Parallel and High Performance Computing

Gene Cooperman

Northeastern University
Boston, MA / USA

Organization of Course on Parallel Computing

1. History
2. Hardware
3. Assembly Language
4. Operating Systems
5. High Level Languages
6. Meta-Computing
7. ... special topics, as time permits ...

History

Those who cannot learn from history are doomed to repeat it.

— George Santayana

History (abridged)

As soon as computers and networks were invented, many people started experimenting with parallel computing. We begin our history at the point when companies began building dedicated parallel computers. The standard wisdom at that time (before the PC), was that anybody could write a parallel program, but one needed to build specialized parallel hardware in order to get good price-performance.

History: Thinking Machines: CM-1 and CM-2

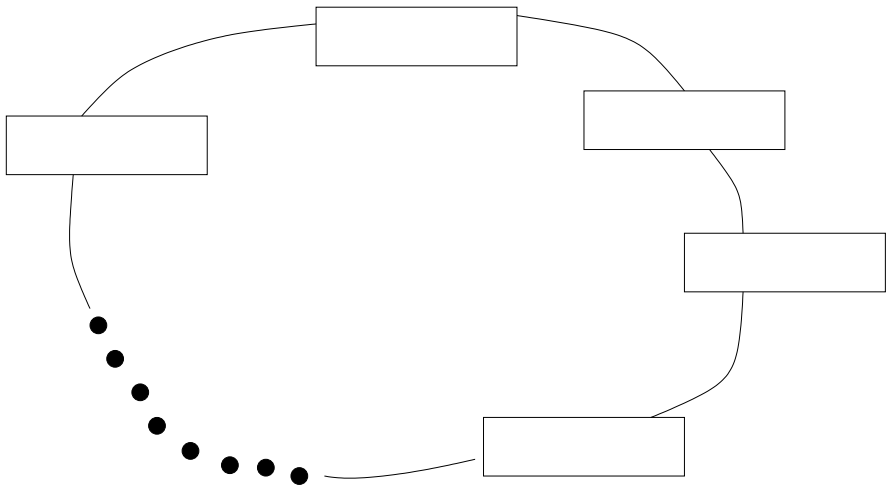
- CM = “Connection Machine”
- goal was massive parallelism: thousands of CPUs (processing nodes)
- Thinking Machines founded in 1983
- Danny Hillis: co-founder of Thinking Machines; based on his research at MIT
- Up to 65,536 processors
- Specialized processors: Many relatively slow processors
- in order to produce so many CPUs at that time, each CPU was a 1-bit CPU (a competitor, MasPar, later produced a 4-bit variation)
- initial languages: StarLisp and StarC
- *SIMD* parallelism
- Hypercube *interconnection network* (few hops between processors)

Terminology: SIMD/MIMD

- "It's all in the name."
- SIMD = "Single Instruction, Multiple Data", as opposed to
  MIMD = "Multiple Instruction, Multiple Data"
  1. Front-End (FE) feeds single instruction stream to all processors
  2. Each processor contains a certain number of 1-bit registers (mechanisms existed to transparently create larger words)
  3. Condition register on each processor determined whether the next few instructions would or would not be executed
- Marketing: Using commodity computers meant multiple instructions and multiple data: hard to program
- (The word SIMD was later reused by Intel in a different context for SIMD assembly instructions.)

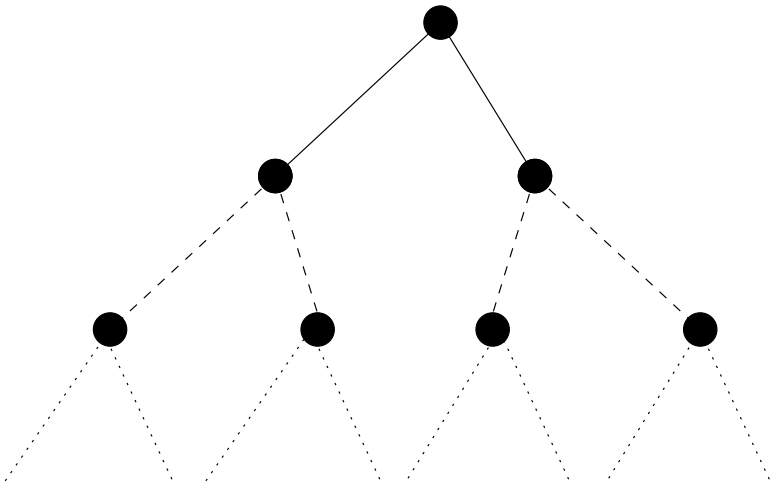
History: KSR (Kendall Square Research)

- Kendall Square Research founded in 1986
- Kendall Square was a location near MIT
- co-founder: Henry Burkhardt III (who had helped found Data General, Encore Computer)
- Fewer extremely fast processors
- specialized ring network: *extremely* fast network to keep up
- expansion: rings of rings



History: CM-5: announced in 1991

- fat-tree network, *NUMA* architecture
- upper paths have higher bandwidth
- highly local computations seldom need to use upper branches
- somewhat local computations seldom need to use the highest branch
- *SPMD* software model (not SIMD or MIMD)



Terminology: SPMD

- Marketing: CM-5 looked a lot like MIMD; but Thinking Machines had been selling SIMD
- Solution: Call it **SPMD**: Single Program Multiple Data
- Write one program with "if" statements to indicate if the particular processor should execute some portion of the program

It was argued that the earliest parallel programs had been MIMD, and that SPMD programs are easier to write, because one maintains a single program. In fact, almost all parallel programs today would be considered SPMD.

Terminology: NUMA

- **NUMA**: Non-Uniform Memory Architecture
- Local memory could be accessed faster
- Distant memory took longer to reach

IBM SP-2: commodity hardware (IBM CPUs)

- developed in early 1990s
- fast, custom network
- fast, commodity CPU (IBM POWER chip)
- commodity software: AIX (IBM dialect of UNIX)

History: Beowulf cluster: commodity computing

- Co-developed in mid-1990s at NASA and NIH by Brooks and Billings; Operating in 1997
- IDEA: It's not how many processors (CM-2), or the speed of the processors (KSR), but the price of the processor. Cheap commodity processors yield better price-performance ratio. Cheap commodity network is good enough. Linux provides commodity software.

TODAY: Commodity Hardware; Looking for Commodity Software

- Hardware: Distributed software; commodity networks
- “Assembly” Level: MPI/sockets, OpenMP/POSIX threads
- High-Level Parallel Languages (still looking for a standard)
- Wide Area Computing: Grid
- Meta-Computing: Legion, etc. (no standard)

Hardware

Hardware: the part of the computer that you can kick.

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

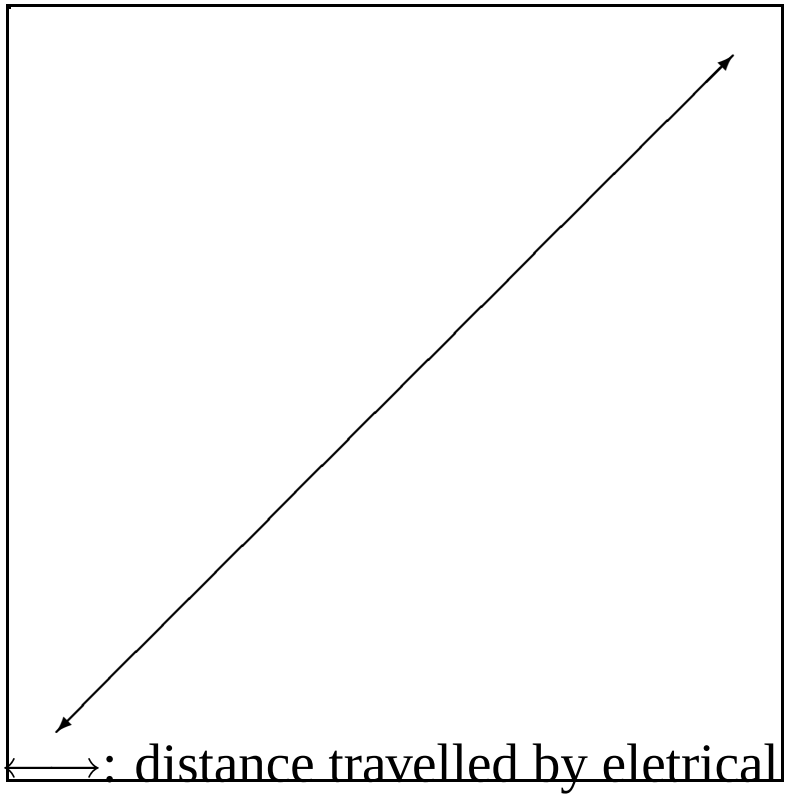
- Speed of electrical signal: $\approx 10^9$ cm/s
- 1 GHz clock rate
- Distance travelled by electrical signal in one clock cycle: ≈ 1 cm
- Chip linear dimension: ≈ 1 cm

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

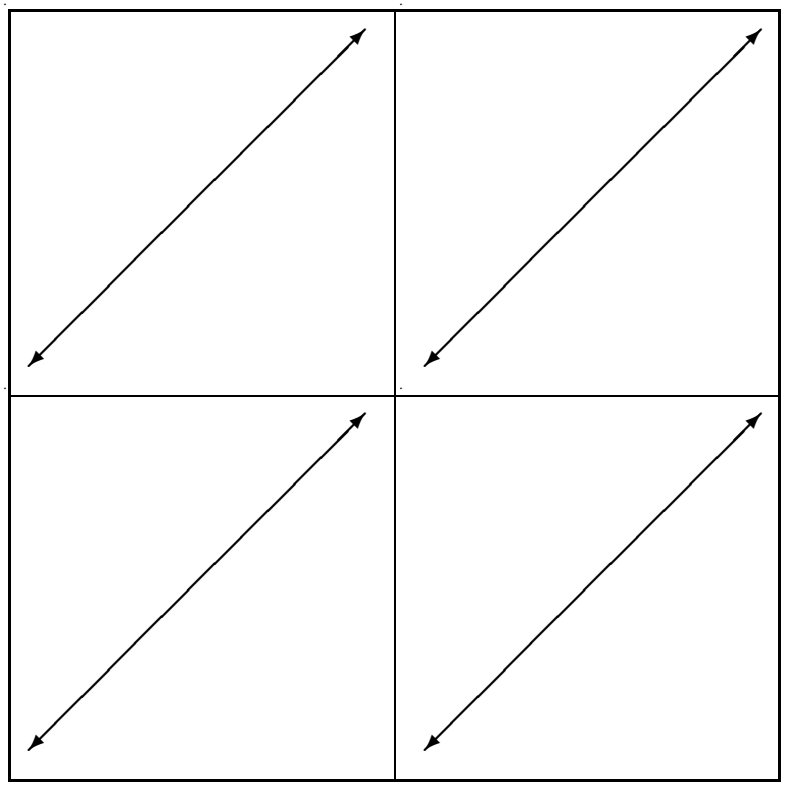
2000: New millenium

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

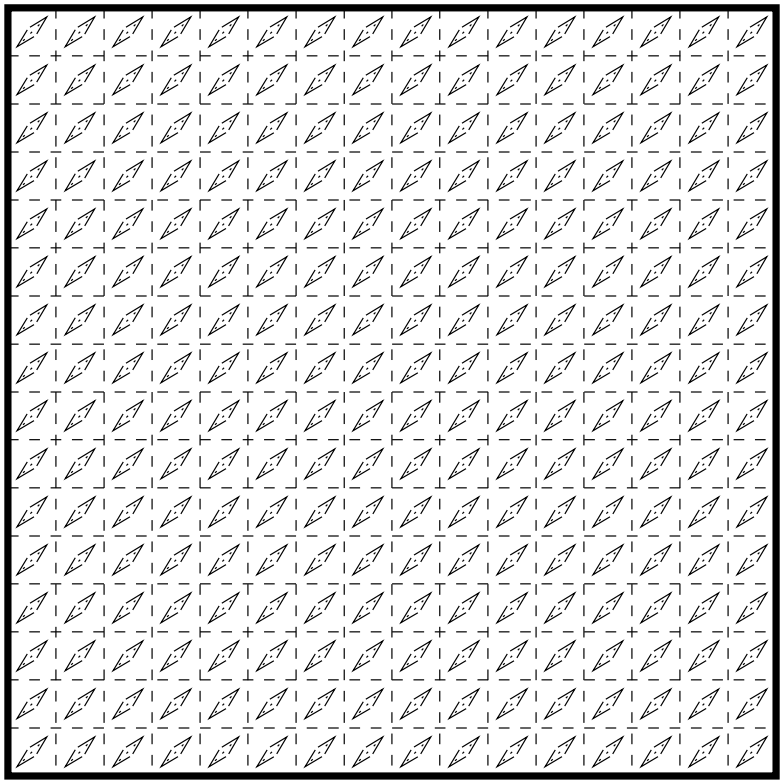
mid-2001: 1.5 years later



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 $\Rightarrow$ 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 × 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

Hardware vs. Software

Software:	Distributed Memory (message passing)	DSM	Shared Memory
Assembly:	Sockets/MPI		POSIX threads (pthreads)
Hardware:	Distributed Computers	hardware-supported lin. addr. space	Shared Memory/SMP

(However, implementations can support message passing model on shared memory hardware (each process has *mailbox* with mutex lock);
or shared memory software on distributed hardware (see DSM).

Shared Memory and Protocols

1. *DSM:* (next slide)
2. Cache Coherency
3. Memory Consistency
4. Relaxed Consistency

Hardware: DSM

- Linear address space
- Generalizes idea of page faults
  - If virtual page is not in RAM, fetch it from disk.
  - If DSM page is not in local RAM, then fetch it from **owner** of page.
- Hardware Support: e.g. SCI: Scalable Coherent Interface
- ISSUE: heavy sharing of pages by many CPUs
- ISSUE: *ping-pong:* (two processes both want to write to variable *x*)
- ISSUE: *false sharing:* (Process A continually writes to variable *x*, and process B continually writes to variable *y*. Unfortunately, variable *x* and variable *y* are located on the same DSM page.)

DSM Protocol

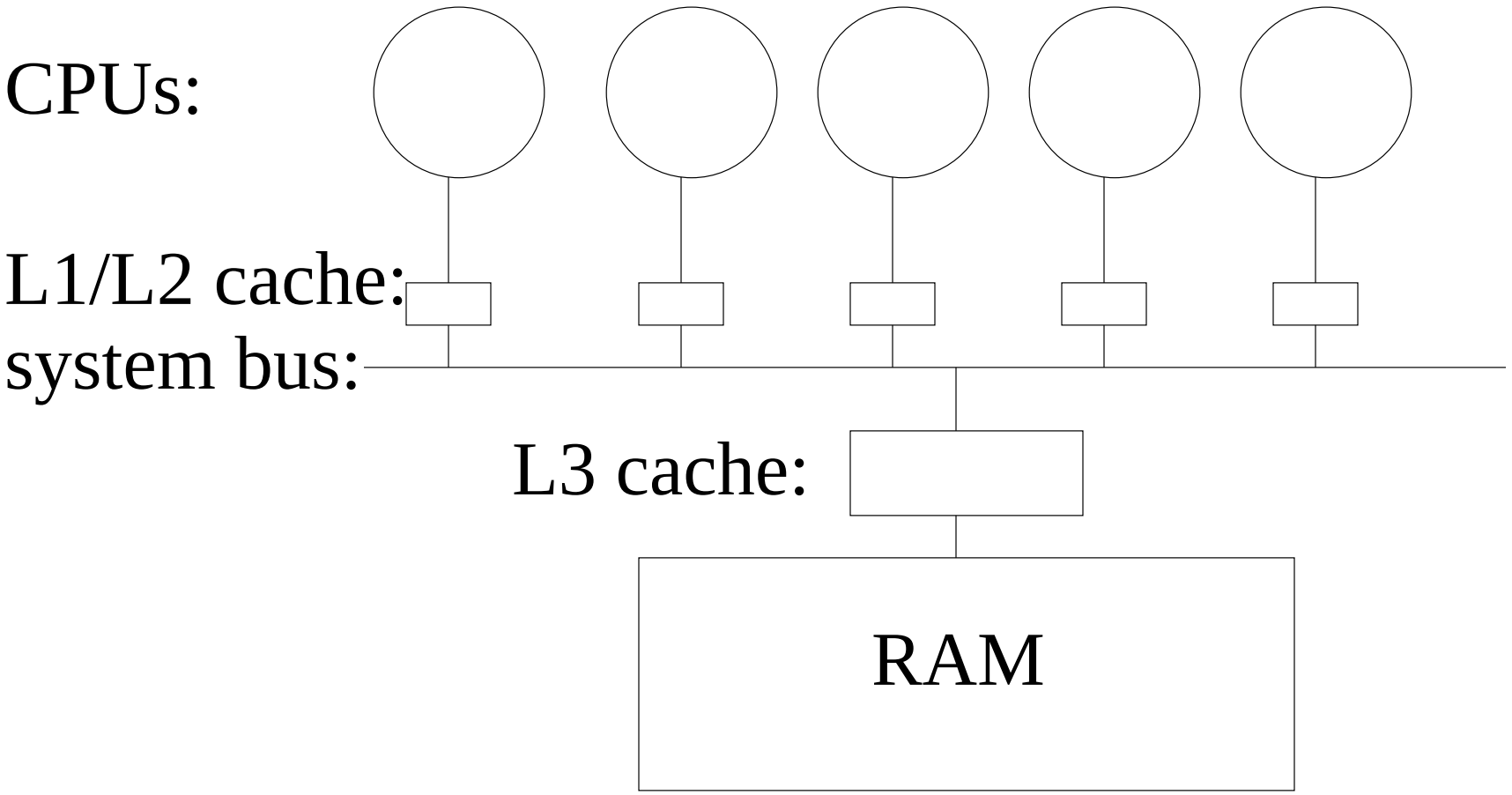
1. Each “virtual page” is owned by some process or node. (For example, the upper bits of the address determine the page number, and the upper bits of the page number determine the owner node.)
2. Example: Node A owns pages 0 → 1,023; Node B owns pages 1,024 → 2,047; etc.
3. Owner can “lend” the page to other processes. Owner can lend out:

   (a) multiple read-only copies, **or**

   (b) a single read-write copy

Hardware: SMP architecture

- **SMP**: **S**ymmetric **M**ulti**P**rocessing
- **Protocol**: *bus snooping*



Shared Memory and Protocols

1. DSM
2. *Cache Coherency*
   (a) **Problem:** CPU A writes to address 0x10a34c in L1 cache; CPU B reads from address 0x10a34c
   (b) **Solution:** *Bus snooping:*
      i. CPU has modified address 0x10a34c, but only in cache.
      ii. CPU A snoops (spies) on a common bus.
      iii. When CPU A detects a read request on bus by CPU B, CPU A raises hardware signal.
      iv. The hardware signal suspends the read request.
      v. CPU A writes 0x10a34c from L1 cache to RAM.
      vi. CPU A lowers hardware signal and CPU B continues its read request.
3. Memory Consistency
4. Relaxed Consistency

Volatile keyword

1. **Problem:** CPU A writes to variable *x*; New value is stored in a register; CPU B reads variable *x*; But the new value of variable *x* is only in a register in CPU A
2. **Solution 1:** Declare variable *x* to be `volatile`.
   (The volatile keyword in C developed for sake of device drivers talking to external peripherals. But it's also useful for multi-threaded programs.)
3. **Solution 2:** C99 requires that the compiler arrange for variables in registers be written out to cache before any function call.

Summary: Cache coherency protocols

1. If a common bus is present, implement *bus snooping* in hardware,
2. Else implement DSM-style cache coherency protocol.

While the use of the `volatile` keyword is not formerly part of a cache coherency protocol, it often serves a similar purpose.

Memory Consistency

1. DSM
2. Cache Coherency
3. *Memory Consistency:*

```
Process A                    Process B
 x = 0; [initially]            y = 0; [initially]
 ...                           ...
 x = 1;                        y = 1;
 if ( y == 0 ) {..}            if ( x == 0 ) {..}
```

4. *Relaxed Consistency Models:* Sequential Consistency; total store order, partial store order; release consistency
   Concept of *memory barrier*

Software

“The box said, Win95 or better required... so I used a Mac !”

[REPLACE Mac BY YOUR FAVORITE OPERATING SYSTEM.]

— Tim Scoff

Software: 8 Parallel Languages as Models

1. MPI/sockets (message abstraction on top of network ”packets” of sockets)
2. OpenMP/POSIX Threads
3. High Performance Fortran *(data parallelism)*
4. Unified Parallel C (UPC) *(linear address space)*
5. Cilk *(work-stealing, shared memory)*
6. Linda / Jini/javaspaces *(tuple space)*
7. TOP-C (Task Oriented Parallel C/C++) *(task-oriented parallelism / master-worker)*
8. Legion *(meta-computing)*

Cilk: Work-Stealing

Two Primitive Operations:

- spawn, sync

Example: Fibonacci numbers: $f_n = f_{n-1} + f_{n-2}$

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y)
  }
}
```

Linda: Tuple space

Three (plus two) Primitive Operations:

1. in (blocking read and delete), out (insert), rd (blocking read)
2. non-blocking versions: inp, rdp

Example: `http://heather.cs.ucdavis.edu/~matloff/Linda/NotesLinda.NM.h`

```
if (NodeNumber == 0) out("sum of all Ys",0);
in("sum of all Ys", ? Tmp);
Tmp += Y;
out("sum of all Ys",Tmp);
/* barrier code would go here */
if (NodeNumber == 0)  {
   in("sum of all Ys", ? Tmp);
   printf("%d\n",Tmp);
}
```

Assembly Language: MPI

1. MPI: Message Passing Interface

   The difference between MPI and sockets is that sockets support network packets, and MPI supports messages. MPI guarantees to process the full message. Sockets do not.

2. Note that “man 2 read” and “man 2 write” say that read and write may fail with the error EAGAIN or EINTR.

3. Calls to MPI_Send and MPI_Recv should not fail, and if they do fail (e.g. for reasons of hardware failure), then MPI is entitled to kill your entire application.

MPI Quick Reference

http://www.cslab.ntua.gr/courses/common/mpi-quick-ref.pdf

MPI Design

Along with reading the general overview of MPI below, you may want to look over some of the enrichment material listed here.

- History (comparison with PVM): standards, versus reference implementation
- MPI Quick Reference
- MPI is a standard:
  MPI 1.1: `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`
  MPI 2.0: `http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html`

MPI Design: seven layer model

The original MPI defines seven layers.

1. Point-to-point communication (send, receive, probe)
2. Collective communication (broadcast, scatter, gather, reduce)
3. Derived data types (support for describing struct, and other more complex data types for messages)
4. Groups (subset of nodes participating in MPI computation
5. Communicators (like a namespace; necessary for parallel libraries calling MPI simultaneously with main application; otherwise, what would happen if the application receives a message destined for the library)
6. Topology (define network topology to MPI; for example, how many hops between nodes)
7. Environment and Miscellaneous

MPI Design: standard

MPI is a standard. The MPI standard does not guarantee:

1. Interoperation (compatibility of two MPI dialects talking to each other)
2. Heterogeneous computation (distinct architectures, software or hardware, on the nodes of the computation)
3. Thread-safety (if two threads call MPI routines at the same time, MPI is allowed to crash)
4. Fault-tolerance (if a node or link fails, MPI is allowed to crash)

Unlike sockets, MPI views a message as atomic (similar to a packet in networking).

MPI Design: hello, world!

First, the obligatory ”Hello, world!” example:

```
#include <string.h>
#include "mpi.h"
int main(int argc, char **argv) {
  int myrank, num_processes, rank;
  char buf[100];
  char send_msg[] "Hello, world!";
  MPI_Init(&argc, &argv);  /* Allow MPI to see and modify the comma
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_processes);
  if (myrank == 0) /* If I'm the first process (often the console)
    for (rank = 1; rank <= num_processes, rank++)
      MPI_Send( send_msg, strlen(send_msg)+1, MPI_CHAR, rank, 0, MP
  else /* See [below|#ReceiveArray] for variation when length of re
    MPI_Recv( buf, strlen(send_msg)+1, MPI_CHAR, 0, 0, MPI_COMM_WOR
  MPI_Finalize();
}
```

MPI Design: message organization

MPICH provides a reference implementation for MPI. MPI can be understood according to the type of message that it sends across a network. A typical implementation would implement a message as follows.

```
-----------------------------------------------------------
| Communicator | Tag | Source | Destination | Size |  ... DATA ..
-----------------------------------------------------------
```

Compare the internal form above to a typical MPI call:

```
MPI_Send( void *buf, int count, MPI_Datatype dtype, int dest,
          int tag, MPI_Comm comm)
```

MPI Design: Count

Note that the message refers to

- Size (bytes); as opposed to
- Count (number of elements in array)

At the application level, one wishes to work with Count. This is important, because a double on one machine may have different size than a double on another machine. By having the application specify that the message contains two doubles, one delegates to a lower layer the problem of translating, if, for example, the first machine uses 8 bytes for a double, and the second machine uses 10 bytes for a double.

Hence, the potential for heterogeneous computing has driven the MPI standard. The application specifies a count, which is architecture independent. When sending messages across the network, MPI is forced to instead use a size, since most networking system calls require knowledge of size, rather than count.

MPI Design: rank

The source and destination of the message are specified as a *rank*. This is a unique integer, similar to a process id. Generally, the process started at the console is rank 0, and the remaining processes are numbered consecutively.

MPI Design: communicator

The Communicator is initially one of `MPI_COMM_WORLD` or `MPI_COMM_NULL`. `MPI_COMM_WORLD` is a namespace in which all processes participate, while no process participates in `MPI_COMM_NULL`. One can construct a new group (subset of processes in the computation), and associate it with a new communicator.

MPI Design: tags

Tags are integers specified by the user. Tags are generally used to:

1. distinguish different application-defined message formats (e.g. `struct int x; int y` vs. `struct char x; double y`);
2. distinguish different array lengths (e.g. `int x[27]` vs. `int x[43]`)
3. distinguish different tasks (e.g. taskA, taskB, taskC, exitComputation)

MPI Design: Non-overtaking

If two messages have the same source, communicator and tag, then MPI guarantees that they will be received in order. This is sometimes called the *non-overtaking* property.

MPI Design: Pattern 1

Here are three common patterns for receiving a message, corresponding to the three common patterns above for use of tags. Note that these patterns depend heavily on the non-overtaking property:

```
/* Receive any of several types of struct, depending on tag: */
MPI_Status *status;
char *buf[10000]; /* large enough for any message format */
MPI_probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
tag = status->MPI_TAG;
switch(tag) {
  case 0: /* We assume that tag 0 implies that there were
             two calls to MPI_Send: int first, and then float */
          MPI_Recv( buf, 4, MPI_INT, status->MPI_SOURCE, status->MP
          MPI_Recv( buf, 2, MPI_FLOAT, status->MPI_SOURCE, status->
          break;
  case 1: MPI_Recv( buf, 3, MPI_DOUBLE, status->MPI_SOURCE, status-
          break;
  default: fprintf(stderr, "ERROR: unexpected tag: %d.\n", status-
```

MPI Design: Pattern 2


```
/* [Receive array of arbitrary length|ReceiveArray],  with using th
  (assume tag indicates number of ints being sent): */
MPI_Status status;
int count;
char  *buf [10000]; /* large enough for any array to be received */
MPI_probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
MPI_Get_count(status, MPI_INT, &count);
buf = malloc( count * sizeof(int) );
MPI_Recv( buf, count, MPI_INT, status->MPI_SOURCE, status->MPI_TAG,
```

MPI Design: Pattern 2 (cont.)


```
/* ALTERNATIVE:  Receive array of arbitrary length, _without_ need
MPI_Status status;
char *buf[10000]; /* large enough for any array to be received */
int count;
MPI_probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
/* MPI equivalent of doing sizeof() computations in C */
MPI_Get_count(status, MPI_INT, &count);
buf = malloc( count * sizeof(int) );
MPI_Recv( buf, count, MPI_INT, status->MPI_SOURCE, status->MPI_TAG,
```

MPI Design: Pattern 3


```
/* Execute any of several tasks, according to tags: */
MPI_Status status;
MPI_probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
tag = status->MPI_TAG;
switch(tag) {
  case 0: doTaskA();
          break;
  case 1: doTaskB();
          break;
  case 2:  MPI_Finalize(); /* Exit computation */
  default: fprintf(stderr, "ERROR:  unexpected tag: %d.\n", status
}
```

MPI Design: Common Bug 1

Note that when we use both `MPI_Probe` and `MPI_Recv`, we are careful to use the arguments `status->MPI_SOURCE` and `status->MPI_TAG`, and *not* `MPI_ANY_SOURCE` and *MPI_ANY_TAG*. This is important for avoiding bugs. If, for example, *MPI_Recv* uses *MPI_ANY_SOURCE*, then it may receive a completely different message from that analyzed in `MPI_Probe`. This negates the purpose of `MPI_Probe` and is responsible for many bugs.

MPI Design: Common Bug 2

Another bug to be aware of is when the total number of receives does not match the total number of sends, or if each of two processes with rank 0 and rank 1 execute the following.

```
MPI_Send( buf, count, MPI_INT, 1 - myrank, tag, MPI_COMM_WORLD);
MPI_Recv( buf, count, MPI_INT, 1 - myrank, tag, MPI_COMM_WORLD,
&status);
```

Since `MPI_Send` is a blocking call, for larger messages, this can lead to a traditional *deadly embrace* (deadlock).

MPI Design: Seven Layers Revisited

1. Many MPI calls in the point-to-point layer also have non-blocking alternatives (e.g. `MPI_Isend`, `MPI_Iprobe`, `MPI_Irecv`).
2. Beyond this, there is the collective communication layer, which adds another argument, `root`.

As an example,

```
MPI_Bcast(void buf, int count, MPI_Datatype datatype, int
root, MPI_Comm comm)
```

will broadcast `buf` from the process with rank `root` to all processes (including itself). Other collective communications allow for scatter, gather, reductions to a scalar (e.g. sum over all ranks), and barrier (proceed when all other processes have called the caller).

MPI Design: Further Observations

1. Since MPI is designed with the possibility of heterogeneous computing, MPI allows a user to declare new data structures (similar to struct), which can then be translated from host format to network format to the data format of a host with a different architecture. This facility is called derived datatypes.
2. A *group* is simply a subset of the participating processes. MPI provides MPI_GROUP_EMPTY and access to a group of all initially participating processes, and the ability to do unions and intersections of these groups and the current process. This is needed to support communicators in which only a subset of the processes participate.
3. A *communicator* is similar to the concept of a namespace in C++. Two messages with the same source and tag are distinguished if they have different communicators. A communicator is based on a group. The communicators MPI_COMM_WORLD and MPI_COMM_EMPTY are predefined. Communicators are useful to allow a library built with MPI to carry out a concurrent computation without confusing messages among processes on behalf of the library and messages among processes on

number of hops, etc.

5. Finally, environment utilities allow for things like startup and finish (`MPI_Init`, `MPI_Finalize`) and implementation limits (e.g. `MPI_TAG_UB` is an upper bound on the value of a tag).

Assembly Language: OpenMP

- http://www.llnl.gov/computing/tutorials/openMP/

Operating System Issues

1. resource discovery
2. task scheduling
3. dynamic load balancing
4. process migration
5. virtualization
6. checkpointing
7. fault tolerance
8. *there are other issues ...*

Cilk: Work-Stealing

Two Primitive Operations:

- spawn, sync

Example: Fibonacci numbers: $f_n = f_{n-1} + f_{n-2}$

```
cilk int fib (int n) {
  if (n<2) return (n);
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y)
  }
}
```

Linda: Tuple space

Three (plus two) Primitive Operations:

1. in (blocking read and delete), out (insert), rd (blocking read)
2. non-blocking versions: inp, rdp

Example: `http://heather.cs.ucdavis.edu/~matloff/Linda/NotesLinda.NM.h`

```
if (NodeNumber == 0) out("sum of all Ys",0);
in("sum of all Ys", ? Tmp);
Tmp += Y;
out("sum of all Ys",Tmp);
/* barrier code would go here */
if (NodeNumber == 0)  {
   in("sum of all Ys", ? Tmp);
   printf("%d\n",Tmp);
}
```

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

A package for easily writing applications for both distributed and shared memory architectures

`http://www.ccs.neu.edu/home/gene/topc.html`
`http://www.ccs.neu.edu/home/gene/topc-aux/topc-overview.pdf`

Meta-Computing: Legion

”The Core Legion Object Model”, *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing*, August 1996 Legion technical papers: `http://legion.virginia.edu/papers.html`

Performance Estimates and Effect of RAM

Models for predicting the time of a program tend to be of two forms.

1. There are theoretical asymptotic formulas with unknown asymptotic coefficients
2. There are instrumentations or simulations of existing code to estimate the running time of components

The MBRAM model is distinguished from both. It estimates the running time of a program based on pseudo-code and on architectural parameters of the target architecture.

It does not require code on the target architecture, and yet there are no asymptotic coefficients!!!

CAVEAT: The MBRAM model is valid only for memory-bound programs.

ACCURACY: The MBRAM model consistently underestimates the running time in a range from 10% to 44% in our test suite. By adding a bias of 22%, the MBRAM model has an **accuracy of** $\pm 22\%$.

MBRAM Model Parameters

- the sequential bandwidth of RAM (β_1);
- the random access bandwidth of RAM (β_2: the bandwidth, assuming that a cache block is loaded from RAM with each successive cache block coming from a random location in RAM);
- the cache size (C) and cache block size (B) for the largest level of cache; and
- the size of the branch misprediction pipeline.

Experimental Results (Perm. Mult. and Integer Matrix Mult.)

CPU/RAM	Time (new Fast alg., s)		Time (traditional alg., s)	
	Experiment	Predicted	Experiment	Predicted
2.66 GHz Pentium 4 / DDR-266 RAM	0.042	0.042	0.159	0.147
1.7 GHz Pentium 4 / PC-133 RAM	0.060	0.047	0.176	0.158
0.6 GHz Pentium III / PC-100 RAM	0.131	0.087	0.097	0.083
0.35 GHz Pentium II / PC-66 RAM	0.222	0.151	0.148	0.143

1: Fast multiplication of two random permutations on 1,048,576 points (4 MB per permutation)

```
for (i=0; i<1048576; i++) Z[i]=Y[X[i]];
```

CPU/RAM	Time (Mat. Dim. $n = 500$)		Time (Mat. Dim. $n = 4000$))	
	Experiment	Predicted	Experiment	Predicted
2.66 GHz Pentium 4 / DDR-266 RAM	1.67*	0.53 (1)	11,241.23	8,743.91 (2)
1.7 GHz Pentium 4 / PC-133 RAM	1.47*	0.59 (1)	11,306.00	9,879.16 (2)

2: Matrix Multiplication over Integers (word size $w = 4$); Predictions according to MBRAM formula (1) wn^3/β_2 (when $wn > B$, $Bn < C$)); or (2) $wn^3/\beta_1 + Bn^3/\beta_2$ (when $Bn \geq C$)

*Computation is CPU-bound

Experimental Results (Sorting using 7 $O(n \log n)$ algorithms)

Sorting Algorithm	Exper. (s)	Pred. (s)	MBRAM formula
Quicksort	2.40	1.51	$\frac{2wN}{\beta_1}(\log_2 N - \log_2 C + 1) +$
Mergesort	3.14	1.75	$(\frac{wN}{\beta_2} + \frac{2wN}{\beta_1})(\log_2 N - \log_2 C$ $+\frac{N}{2}n$
Heapsort	24.88	16.83	$\frac{B}{\beta_2}N(\log_2 N - \log_2(C/w) + l$ $+mN$
[Input data uniformly distributed between 0 and $N = 8$ Meg]			
Simple Bucket Sort ($b = 64$)	0.62	0.47	$\frac{3wN}{\beta_2}(\lceil \log_b N \rceil - \lfloor \log_b(C/$
Distribution-count Bucket Sort ($b = 64$)	0.92	0.57	$\frac{3wN}{\beta_2}(\lceil \log_b N \rceil - \lfloor \log_b(C/2) \rfloor + 1)$ $+\frac{wN}{\beta_1}(\lceil \log_b N \rceil - \lfloor l$
Simple Radix Sort ($b = 64$)	0.77	0.47	$\frac{3wN}{\beta_2}\lceil \log_b N \rceil$
Distribution-count Radix Sort ($b = 64$)	1.01	0.60	$(\frac{wN}{\beta_1} + \frac{3wN}{\beta_2})\lceil \log_b N$

Prediction and Experiment for Various Sorting Algorithms;
1.7 MHz Pentium 4 with PC-133 RAM; input data is uniformly distributed

And Now for Something Completely Different: Effect of RAM, Netw

Some Personal Experiences from High Performance Computing

> Whatever women must do, they must do twice as well as men to be thought half as good. Luckily, this is not difficult.

— Charlotte Whitton
(born 1896), Canadas first woman mayor; elected in 1951, mayor of Ottawa

History of Rubik's Cube

- Invented in late 1970s in Hungary.
- In 1982, in *Cubik Math*, Singmaster and Frey conjectured:

  No one knows how many moves would be needed for "God's Algorithm" assuming he always used the fewest moves required to restore the cube. It has been proven that some patterns must exist that require at least seventeen moves to restore but no one knows what those patterns may be. Experienced group theorists have conjectured that the smallest number of moves which would be sufficient to restore any scrambled pattern — that is, the number of moves required for "God's Algorithm" — is probably in the low twenties.

- Current Best Guess: 20 moves suffice
  - States needing 20 moves are known

History of Rubik's Cube (cont.)

- 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)
- `http://news.google.com/news?hl=en&ned=&q=rubik\%27s+cube&btnG=Sear`
- D. Kunkle and G. Cooperman, "Twenty-Size Moves Suffice for Rubik's Cube", *International Symposium on Symbolic and Algebraic Computation* (ISSAC-07), 2007, *to appear*

Ideal Algorithm to Bound Moves Needed for Rubik's Cube

Breadth-First Search

1: Create hash array with several times 4.3×10^{19} entries
2: Add trivial (solved) state to open list and to hash array
3: Set level $\leftarrow 0$
4: **while** Open list not empty **do**
5: Increment level; move open list to frontier; and set open list to empty set
6: **while** Frontier not empty **do**
7: Remove first element of frontier and find its neighbors
8: If a neighbor is not in hash array, then add it both to hash array and to open list
9: Return level (number of moves needed in worst case)

Two Primary Techniques Used

1. Use of Large amounts of intermediate disk space (7 TB) for hash array (for duplicate elimination)
2. Fast multiplication of symmetrized cosets ($>$ 10,000,000 multiplies per second)

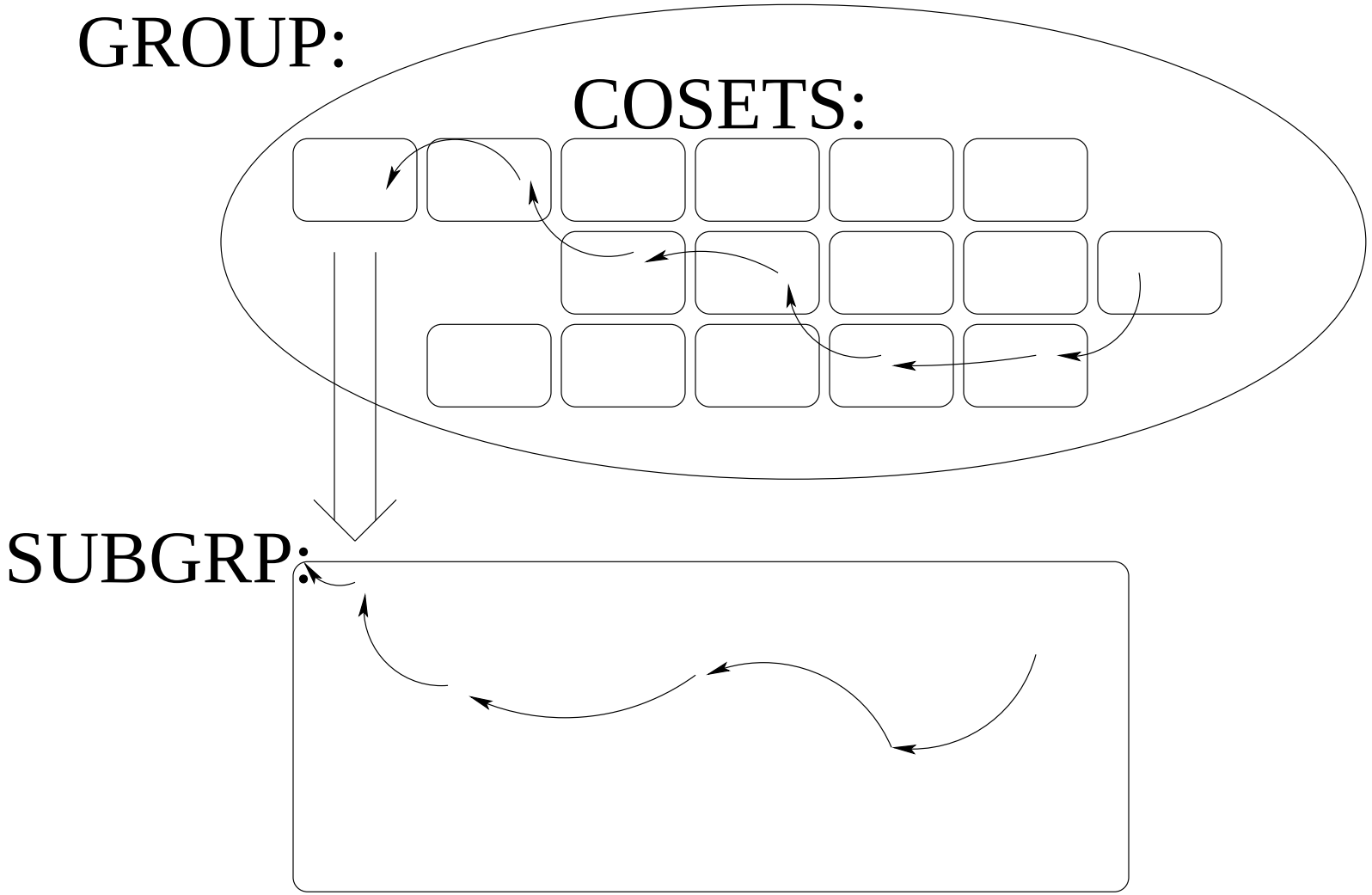
Fast Multiplication: $>$ 10,000,000 mults/second

- Table-based multiplication; Form smaller subgroups, factor each group element into the smaller subgroups; Use tables for fast multiplication among the small subgroups
- Tables are kept mostly in L1 cache; Most subgroups have less than 100 elements; Multiplication table has $< (100)^2$ elements, or $< 10,000$.
- Group of Rubik's cube
  - Group of permutations acting only on corner cubies
  - Group of permutations acting only on edge cubies
    - Flips of the two faces of each edge cubies (while holding location of edge cubie fixed)
    - Moving edge cubies (while ignoring fips of the two faces)

Fast Multiplication (cont.)

- Moving edge cubies (while ignoring fips of the two faces)
  - Moving edge cubies using half-twists (180 degrees) only:
    - Half-twists split the 12 edge cubies into three invariant subsets, each containing 4 edge cubies (can't move edge cubie from one subset to the other using only half-twists)
  - Moving edge cubies using quarter-twists (but "divided by" half-twists: using the group theory concept of cosets and normal subgroups)

Divide and Conquer: Groups, Subgroups and Cosets



Groups, Subgroups and Cosets

1. Rubik's cube forms a natural mathematical group
   (a) Identity transformation (do nothing)
   (b) Inverse transformation (do opposite of previous move)
   (c) Associativity (do first two moves, then third; or do first move; then next two)
2. The set of all transformations generated by quarter- or half-turn twists is a group.
3. A subgroup is a subset of transformations, such that composition of transformations stays within the subgroup (example: all transformations achievable by half twists (180 degrees))
4. If a group G of size $|G|$ has a subgroup H of size $|H|$, then there are $|G|/|H|$ cosets. Given a transformation $g \in G$, each coset Hg is a set of transformations given by:

   an arbitrary transformation from H followed by the transformation g

Symmetries and Fast Multiplication

1. Rubik's cube has 48 symmetries that take quarter-twist moves to quarter-twist moves:
   (a) 24 of them correspond to physically turning the cube to show a different face
   (b) There is also an inversion symmetry (take each cubie to its opposite one).
   (c) Combining inversion with the original 24 yields 48 symmetries.
2. A symmetrized coset is a collection of cosets that are equivalent under symmetries.
3. The advantage of symmetrized cosets is that if a given coset is *n* moves away from the trivial coset (original subgroup), then so is every other coset of the symmetrized coset.

Fast Multiplication

Using the idea of a small normal subgroup of the given subgroup, one can derive a fast table-based lookup of the product of a symmetrized coset by a generator. The key idea is that the small normal subgroup and the number of cosets of the normal subgroup are both small enough to fit in L2 cache (modulo some mathematical manipulations). *The details are outside the scope of this talk.*

Sizes of Subgroups, Numbers of Cosets

The theory is clear enough, but what systems techniques are needed to make it work?

1. A subgroup based on 180 degree twists (only 663,552 states: only 15,752 states after symmetries)
2. An efficient perfect hash function on cosets (based on ideas of mixed radix)
3. *Only* 2.8×10^{12} cosets (and a hash array that's not much larger)
4. Storing only 4 bits per state for each coset: level of coset in breadth-first spanning tree.

Bottom line: hash array is small multiple of 1.4×10^{12} bytes; how to store and access it?

Algorithm to Bound Number of Moves Needed for Rubik's Cube

1: Initialize array of symmetrized cosets with all levels set to *unknown* (four bits per coset).
2: Add trivial coset to array; set level ℓ to 0.
3: **while** previous level had produced new neighbors, at next level **do**
4: {Generate new elements from the current level}
5: Let a segment be those nodes at level ℓ among N consecutive elements of the array.
6: Scan array starting at beginning.
7: **while** we are not at the end of the array, extract next segment of array and **do**
8: **for** each node at level ℓ (representing a symmetrized coset) **do**
9: **for** each generator **do**
10: Compute product by fast multiplication.
11: Compute hash index of product.
12: Save hash index in bucket b, where b is the high bits of the hash index. Note, we only save the low order bits of the hash index not encoded by the bucket number. (This value fits in four bytes.)

13: If bucket b is full, transfer it (write it) to a disk file for bucket b.
14: Transfer all buckets to corresponding disk files.
15: {Now merge buckets into array of symmetrized cosets.}
16: **for** each bucket b on disk **do**
17: Load portion of level array corresponding to bucket b into main memory.
18: **for** each buffered element on disk for this bucket (read in large chunks) **do**
19: Look up corresponding level value in array.
20: If a value already exists for the element, it is a duplicate. Otherwise, set its level to ℓ.
21: Write portion of level array back to disk.
22: Increment level ℓ.

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)

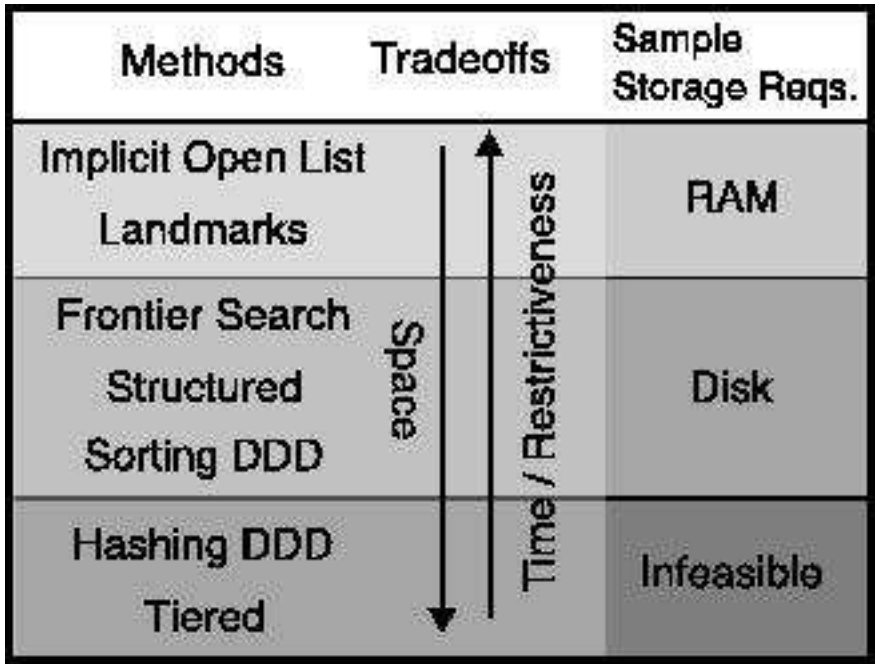
Initial Testbed: large search and enumeration

- 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

Space-Time Tradeoffs using Additional Disk



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. . . .

Outline

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. ...

Outline

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. . . .

Outline

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 as a vector of dimension 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)

Relative Speeds: CPU, RAM, Disk and Network

CPU bandwidth	2,400 MB/s	(3 GHz × 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 $1/2^k$.

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")

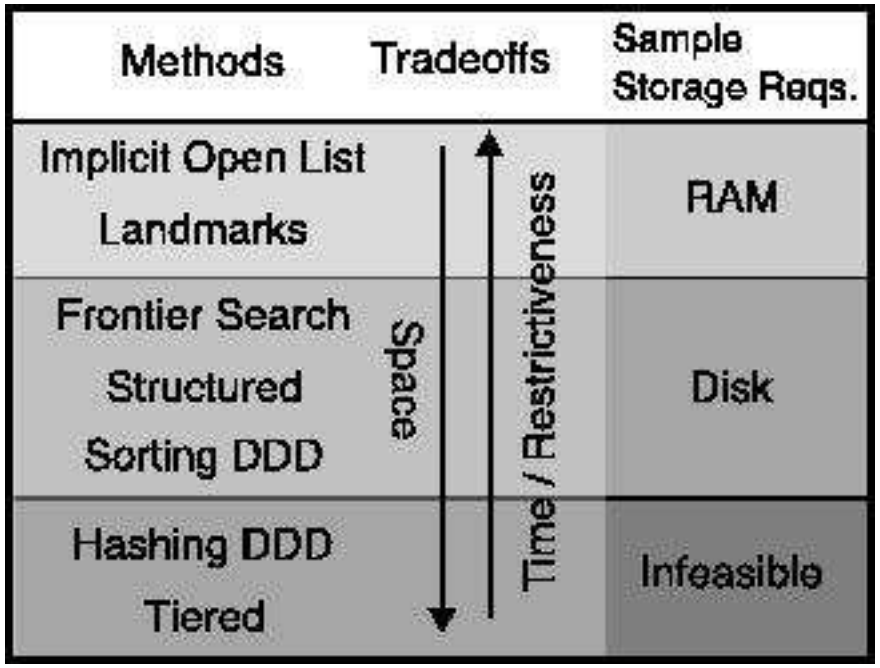
How close can we get?

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.

Space-Time Tradeoffs Again



Why is Larger Memory Important?

1. **Devil's Advocate:** Problems are mostly in P or NP. Problems in P are doable. Problems in NP blow up so fast, that very few problems will be solvable just because we have 100 times as much memory.
2. **Answer:** Why are faster CPUs important? Problems in P are doable. Problems in NP blow up so fast, that very few problems will be solvable just because CPUs are 100 times faster.

How do we Use Very Large Memory?

Question: What does a program want from large memory?
Answer: Fast, large look-up table for key-value pairs. **Question:** What does one want from key-value tables?

1. Dictionary Operations (à la Cormen, Leiserson, Rivest, Stein): Add, Delete, Next Value, Random Value, etc.
2. Convert table format to more efficient format among:
   (a) Convert to larger hash table (or smaller one)
   (b) Convert to sorted array
   (c) Convert to binary tree
   (d) . . .

Richness of Algorithmic Techniques

- Based on:
  E. Robinson, D. Kunkle, and G. Cooperman,
  "A Comparative Analysis of Parallel Disk-Based Methods for Enumerating Implicit Graphs",
  Proc. of PASCO-07 (Parallel Algebraic and Symbolic Computation, 2007, *to appear*

Richness of Algorithmic Techniques

1. Landmarks: Instead of storing all states in the graph, store only a small fraction (the *landmarks*)

   (a) Non-landmark states can be stored more compactly as a shortest path to a landmark.

   (b) Non-landmark states can simply *not* be stored. (Riskier, but by starting from each landmark, one can recreate all other states. When we approach a second landmark, we halt the expansion from the first landmark.)

Richness of Algorithmic Techniques (cont.)

2. Two-bit trick: (reviewed earlier)

   (a) Requires perfect, reasonably dense hash function

   (b) Can be combined with ideas from Bloom Filters

   (c) Added benefit of finding levels within breadth-first spanning tree

Algorithms: Delayed Duplicate Detection

3. Delayed Duplicate Detection

   (a) Sorting-Based DDD

   (b) Hash-Based DDD

   (c) Tiered Duplicate Detection

Algorithms: Sorting-Based Duplicate Detection

3. Delayed Duplicate Detection

(a) *Sorting-Based DDD:*

i. Save all frontier states on disk, and externally sort it.

ii. Then compare with list of known states.

(b) Hash-Based DDD

(c) Tiered Duplicate Detection

Algorithms: Hash-Based Duplicate Detection

3. Delayed Duplicate Detection

(a) Sorting-Based DDD

(b) *Hash-Based DDD:*

i. Hash states.

ii. Use upper hash bits to determine a bucket of states on disk.

iii. Save frontier states in appropriate bucket.

iv. Lower hash bits can be used to compare against known states in a corresponding bucket.

v. Tiered Duplicate Detection

Algorithms: Tiered Duplicate Detection

3. Delayed Duplicate Detection

(a) Sorting-Based DDD

(b) Hash-Based DDD

(c) *Tiered Duplicate Detection:* Use frontier queue, and collision queue; along with a hash array of 1-bit slots.

i. If hash slot is 0, it must be new: add to frontier.

ii. If hash slot is 1, it may be old or it may be new (hash collision with another state): add to collision queue.

iii. Later, sort and process (relatively small) collision queue.

iv. Note: Can stop and process collision queue at any time: either *before* current frontier level is complete, or *after* completing several levels.

Algorithms: Structured Duplicate Detection

4. Structured Duplicate Detection: Use structural information about the state space. (more specialized to problem)

Algorithms: Frontier Search

5. Frontier Search (Korf)

   (a) Normally, need only check against duplicates in the current level, next level, or previous level.

   (b) Frontier search eliminates the need to check against duplicates in the previous level.

   (c) Upon processing a state from the current level, find neighbors and at each neighbor, store a bit indicating the inverse generator/move compared to the current one used

Algorithms: Implicit Open Search

6. Implicit Open Search

(a) Used for Rubik's cube

(b) Open List exceeds available disk space

(c) Relies on 2-bit trick, and delayed duplicate detection

(d) Scans entire hash array stopping at frontier states