Disk-Based Parallel Computing: A New Paradigm

Gene Cooperman

Director, Institute for Complex Scientific Software
http://www.icss.neu.edu/

Head of High Performance Computing Lab
Daniel Kunkle
Xiaoqin Ma
Michael Rieker
Eric Robinson
Vlad Slavici
Ana Visan

Northeastern University
Boston, MA / USA

I: What do we want and what can we expect from applying parallel techniques to pure mathematical research tools?

1. ParGAP: Parallel GAP, 1995 — DIMACS Workshop
2. ParGCL: Parallel GCL (GNU Common Lisp/parallel Maxima), 1995 — ISSAC-95: STAR/MPI
3. Marshalgen for C/C++; 2003–2004 (Nguyen, Ke, Wu and Cooperman); Like pickling for python, serialization for Java; but now, use **Boost.serialization** for C/C++: `http://www.boost.org/libs/serialization/doc/index.html`
4. DMTCP: Distributed Multi-Threaded Checkpointing, 2007 (alpha version: Ansel, Rieker and Cooperman); checkpoint-restart = saveWorkspace/loadWorkspace

---

1. SCIEnce Project: Symbolic Computation Infrastructure in Europe, 2006–2011 (consortium) `http://symbolic-computing.org`

Experience at Interactive, Parallel Computational Algebra (Others)

I: What do we want and what can we expect from applying parallel techniques to pure mathematical research tools?

1. Symbolic Computing over Grid: SCIEnce, 2006- 2011 (U. St. Andrews, RISC-Linz, IeAT-Timisoara, Eindhoven, Tech. Uni. Berlin, Uni-Paderborn, Ecole Polytechnique, Heriot-Watt, MapleSof)
   `http://symbolic-computing.org`
   5-year 3.2M euro Framework VI Project (RII3-CT-2005-026133) Goal: produce a portable framework (SymGrid-Services) that will ...
   Maple, GAP, muPad, KANT

2. Meat-Axe

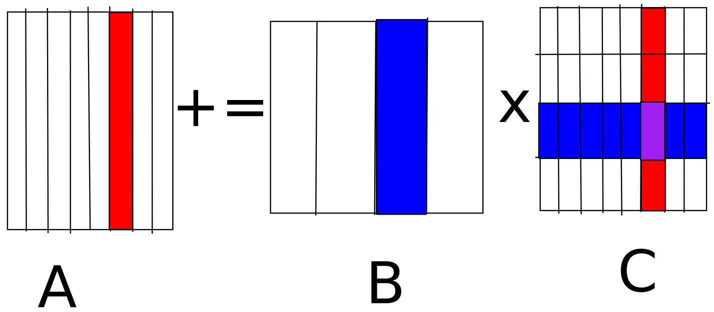
Meataxe: Origins

Efficient Computation with Dense matrices over finite fields:

- First versions of the meataxe (1970's): based around compact representations of vectors over small finite fields (multiple field entries per byte when appropriate) and efficient vector addition and scalar-vector multiply algorithms.
- Next innovation (1980s and early 1990s): *grease* — precompute all (or sometimes just some) linear combinations of a block of rows. In A += B*C, grease blocks of C.
- Around 2000, Jon Thackray started reorganizing the greased multiply working with blocks of rows of B to improve locality of memory access when working from disk, and to improve cache hit ratios.

Meataxe: New Development in C/Assembly Libraries

Steve Linton, Beth Holmes and Richard Parker
{sal,bh}@mcs.st-and.ac.uk,rparker@amadeuscapital.com

Greasing large matrices; key is multiply-add: 

Subdivide A and B vertically and C in both directions. Fill L2 cache with pre-computed linear combinations of rows from the purple block of C. Work sequentially through red and blue strips modifying red strip. Repeat for all pair of strips of A and B.

- Highly optimized representations for matrices and low-level vector arithmetic (field-specific).
- Gaussian elimination can be efficiently reduced to multiply-adds.
- Random 25000x25000 dense matrices over GF(2) multiply in 50 s on Pentium 4/2.4 GHz (about 7 times faster than previously).

Software Demonstrations

1. ParGAP: Parallel GAP, 1995
   `http://www.ccs.neu.edu/home/gene/pargap.html`
   `http://www.gap-system.org/Packages/pargap.html`
2. ParGCL: Parallel GCL (GNU Common Lisp, parallel Maxima), 1995
   `http://www.ccs.neu.edu/home/gene/pargcl.html`
   Compatible with older GCLs and with upcoming GCL-2.7:
   `http://www.gnu.org/software/gcl/`
3. DMTCP: Distributed Multi-Threaded Checkpointing, 2007 (alpha version: Ansel, Rieker and Cooperman);
   checkpoint-restart = saveWorkspace/loadWorkspace
   GPL; write to request a beta test copy when available
4. TOP-C/C++: Task Oriented Parallel C/C++, 1996 Easy task farming in C/C++; `http://www.ccs.neu.edu/home/gene/topc.html`

ParGAP

```
SendMsg( "Print(3+4)" ); # send to slave 1 by default
SendMsg( "3+4", 2); # send to slave 2
RecvMsg( 2 );
SendRecvMsg( "3+4", 2);
squares := ParList( [1..100], x->x^2 );
SendRecvMsg( "Exec(\"pwd\")" ); # Your pwd will differ :-)
SendRecvMsg( "x:=0; for i in [1..10] do x:=x+i; od; x");
SendRecvMsg( "fro i in [1..10]; x:=x+1; od"); #syntax error tolerated
SendRecvMsg( "a:=45", 1 );
SendRecvMsg( "a", 2 ); # "a" undefined, error-tolerant
myfnc := function() return 42; end;;
BroadcastMsg( PrintToString( "myfnc := ", myfnc ) );
SendRecvMsg( "myfnc()", 1 );
FlushAllMsgs();
SendMsg( "while true do od;"); # start infinite loop
ParReset();
```

ParGCL

Similar capability for GCL: GNU Common Lisp;
NOTE: Maxima based on GCL

```
(send-message '(print (+ 3 4)))
(send-message "(+ 3 4)" 2)
(receive-message 2)
(flush-all-messages)
(par-reset)
(send-receive-message '(progn (setq a 45) (+ 3 4)) 1)
```

DMTCP: Distirbuted Multi-Threaded Checkpointing

Alpha version of DMTCP:

```
# Assume on startHost and initially using startPort
./dmtcp_master  # start DMTCP checkpoint controller
# Separate window:
./dmtcp_checkpoint sh pargap.sh
# Request checkpoint of dmtcp_master (or request periodic ckpt)
# After a checkpoint, can quit, or allow software to crash
```

```
./dmtcp_master  # start new DMTCP controller
./dmtcp_restart ckpt_gap_17436930_2326_1170308795.mtcp \
                ckpt_gap_17436930_2333_1170308795.mtcp \
                ckpt_gap_17436930_2334_1170308795.mtcp
ssh remoteHost
env DMTCP_HOST=startHost DMTCP_PORT=startPort ./dmtcp_restart \
                ckpt_gap_17437250_1732_1170308775.mtcp
# Continue calling dmtcp_restart
# Computation resumes after last process restarted
```

TOP-C: Task Oriented Parallel C/C++

Simple task farming in C/C++, plus extensions for *non-trivial* parallelism

TOP-C from the Command Line

```
./topcc --mpi myapp.c
[ OR:  ./topcc --pthread myapp.c
  OR:  ./topcc --seq myapp.c ]

./a.out --TOPC-help

./a.out --TOPC-trace --TOPC-stats --TOPC-num-slaves=50
    --TOPC-aggregated-tasks=5  <APPLICATION_PARAMS>
```

G. Cooperman, "TOP-C: A Task-Oriented Parallel C Interface", 5^{th} *International Symposium on High Performance Distributed Computing* (HPDC-5), 1996, IEEE Press, pp. 141–150

Running TOP-C

```
./topcc -c -g -O2 /tmp/topc-2.5.0/examples/parfactor.c
./topcc -g -O2 parfactor.o
./a.out 123456789

FACTORING 123456789
master -> 1: 2
master -> 2: 1002
master -> 3: 2002
master -> 4: 3002
master -> 5: 4002
1 -> master: TRUE
  UPDATE: TRUE
master -> 1: 5002
...
2 -> master: FALSE
3 -> master: FALSE
3 3 3607 3803
```

Getting Help with TOP-C

```
gene@auditor:/tmp/topc-2.5.0/bin$ ./a.out --TOPC-help

TOP-C Version 2.5.0 (September, 2004);  (distributed (mpi) memory model)
Usage:  ./a.out [ [TOPC_OPTION | APPLICATION_OPTION] ...]
--TOPC-stats[=<0/1>]       display stats before and after
                                                        [default: false]
--TOPC-verbose[=<0/1>]     set verbose mode             [default: false]
--TOPC-num-slaves=<int>    number of slaves (sys-defined default)
--TOPC-aggregated-tasks=<int>  number of tasks to aggregate
                                                        [default: 1]
--TOPC-slave-wait=<int>    secs before slave starts (use w/ gdb attach)
--TOPC-slave-timeout=<int> dist mem: secs to die if no msgs, 0=never
                                                        [default: 1800]
--TOPC-trace=<int>         trace (0: notrace, 1: trace, 2: user trace fncs.)
--TOPC-procgroup=<string>  procgroup file (--mpi)       [default: "./procgroup"]
--TOPC-safety=<int>        [0..20]: higher turns off optimizations,

The environment variable TOPC_OPTS and the init file ~/.topcrc
  are also examined for options (format:  --TOPC-xxx ...).
You can change the defaults in the application source code.
```

First-Ever Computations Using TOP-C Model/Tools

	Baby Monster perm. rep. (deg. $\approx 1.3 \times 10^{10}$) (over $GL(4370,2)$) J_4 perm. rep. (deg. 173,067,389) (over $GL(1333,11)$)	Th condensation (from perm deg. 976,841,775 to matrix dim. 1,403) (over $GL(248,2)$)
Ly coset enum. (8,835,156 cosets)	Parallelization of GNU Common Lisp (GCL) Parallelization of GAP (Groups, Algorithms and Programming) Parallelization of Geant4	J_4 condensation (from perm deg. 173,067,389 to matrix dim. 5,693) (over $GL(112,2)$) Ly perm. rep. (deg. 9,606,125) (over $GL(111,5)$)
TOP-C (shared mem.)	**TOP-C (dist. mem.)**	
POSIX threads	**MPI (Message Passing Interface)**	

Paradox: Interactive, Parallel Computation

- Paradox 1:
  1. Parallel Computing is good for accelerating long-running jobs.
  2. Interactive Computing is good for computationally steering a sequence of short jobs.
- Paradox 2:
  1. Large parallel jobs require reservation of large resources by placing job in a batch queue.
  2. Interactive jobs require immediate access to resources.
- Paradox 3:
  - Long-running jobs in computer algebra often generate large intermediate swell; computations overflow from RAM to disk

Different cases

1. Large resources (1000+) CPUs is not currently an interactive job
2. Moderate resources on a medium-size cluster can be used interactively, but one wants to save the ”parallel workspace”, while thinking about the problem, and then return later. REQUIREMENT: checkpointing
3. Multi-core CPUs on a desktop — one ideally wants thread parallelism, to save on use of RAM and cache; This will become especially important with 4-core and 8-core CPUs.

William Stein's Question:

Parallel Implementations of Common Algorithms are Not in Standard Use Today. WHY?

1. **2-core desktop/laptop:**
   Advantage: twice the speed;
   Disadvantage: must learn a parallel programming tool;
   Interactive computation: twice the speed = 1 second per step instead of 2 second per step — not enough reward to overcome the learning barrier.

2. **medium-size cluster (32 CPUs?):**

   (a) **Task farming:** low barrier to entry *if someone else sets up the software.*

   (b) **Parallel programming:** high barrier to entry, but potential high rewards — requires a new generation more accustomed to the tools??

   (c) **Checkpointing = SaveWorkspace:** highly desirable for long interactive sessions

3. **large cluster (1000+ CPUs):** *interactive computing?????*

PART II: Disk-Based Parallel Computing

Disk as the New RAM

Bandwidth of RAM: 3.2 GB/s (PC-3200 RAM, single channel)
Bandwidth of Disk: ~ 50 MB/s
Bandwidth of Cluster of 64 nodes: 3.2 GB/s

Issues: Bandwidth of Network, ability of CPU to keep up

Disk: the New RAM (example)

Initial Testbed: large search and enumeration

- Key data structure: sorted array
- Key algorithm: sorting $\Rightarrow$ *merge*, *union*, *intersection*
  (sorting on disk done as *external sort*: 4 passes in practice; fewer passes when there are opportunities to pipeline it with previous phase of computation)

Problem: Insertion of new elements

Solution: Defer insertions; sort elements to insert; and merge them into sorted array in large batch

Duplicate Elimination in Baby Monster

Optimization: eliminate duplicate insertions before merge; Use a new hash array in RAM to accumulate elements to insert. Need only store one bit per hash element: 1 = present; 0 = not present

Example: AI search: enumeration of states via open queue, as in breadth-first search

1. If element to insert hashes to 0, it is new; add to *open queue* on disk
2. If element to insert hashes to 1, it is either a hash collision or a duplicate: add to *collision queue* on disk
3. Continue to read from open queue and hash its neighbors: neighbors will also be stored either in open queue or collision queue
4. sort collision queue and eliminate duplicates
5. sort open queue
6. merge collision queue, open queue and original sorted array on disk
7. elements of collision queue that are determined to be new become the next open queue, and we repeat step 1.

Duplicate Elimination in Baby Monster (case 2)

- Hash array too large for RAM; must be stored on disk
  1. All new elements to insert are saved on disk in *open queue*
  2. As neighbors of elements in open queue are expanded, portions of the open queue are transferred into a closed set
  3. The closed set is then externally sorted according to hash index
  4. The closed set is then merged into the existing hash array

NOTE: Both RAM-based and disk-based hash arrays adapt easily to distributed computing. Each node is responsible for a contiguous sequence of hash indexes.

Times for Different Phases of Baby Monster Computation

(joint work, Eric Robinson and C.)

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	—
Approximate Total	*2 days*	*10 days*	*2 days*

NOTE: Uses approximately 7 terabytes of disk space

NOTE: Between CPU time and RAM bandwidth, the computation is primarily limited by RAM bandwidth.

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

Application: Search and Enumeration Problems

Branch-and-Bound, A* search

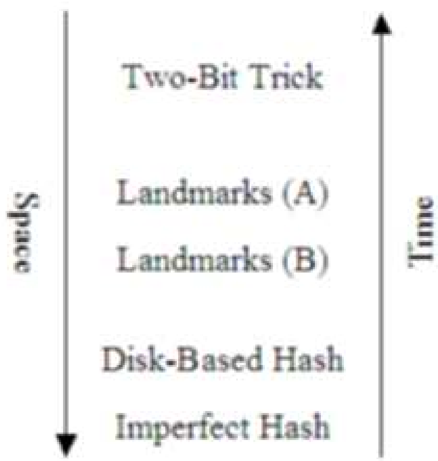
Given a state, and a generator/operation, produce a new state
This gives rise to a natural graph in which nodes correspond to states, and edges are labelled by generators or operations. A search/enumeration proceeds by breadth-first search, developing a spanning tree.
Potential applications (some of it is future work):

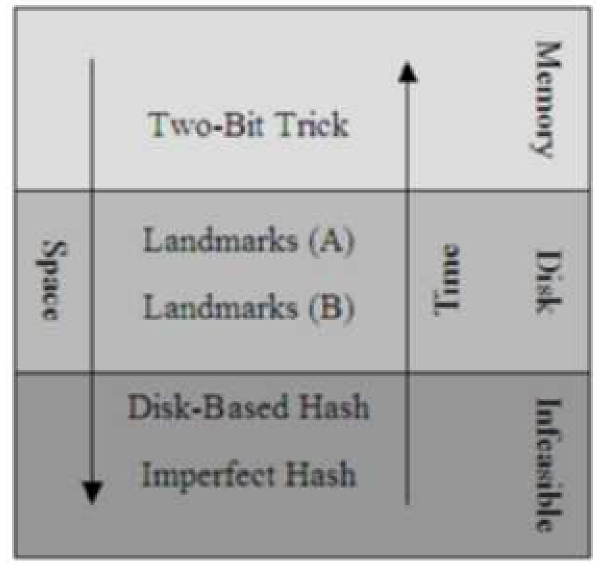
- Enumeration of Orbit Elements
- Orderly Generation of Brendan McKay (symmetry and search)
- Gröbner bases, Knuth-Bendix, similar "completion algorithms"
- SAT (satisfiability) *Example use: VLSI circuit verification*
- Integer Programming
  Example use: Travelling Salesman Problem, Airline schedules

Outline

Disk-Based Computation

General Philosophy in case of Search:





Two-Bit Trick

- Assumes dense, perfect hash function w/ inverses (no hash collisions)
- Breadth-first search, storing level of node *modulo 3* of spanning tree in hash table (2 bits/node)
- Given a node, can now find minimal length path to origin:
  1. Look up level of current state in hash table
  2. Given state, use operators to find all neighbors of node
  3. Look up levels of all neighboring states in hash table
  4. Choose a state whose level is one less than the current level, modulo 3
  5. Repeat on the newly chosen state

Showed Rubik's $2 \times 2 \times 2$ cube (corners, only) always solvable in 11 moves. Used 1 MB on a SUN-3 workstation having only 4 MB of RAM. G. Cooperman, L. Finkelstein, and N. Sarawagi, Applications of Cayley Graphs, Algebra, Algebraic Algorithms and Error-Correcting Codes (AAECC-8), Springer-Verlag Lecture Notes in Computer Science **508**, pp. 367–378, 1990. (Also in G. Cooperman and L. Finkelstein. "New methods for using Cayley graphs in interconnection networks", *Discrete Applied Mathematics*, **37/38**, pp. 95–118, 1992.)

Outline

Disk-Based Computation

1. Data Structure: Distributed Database of Key-Value Pairs
2. Building Blocks: Algorithmic Subroutines
3. Integration into General Search Routines
4. Example Large Computations: Baby Monster; Rubik's Cube
5. Other Applications
6. Natural API (in progress)

Outline

Disk-Based Computation

1. Data Structure: Distributed Database of Key-Value Pairs
   (a) Goals
      i. Key-Values: *Set(key, value); Get(key); Delete(key)*
      ii. Duplicate Elimination
   (b) Data Structures for Database
      i. Distributed Hash Array
      ii. Distributed Sorted Array
      iii. Double Hashed Array: (hybrid of above two data structures)
2. Building Blocks: Algorithmic Subroutines
3. . . .

Disk-Based Computation

1. Data Structure: Distributed Database of Key-Value Pairs
   (a) Goals
   (b) Data Structures for Database
2. **Building Blocks: Algorithmic Subroutines**
   distributed hashing, sorting, duplicate elimination, binary search, batching of queries, pipelining of computations, striped access to distributed data structures, on-the-fly compression and expansion of data structures, Bloom filters, two-phase commit in support of persistent data, structures, ...
3. ...

Disk-Based Computation

1. Data Structure: Distributed Database of Key-Value Pairs
   (a) Goals
   (b) Data Structures for Database
2. Building Blocks: Algorithmic Subroutines
   (a) **EXAMPLE: Bloom filters:** Use hash array with only one bit per hash entry; We wish only to record if key is present or not present in hash table; Use k hash functions, and for a given key, set k bits of hash table (one bit for each hash function); To test presence of key, test all k bits; This greatly reduces hash collisions.
3. . . .

Disk-Based Computation

1. Data Structure: Distributed Database of Key-Value Pairs

   ii. **Data Structures for Database**

      i. Distributed Hash Array: Good for key-value database
         Batching of queries important for efficiency

      ii. Distributed Sorted Array: Good for duplicate elimination
         Given source of new key-values, externally sort it, and compare with original sorted array; Merge on the fly

      iii. Doubly Hashed Array: Good for duplicate elimination
         Key-value pairs stored in buckets, based on high bits of hash index;
         High bits also determines node to hold bucket;
         Key-value pair stored unsorted in bucket; For duplicate elimination, sort elements of bucket in RAM

2. Building Blocks: Algorithmic Subroutines

3. . . .

Disk-Based Computation

4. Example Large Computations:

   (a) **Construction of Permutation Representation of Baby Monster**
   GOAL: enumerate al 13,571,955,000 “points”
   Each point given asvector of dimenstion 4,370 over $GF(2)$ (547 bytes per “point”)
   STORAGE: about 7 terabytes ($13,571,955,000 \times 547$ bytes)
   TIME: About 750 hours BOTTLENECK: RAM: limited by speed of reading vectors/matrices from RAM for matrix-vector multiplication

   (b) Rubik’s Cube

Disk-Based Computation

4. Example Large Computations:

   (a) Construction of Permutation Representation of Baby Monster

   (b) Rubik's Cube
   ˜4.3×10^{19} states
   Square subgroup of about 6.6×10^{5} elements
   SUBGOAL: enumerate all 6.5×10^{13} cosets $(4.3 \times 10^{19}/(6.6 \times 10^{5}))$
   Reduction: Only enumerate cosets up to symmetries of cube
   About 1.5×10^{12} symmetrized cosets
   STORAGE: 1 byte per symmetrized coset (1.5 terabytes) times a factor of at least two for frontier expansion in search

Outline

Disk-Based Computation

5. **Other Applications:**
   (a) Integer Programming
   Example use: Travelling Salesman Problem, Airline schedules
   (b) SAT (satisfiability) *Example use: VLSI circuit verification*
   (c) Applications to distributed linear algebra

1. Natural API (in progress)

Why are CPU vendors selling multi-core instead of faster CPUs?

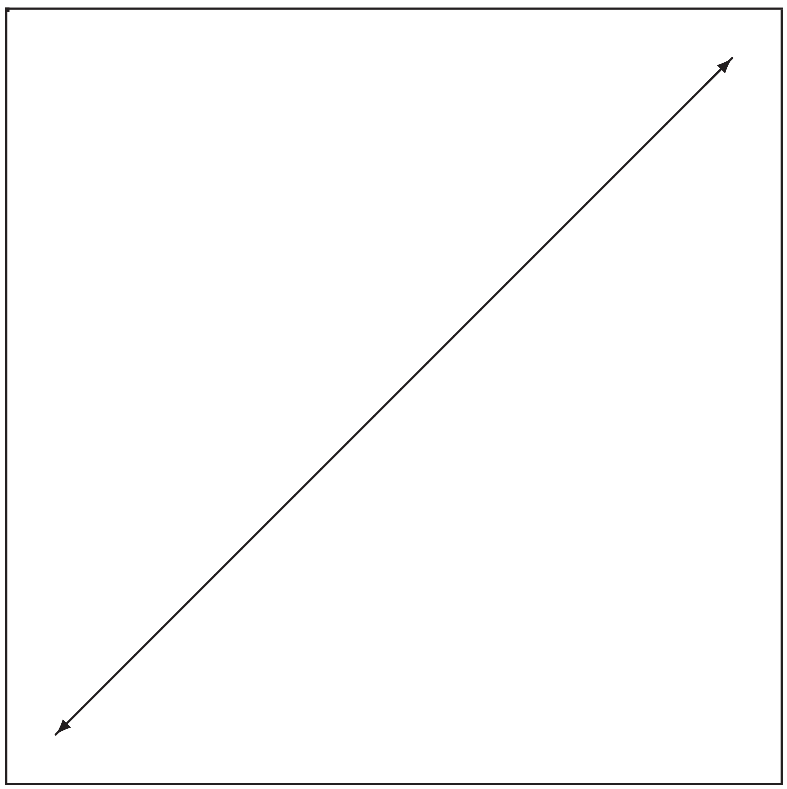
- Speed of electrical signal: $\approx 10^9$ cm/s
- 1 GHz clock rate
- Distance travelled by electrical signal in one clock cycle: $\approx$ 1 cm
- Chip linear dimension: $\approx$ 1 cm

Moore's Law (every 18 months)

- Twice as many gates/mm^2
- Twice the clock speed
- Half the distance travelled by electrical signal per clock cycle

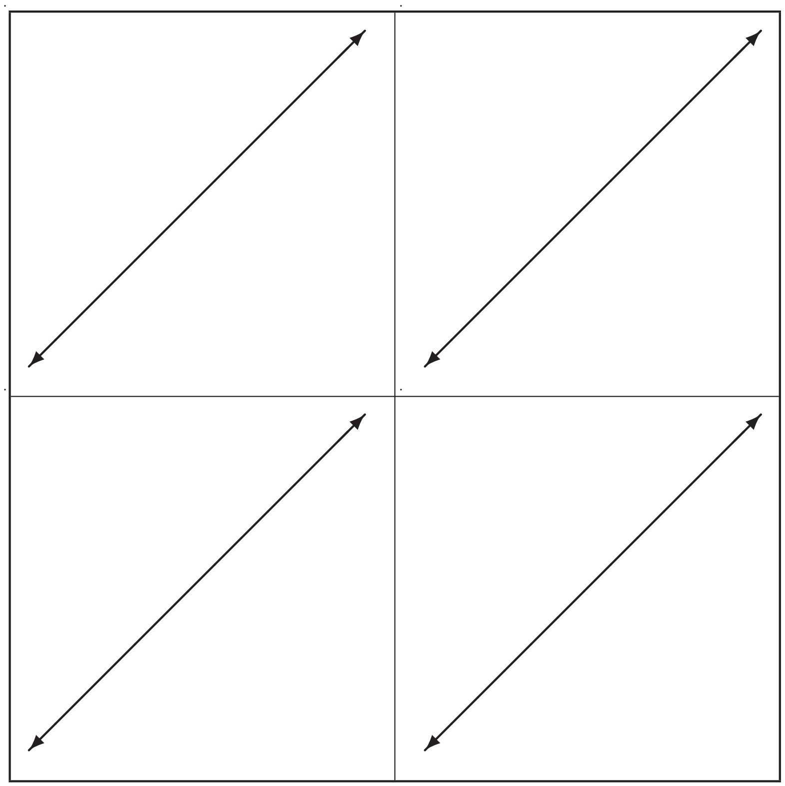
2000: New millenium

Define a chip unit as a chip rectangle such that an electrical signal can cross the diagonal in one clock cycle.



$\longleftrightarrow$: distance travelled by eletrical signal in one clock cycle

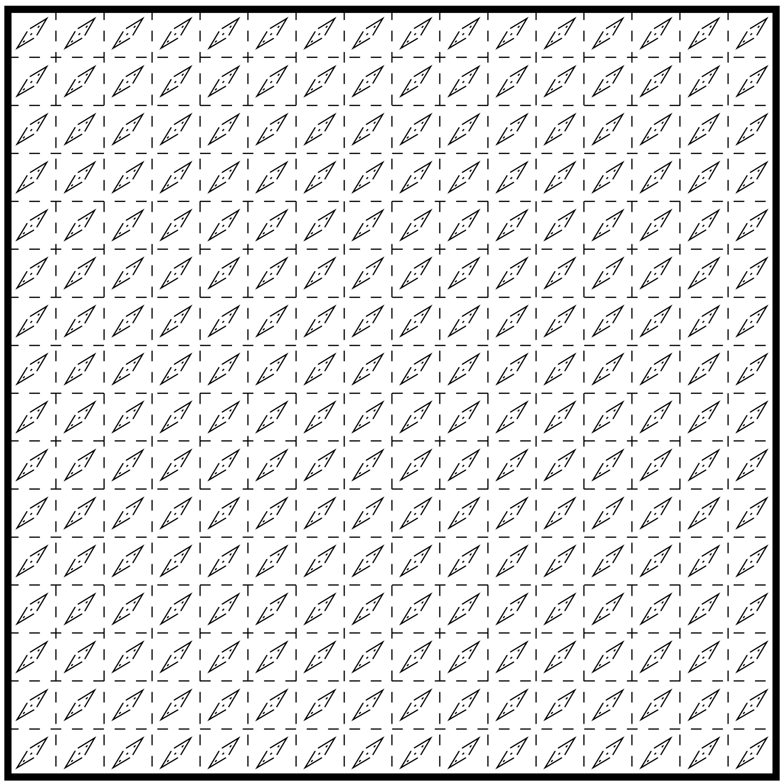
mid-2001: 1.5 years later



1) 4 times as many units

2) Twice as many total gates $\Rightarrow$ 1/2 as many gates/unit

2009: 9 years later



1) 4,096 times as many units

2) 64 times as many total gates ⇒ 1/64 as many gates/unit

Common sense check

- 1985 – 1995: pipelining
- 1990 – 2000: superscalar CPU's, ILP (Instruction Level Parallelism)
- 2000: Intel/H-P Itanium, EPIC (Explicitly Parallel Instruction Computer)
- 2002: Simultaneous MultiThreading: Intel Xeon Pentium-4 (Hyper-Threading)
- 2002: Dual Core Chips: IBM Power4
- 2005: Mainstream Dual Core: AMD/Intel, IBM Power5 (dual core + simultaneous multithreading = 4 processors)
- 2006: Cell Architecture (Playstation 3): Sony/Toshiba/IBM (1 core + 8 vector processors — on-chip pipelined parallelism support)
- 2010: IRAM (?), CRAM (?)

Memory Wall

CPU/RAM	New, Two-Pass Algorithm	Traditional Algorithm
2.66 GHz Pentium 4 DDR-266 RAM	0.042 s	0.159 s
0.6 GHz Pentium III PC-100 RAM	0.131 s	0.097 s

Two-Pass Permutation Multiplication versus Traditional Algorithm (joint work: X. Ma, V.H. Nguyen and C.)

```
Object Z[N], Y[N]; // Object is ''int'' in above experiments
int X[N];
for (i=0; i<N; i++)
     Z[i]=Y[X[i]];
```

Why is Two-Pass Permutation Now Faster?

Two Large Reasons:

1. The Pentium 4 has a longer cache line.

   The Pentium 4 has a 128 byte cache line: four times longer than the 32 byte cache line of the Pentium III.

2. The bandwidth of DDR-266 (PC-2100) RAM is higher, but the latency is not faster.

   - *DDR-266/PC-2100 has a bandwidth of 2.2 GHz, as compared to 1.1 GHz for older PC-100 RAM.*
   - *The latency of DDR-266 RAM and PC-100 RAM are both about 25 ns.*

Future Trends:

1. Higher bandwidth memory

   - **Evidence:** Today, we see dual-channel memory offering effective 800 MHz system busses (seven times faster than DDR-266)
   - **Evidence:** Some scientific applications, such as matrix multiplication, FFT, etc., are now being programmed to use the high-bandwidth memory (and greater parallelism) of video boards. *(Some applications are even using dual video boards to double this speed.)*
   - **Side Effect:** Possibly *even longer CPU cache lines*, in order to keep up with the high bandwidth)

2. Latency mostly unchanged

   - The time to precharge the external buffer of a DRAM chip is increasing slightly, as lower on-chip voltages must be raised to the higher voltage levels of the motherboard. *This is a long-term problem, for as long as DRAM and CPU are on different chips!*

Relative Speeds: CPU, RAM, Disk and Network

CPU bandwidth	2,400 MB/s	(3 GHz $\times$ 8 byte words)
Network bandwidth (point-to-point)	100 MB/s	(1 Gb/s theoretical max for Gigabit Ethernet)
Network bandwidth (aggregate)	1,000 MB/s	(varies by vendor)
RAM bandwidth (DDR-400)	3,300 MB/s	(maximum)
Disk bandwidth (per disk)	50 MB/s	(typical)

Aggregate bandwidth of 50 disks: $50 \times 50 = 2{,}500$ MB/s

Duplicate Elimination

Optimization: eliminate duplicate insertions before merge; Use a new hash array in RAM to accumulate elements to insert. Need only store one bit per hash element: 1 = present; 0 = not present

Example: AI search: enumeration of states via open queue, as in breadth-first search

1. If element to insert hashes to 0, it is new; add to *open queue* on disk
2. If element to insert hashes to 1, it is either a hash collision or a duplicate: add to *collision queue* on disk
3. Continue to read from open queue and hash its neighbors: neighbors will also be stored in open queue or collision queue
4. sort collision queue and eliminate duplicates
5. sort open queue
6. merge collision queue, open queue and original sorted array on disk
7. elements of collision queue that are determined to be new become the next open queue, and we repeat step 1.

Duplicate Elimination (case 2)

- Hash array too large for RAM; must be stored on disk

  1. All new elements to insert are saved on disk in *open queue*
  2. As neighbors of elements in open queue are expanded, portions of the open queue are transferred into a closed set
  3. The closed set is then externally sorted according to hash index
  4. The closed set is then merged into the existing hash array

NOTE: Both RAM-based and disk-based hash arrays adapt easily to distributed computing. Each node is responsible for a contiguous sequence of hash indexes.

Optimization: Bloom Filters

Recall in-RAM hash array with one bit per hash element: 1 = present; 0 = not present; **Idea of Bloom filters:**

1. Make hash array *k* times larger (retain same load factor for hash array)
2. Define *k* distinct hash functions
3. For each new element, apply *all* of the *k* hash functions, and set each corresponding entry of the hash array to 1
4. If any entry of the hash array was formerly 0, then this element is new: add to *open queue*
5. Else, add to *collision queue*

Example: Assume for simplicity that no duplicates are generated.
if the original hash array had a load factor of 1/2, then the new hash array will be *k* times larger, but the the size of the collision queue will be reduced by a factor of $1/2^k$.

Planned Computation for Rubik's Cube

(joint work, Daniel Kunkle and C.)

Rubik's cube has approximately 4.3×10^{19} states

20-year conjecture: all states of Rubik's cube can be solved in at most 20 moves (known as "God's number")

How close can we get?

Standard strategy: partition 4.3×10^{19} states into *cosets* of equal size

Previously (Reid, 1993): Each coset has approximately 10^{10} states;
approximately 5×10^{8} such cosets to check;
all states solved in 29 moves (recently shaved to 27 moves)

Planned: Each coset has approximately 10^{4} states;
after symmetries, *only* 10^{14} cosets to check; (required data structure fills about 6 terabytes of aggregate disk);
small cosets imply that each can be checked fully.

Questions?

QUESTIONS?