?



# When is a cluster NOT like a 10 TB shared memory computer?

... and why doesn't it matter for our purposes?

1. We require a *parallel* program. (We must access the local disks of many nodes in parallel.)
2. The latency problem of disk.
3. Can the network keep up with the disk?  
(In our experience to date, the network does keep up. Here are some reasons why it seems to just work.)
  - The point-to-point bandwidth of Gigabit Ethernet is about 100 MB/s. The bandwidth of disk is about 100 MB/s. *As long as* the aggregate bandwidth of network can keep up, everything is fine.
  - Researchers already face the issue of aggregate network bandwidth in RAM-based programs. The disk is slower than RAM. So, probably traditional parallel programs can cope.

## Space-Time Tradeoffs using Additional Disk

- Use even more disk space in order to speed up the algorithm.

Methods	Tradeoffs	Sample Storage Reqs.
Implicit Open List Landmarks	Space ↑ Time / Restrictiveness ↓	RAM
Frontier Search Structured Sorting DDD		Disk
Hashing DDD Tiered		Infeasible

“A Comparative Analysis of Parallel Disk-Based Methods for Enumerating Implicit Graphs”, Eric Robinson, Daniel Kunkle and Gene Cooperman, *Proc. of 2007 International Workshop on Parallel Symbolic and Algebraic Computation (PASCO '07)*, ACM Press, 2007, pp. 78–87



# Example Times for Different Phases of Computation

Baby Monster group:  $4.2 \times 10^{33}$  elements

Goal: Construct 13,571,955,000 “points” of permutation representation

Manager	Disk Time	CPU/RAM Time	Network Time
Read/Write	0.5 days	0 days	—
Computation	0 days	3 days	2 days
Check	0 days	0 days	—
Hash	$\ll$ 1 day	$\ll$ 1 day	—
Formatting/Sorting	$\ll$ 1 day	$\ll$ 1 day	—
Duplicate Elimination	$\ll$ 1 day	$<$ 1 day	—
Rebuilding	$\ll$ 1 day	6 days	—
<i>Approximate Total</i>	<i>2 days</i>	<i>10 days</i>	<i>2 days</i>

**NOTE:** Between CPU time and RAM bandwidth, the computation is primarily limited by RAM bandwidth. (*Disk time not the bottleneck.*)

*Using faster CPUs has almost no benefit!  
(Only faster RAM helps.)*



## LONGER-TERM GOAL: Roomy Mini-Language

---

Well-understood building blocks already exist: external sorting, B-trees, Bloom filters, Delayed Duplicate Detection, Distributed Hash Trees (DHT)

### GOAL:

1. Extend C/C++ with Roomy run-time library:  
Support for extensible arrays, lists (support append, concatenate, merge, extract. ...), hash arrays; Emphasize streaming access (using MapReduce, Reduce, Modify-in-Place, ...)
2. **Minimally invasive:** common data structures in user *sequential* code are replaced by Roomy data structures
3. Add Roomy Modules for Common Tasks:
  - (a) Parallel Breadth-First and (approximate) Depth-First Search
  - (b) Priority Queues
  - (c) Table lookup/modify

First alpha release of Roomy expected by end of summer. (open source)



## Example Roomy Code

---

```
// Function to be mapped over cur level to produce next level
void genNextLev(void* val) {
    /*
     * User defined code to generate nbrs array inserted here.
     */
    for(int i=0; i<numNbrs; i++) {
        RoomyList_add(nextLevList, nbrs[i]);
    }
}

// Init lists for duplicates, current level, and next level
RoomyList* allLevList = RoomyList_make("allLev", eltSize);
RoomyList* curLevList = RoomyList_make("lev0", eltSize);
RoomyList* nextLevList = RoomyList_make("lev1", eltSize);

// Add start element
RoomyList_add(allLevList, startElt);
RoomyList_add(curLevList, startElt);

// Generate levels until no new states are found
while(RoomyList_size(curLevList)) {
    // generate next level from current
    RoomyList_map(curLevList, genNextLev);

    // detect duplicates
    RoomyList_removeDupes(nextLevList);
    RoomyList_removeAll(nextLevList, allLevList);
    RoomyList_addAll(allLevList, nextLevList);

    // rotate levels
    RoomyList_destroy(curLevList);
    curLevList = nextLevList;
    nextLevList = RoomyList_make(levName, eltSize);
}
}
```

# QUESTIONS?

---

