Computing & Information Science 4205

Effective Use of High Performance Computing

Steve Lantz

slantz@cac.cornell.edu

www.cac.cornell.edu/~slantz

Course Overview

Goals for CIS 4205

- Introduction to HPC – Practical experience for your research
- Finding the parallelism in your work
- Measuring speedup & efficiency and the factors that affect it
- Writing & debugging parallel code (MPI & OpenMP)
- Exposure to using production HPC systems at Cornell
- Effective techniques for inherently (“embarrassingly”) parallel codes
- Critical analysis of current & future HPC solutions

A Little About Me

- **Role at the Cornell Center for Advanced Computing**
  - Senior Research Associate: consulting, training, advising, & participating
  - Involved in HPC around 25 years
- **Background**
  - Education
  - Experience
- **Research interests**
  - Numerical modeling and simulation
  - Fluid and plasma dynamics
  - Parallel computing

A Little About You*

- **Fields of study and/or research interests**
- **Programming experience**
  – C, C++, Fortran, Others…
  – Scripting languages
- **Practical experiences**
  – Ever written a program from scratch for your research?
  – Ever had to work with someone else's code?
  – Which was harder? Why?
- **HPC experience**
- **Your goals for this course**

*** - ("A Little Bit Me, A Little Bit You" – Monkees, 1967)**

Assignments

- **Check the course website before every class**
  - http://www.cac.cornell.edu/~slantz/CIS4205
- **Assignments are due on date specified**
- **Assignments should be emailed to me**
  - slantz@cac.cornell.edu
- **Assignments can be done on any HPC system**
  - Windows, Linux, Macintosh OS X
  - HPC system must have MPI, OpenMP & batch scheduling system
- **Access to CAC HPC systems is available**

Connecting to CAC Resources

- **All students' Cornell NetIDs will be added the course account**
  - Everyone will have the option to use CAC resources for assignments
- **Accessing CAC machines**
  - http://www.cac.cornell.edu/Documentation/Linux.asx
  - http://www.cac.cornell.edu/Documentation/Linux
- **Poll: what's your background?**
  - Familiar with Linux?
  - Familiar with ssh and X Windows?
  - Comfortable with text editors (emacs, vi)?

Where and How to Find Me

- **Physical and and virtual whereabouts**
  - Office: 533 Frank H. T. Rhodes Hall
  - Phone: 4-8887
  - Email: slantz@cac.cornell.edu

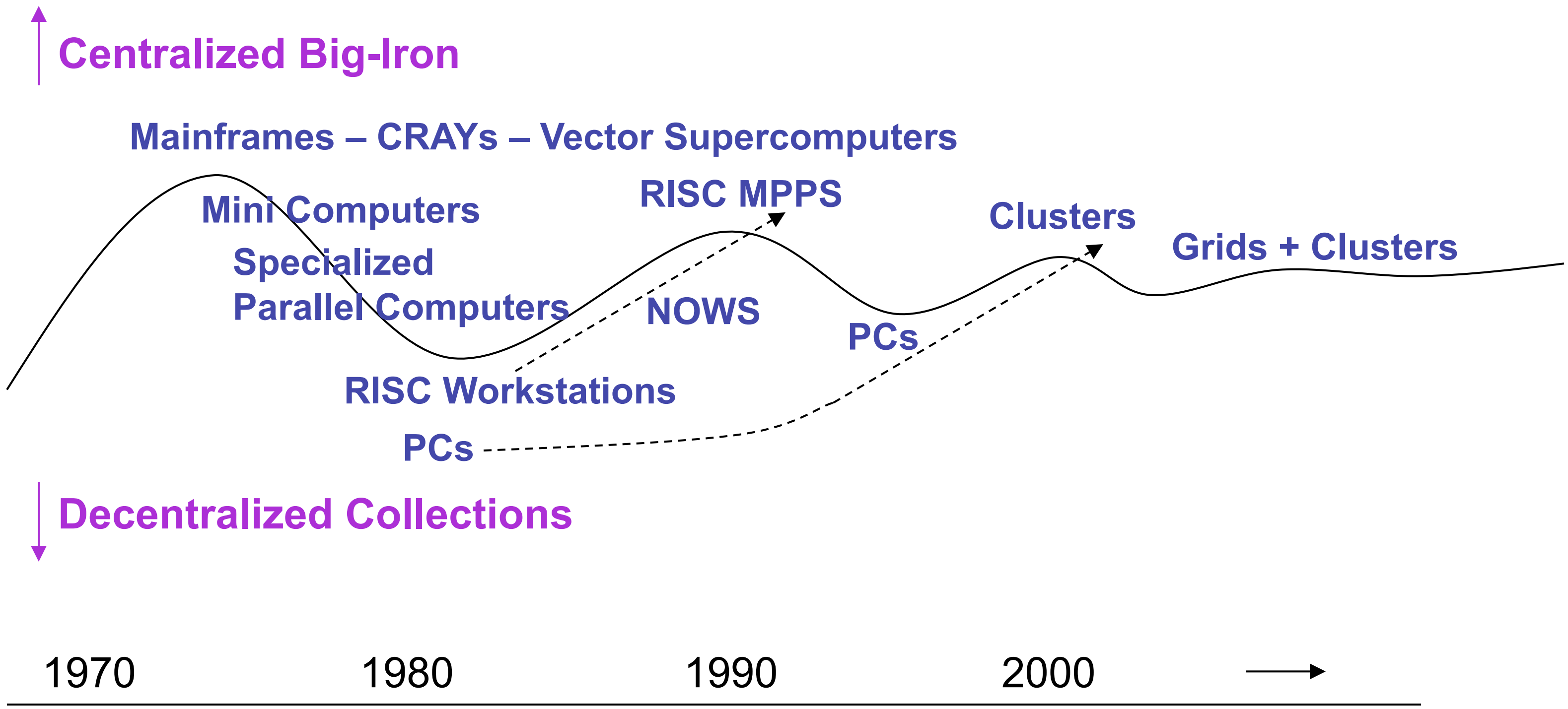
- **Office hours by appointment**

Introduction to High Performance Computing

Why HPC? Why Parallel Computing?

- **Want to have best possible time-to-solution, minimize waiting**
- **Want to gain a competitive advantage**
- **As expected, processor clock speeds have flattened out**
  - Not the end of Moore's law: transistor densities are still doubling every 1.5 years
  - Clock speeds limited by power consumption, heat dissipation, current leakage
  - Bad news for mobile computers!
- **Parallelism will be the path toward future performance gains**
  - Trend is toward multi-core: put a cluster on a chip (in a laptop)
  - Goes well beyond microarchitectures that have multiple functional units

Evolution of HPC



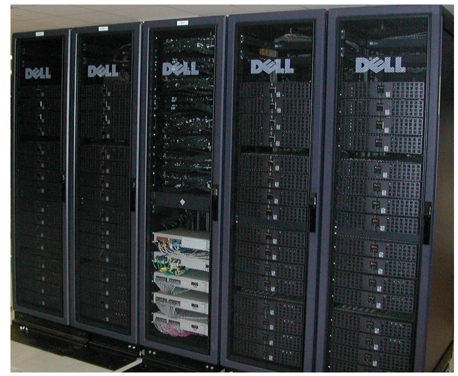
The Cornell Programmer's View

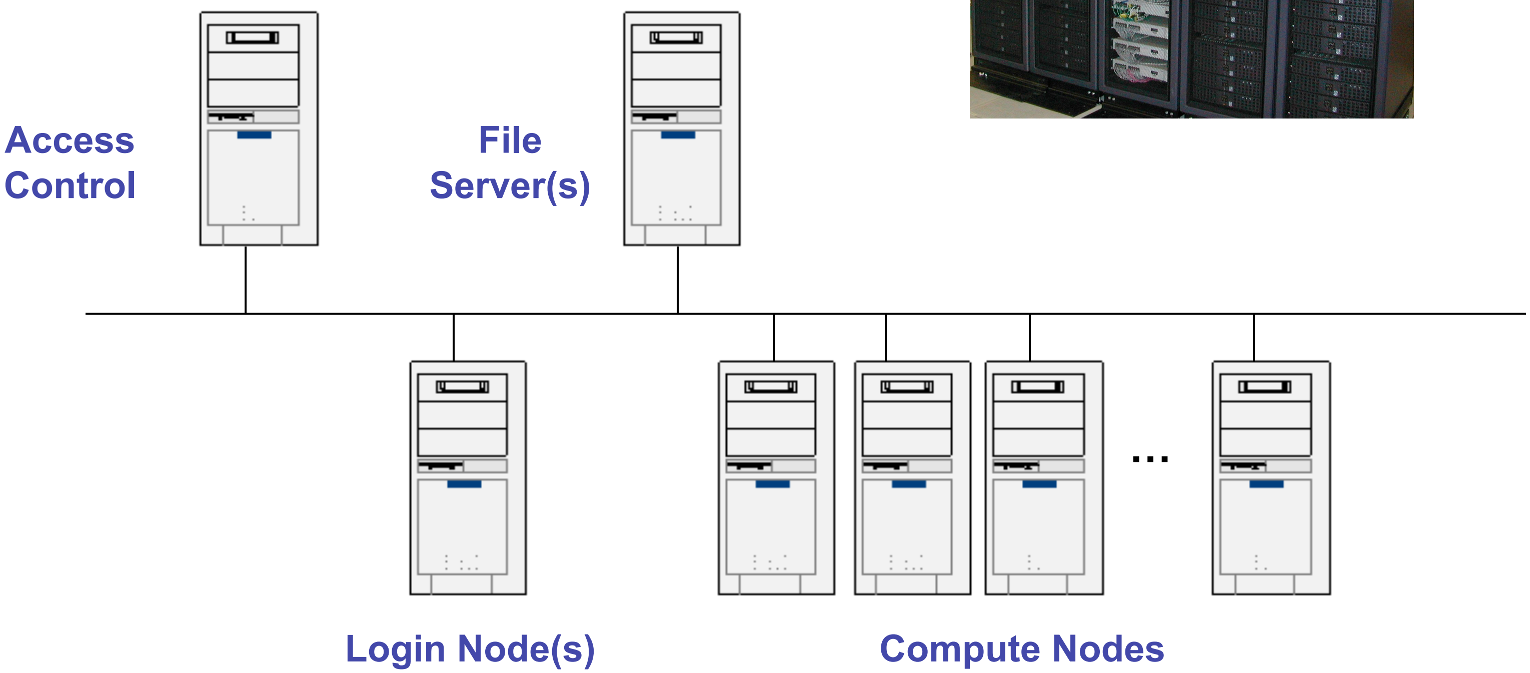
Timeframe *– hardware – parallel programming model*

- **Mid '80's** – IBM mainframe with attached Floating Point Systems array processors – IBM and FPS compiler extensions
- **Late '80's** – Interconnected IBM 3090 mainframes featuring internal vector processors – Parallel VS FORTRAN and APF compilers
- **Early '90's** – IBM SP1, rack-mounted RS6000 workstations with POWER RISC processors networked via multistage crossbar switch; KSR-1 from Kendall Square Research – PVM, MPL, MPI message passing libraries; HPF/KAP directives; KSR compiler for ALLCACHE "virtual shared memory"
- **Mid '90's** – IBM SP2 featuring POWER2 and P2SC – MPI (not a compiler)
- **Late '90's to present** – Several generations of Dell HPC clusters (Velocity 1, 1+, 2, 3), quad or dual Intel Pentiums running Windows, Red Hat Linux – MPI plus OpenMP compiler directives for multithreading

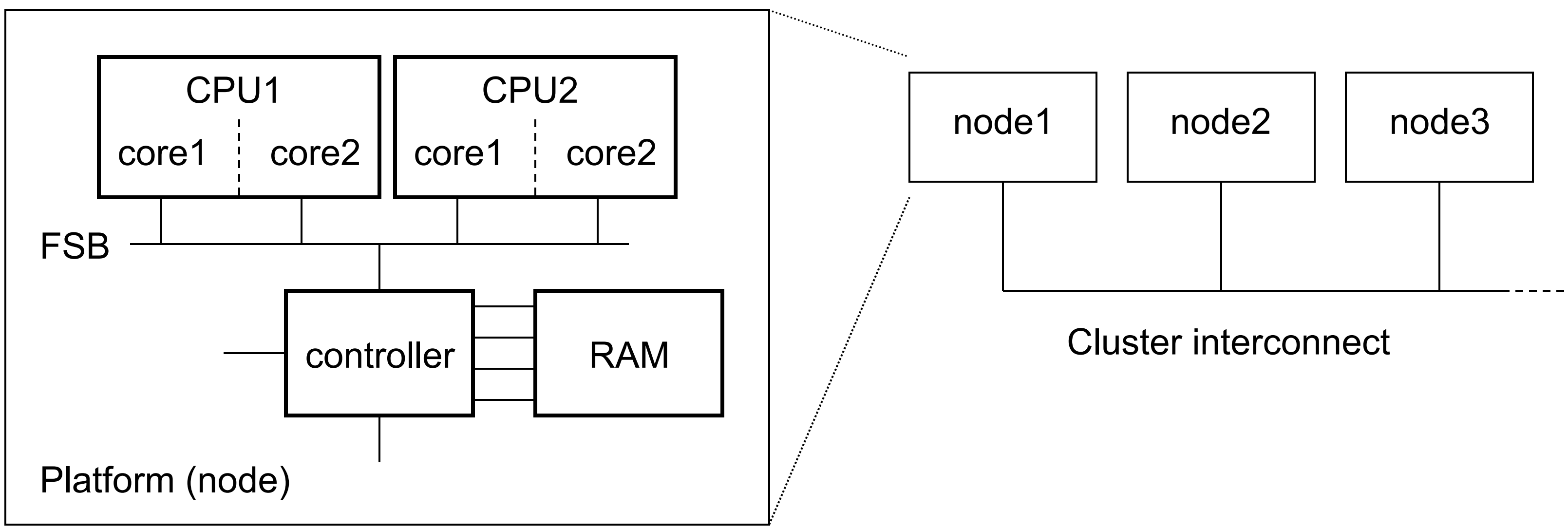
Current HPC Platforms: COTS-Based Clusters

COTS = Commercial off-the-shelf





Shared and Distributed Memory

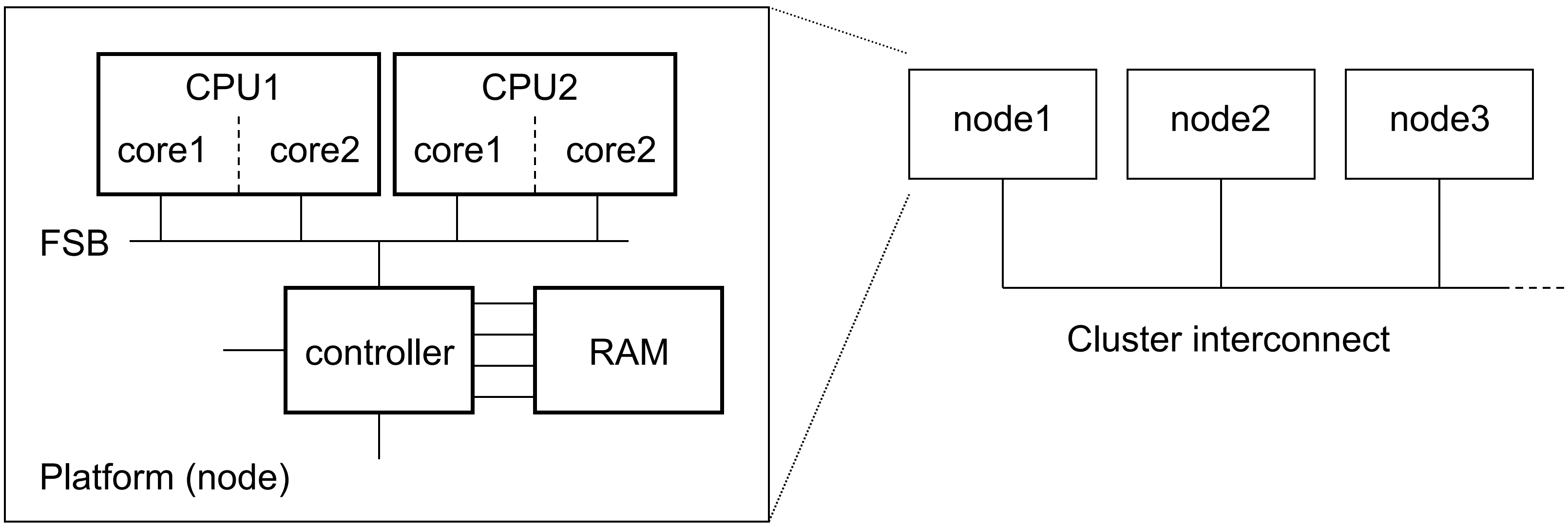


Shared memory on each node…

Distributed memory across cluster

Multi-core CPUs in clusters – two types of parallelism to consider

Shared and Distributed Memory



Shared memory on each node…

OpenMP
Pthreads
MPI

Distributed memory across cluster

MPI

OpenMP and MPI? In 2009?

Strengths:

- Adherence to carefully defined specifications (not standards per se)
- Specifications still under active development
- Cross-platform, multi-OS, multi-language (e.g., pypar in Python)
- Wide acceptance
- Time-tested with large existing code base
- Useful for both data and task (functional) parallelism

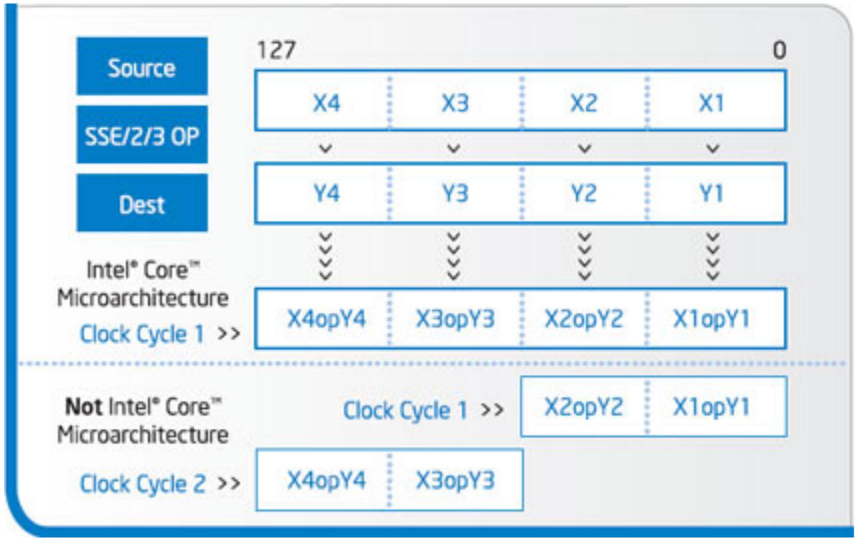
Weaknesses:

- Relatively low-level programming (though not as low as pthreads)
- Mindset taken from procedural languages (C, Fortran)

Where's My Parallel Compiler?

- **You've had it for years! "Serial" compilers produce code that takes advantage of parallelism at the *top* of the memory hierarchy**

Example: SSE(2/3/4) instructions operate on several floats or doubles simultaneously using special 128-bit-wide registers in Intel Xeons (vector processing)

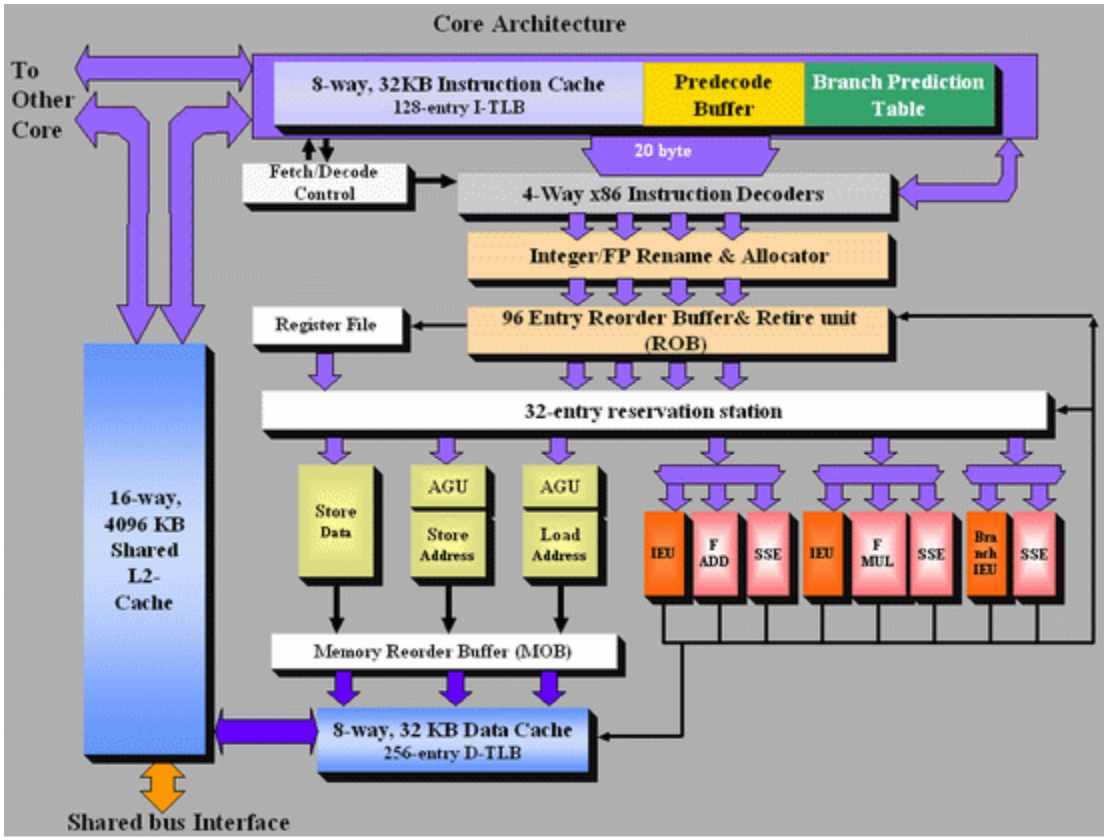


http://www.tomshardware.com/2006/06/26/xeon_woodcrest_preys_on_opteron/page9.html

Parallelism Inside the Intel Core

In the Intel Core microarchitecture:

- **4 instructions per cycle**
- Branch prediction
- Out-of-order execution
- 5 prefetchers
- 4MB L2 cache
- 3 128-bit SSE registers
- 1 SSE / cycle

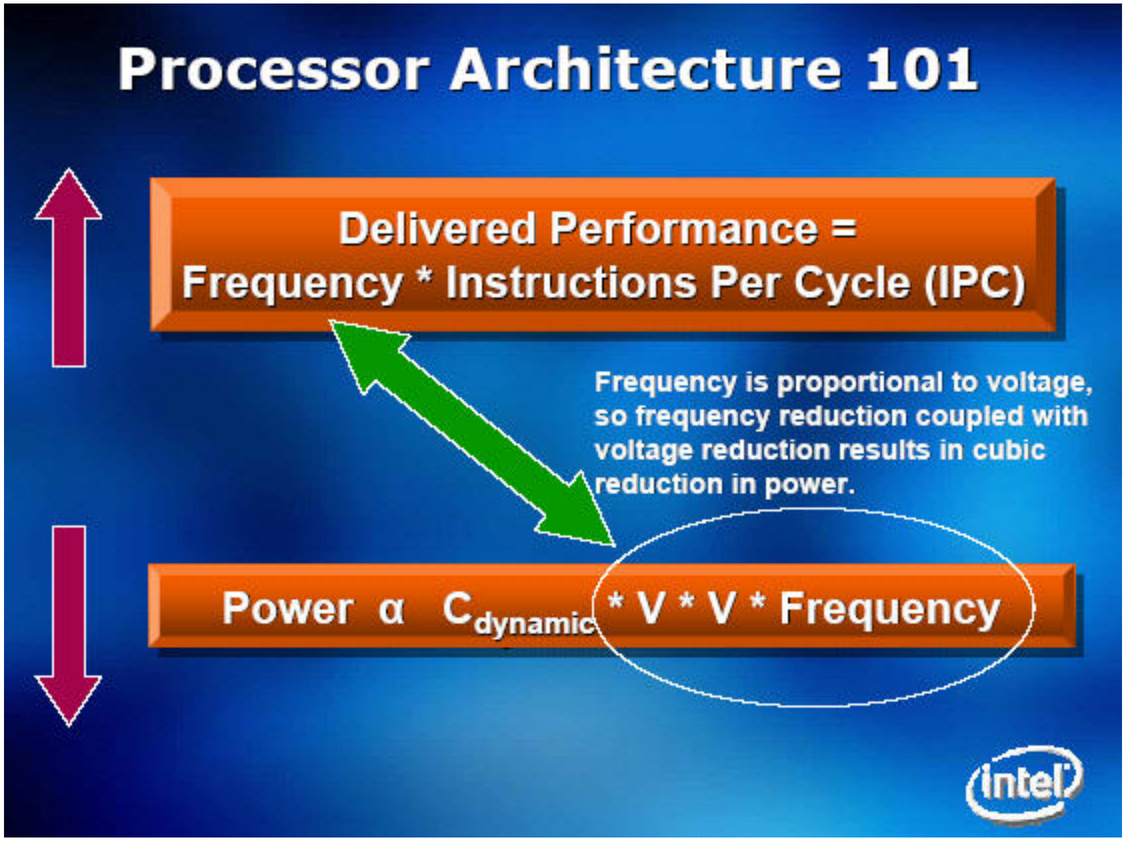


http://www.anandtech.com/cpuchipsets/showdoc.aspx?i=2748&p=4

Why Not Crank Up the Clock?

Because the CPU's power consumption goes up like the cube of frequency!

No wonder Intel tries so hard to boost the IPC…

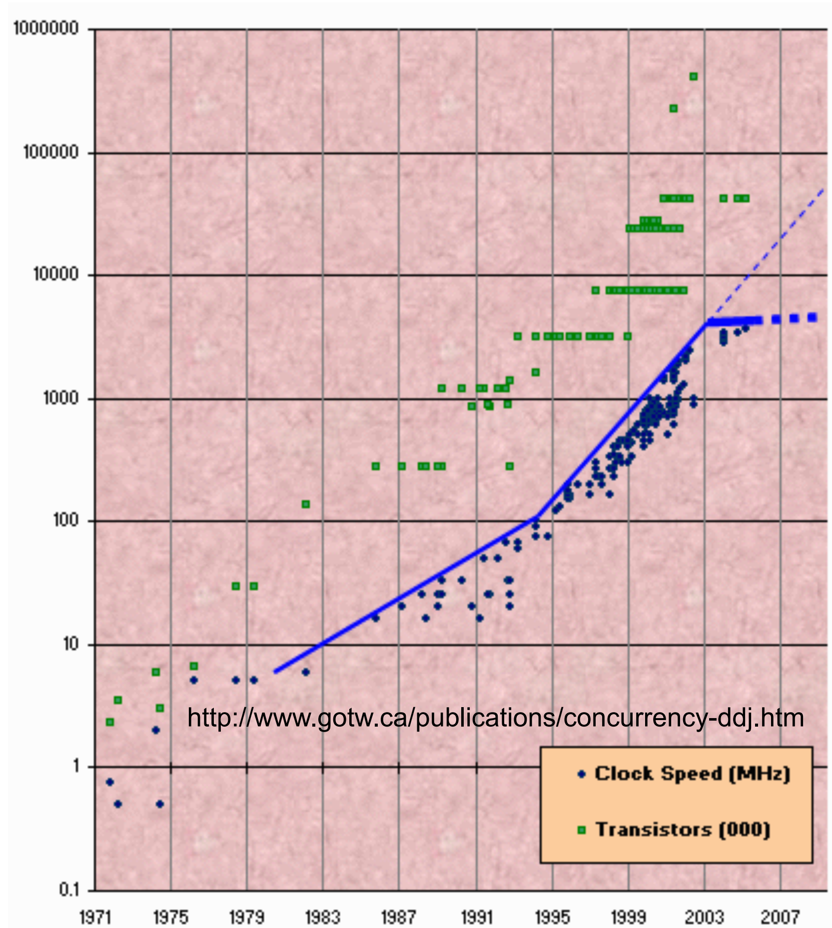


Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

OK, What's Next?

Trend toward growing numbers of cores per processor die

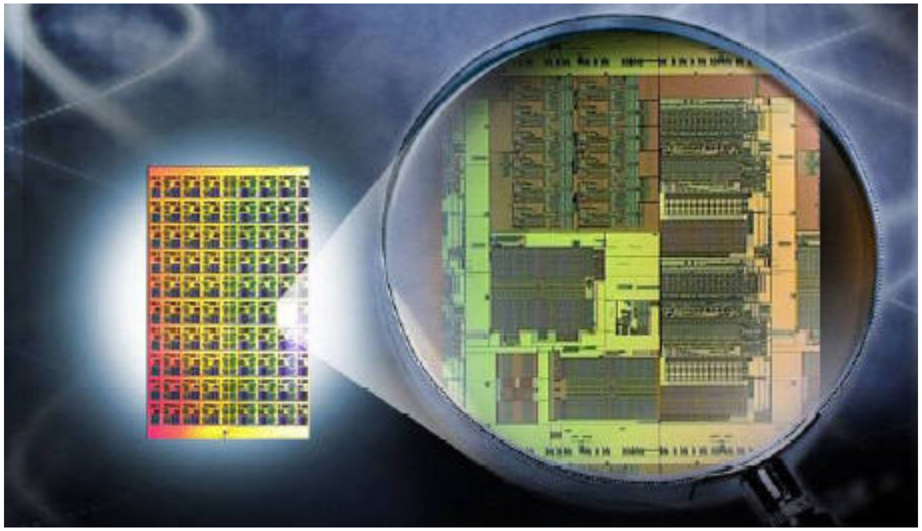
- **Moore's Law still holds; transistor densities are still increasing**
- **Higher densities don't translate into faster speeds due to:**
  - Problems with heat dissipation
  - Hefty power requirements
  - Leakage current
- **The "free lunch" of ever-increasing clock speeds is *over!***



Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

Signs of the Times…

- **IBM promotes BlueGene HPC line with 1000's of low-frequency, low-power chips (700 MHz PowerPCs)**
- **On 2/11/07, Intel announces successful tests of an 80-core research processor – "teraflops on a chip"**



Implications for Programmers

- **Concurrency will no longer be just desirable, it will be essential**
  - Previously it was the economical path to greater performance
  - Now it has become the physically reasonable path
- **Compilers (still) aren't the answer**
  - Degree of concurrency depends on algorithm choices
  - Need for high-level creation (as opposed to mere identification) of concurrent code sections
- **Improved programming languages could make life easier, but nothing has caught on yet**
- **Some newer languages (Java, C++) do have mechanisms for concurrency built in… but kind of clumsy to use…**
- **In the final analysis: TANSTAAFL**

Conclusions

- **Future processor technology will drive programmers to *create* and *exploit* concurrency in their software to get performance**
- **Some problems are inherently not parallelizable; what then?**
  - “9 women can't produce a baby in one month”
  - …But… what if the goal isn't just to produce one baby, but many?
  - “Embarrassing parallelism” isn’t so embarrassing any more
  - Examples: optimization of a design; high-level Monte Carlo simulation
- **Coding for efficiency and performance optimization will get more, not less, important**
  - Not all performance gains need to come from high-level parallelism
  - Nevertheless, parallelism needs to be designed into codes, preferably from the beginning

Parallel Computing: Types of Parallelism

Parallel Computing: Definitions

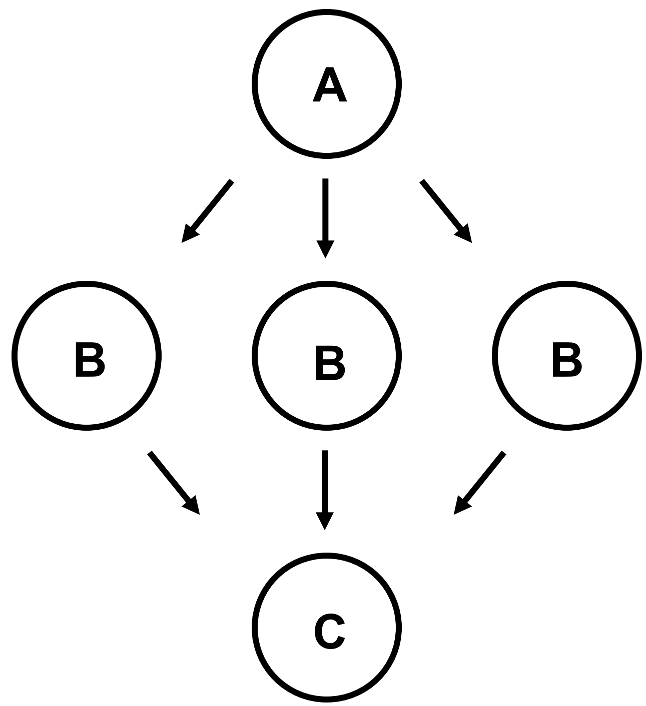
- **As we have seen, HPC necessarily relies on parallel computing**
- ***Parallel computing…***
  - Involves the use of multiple processors simultaneously to reduce the time needed to solve a single computational problem.
- **Examples of fields where this is important:**
  - Climate modeling, weather forecasting
  - Aircraft and ship design
  - Cosmology, simulations of the evolution of stars and galaxies
  - Molecular dynamics and electronic (quantum) structure
- ***Parallel programming…***
  - Is writing code in a language (plus extensions) that allows you to explicitly indicate how different portions of the computation may be executed concurrently
- **Therefore, the first step in parallel programming is to identify the parallelism in the way your problem is being solved (algorithm)**

Data Parallelism

Definition: when independent tasks can apply the same operation to different elements of the data set at the same time.

Examples:

- 2 brothers mow the lawn
- 8 farmers paint a barn



Data Parallelism

Partitions of the Data Can Be Processed Simultaneously

```
// initialize array values to 1
   a[]=1;
   b[]=1;
   c[]=1;

   for (i=0; i<3; i++)
    {
     a[i] = b[i] + c[i];
    }

// Serial Execution
   i=0 (a[0] = 2)
   a[0] = b[0] + c[0];
   i=1 (a[1] = 2)
   a[1] = b[1] + c[1];
   i=2 (a[2] = 2)
   a[2] = b[2] + c[2];

// Parallel Execution
   i=0 (a[0] = 2)            i=1 (a[1] = 2)            i=2 (a[2] = 2)
   a[0] = b[0] + c[0];       a[1] = b[1] + c[1];       a[2] = b[2] + c[2];
```

Data Parallelism
MPI Example

```
#include <stdio.h>
#include <mpi.h>
#include <malloc.h>

void main(int argc, char **argv )
 {
  int myid, numprocs;
  int i;
  int *a,*b,*c;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  a = (int *) malloc(numprocs*sizeof(int));
  b = (int *) malloc(numprocs*sizeof(int));
  c = (int *) malloc(numprocs*sizeof(int));

  for (i=0;i<numprocs;i++)
   {
    // initialize array values to i
    a[i]=i;
    b[i]=i;
    c[i]=i;
   }
  a[myid] = b[myid] + c[myid];
  printf("a[%d] = %d\n",myid,a[myid]);
  MPI_Finalize();
 }
```

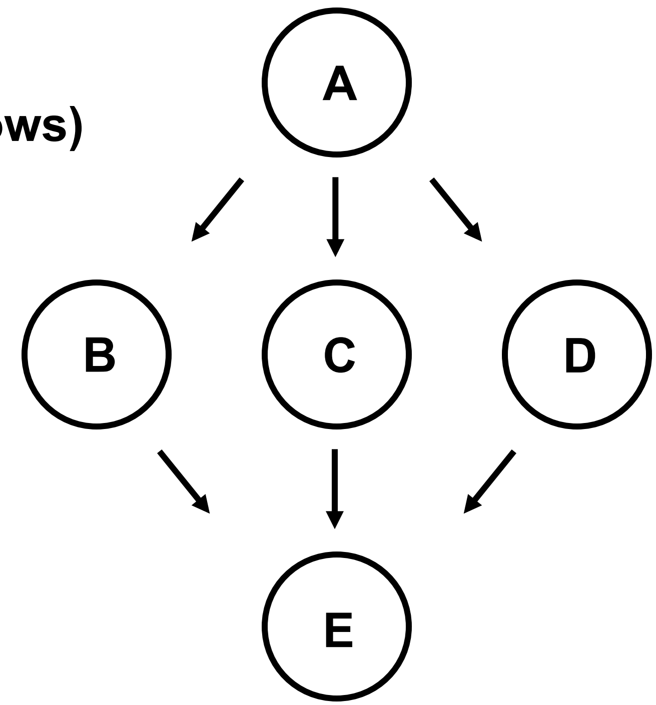
```
mpiexec -n 8 dp1.exe
3: a[3] = 6
4: a[4] = 8
5: a[5] = 10
7: a[7] = 14
0: a[0] = 0
2: a[2] = 4
6: a[6] = 12
1: a[1] = 2
```

Functional Parallelism

Definition: when independent tasks can apply different operations to the same (or different) data elements at the same time.

Examples:

- **2 brothers do yard work (1 rakes, 1 mows)**
- **8 farmers build a barn**



Functional Parallelism
Partitions of the Program Can Execute Simultaneously

```
// initialize values 0-9
   a[]=i;
   b[]=i;

// These different operations can happen at the same time
   for (i=0; i<10; i++)
    {
     c[i] = a[i] + b[i];
    }
   for (i=0; i<10; i++)
    {
     d[i] = a[i] * b[i];
    }

// This part requires solutions from above
   for (i=0; i<10; i++)
    {
     e[i] = d[i] - c[i];
    }
```

Functional Parallelism
MPI Example

```
#include <stdio.h>
#include <mpi.h>
#include <malloc.h>

void main(int argc, char **argv )
 {
  int myid, numprocs;
  int i;
  int iter=10;
  int *a,*b,*c,*d,*e;
  MPI_Status status;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  if (numprocs < 3)
   {
    printf("ERROR: This example requires 3 processes\n");
   }
  else
   {
    a = (int *) malloc(iter*sizeof(int));
    b = (int *) malloc(iter*sizeof(int));
    c = (int *) malloc(iter*sizeof(int));
    d = (int *) malloc(iter*sizeof(int));
    for (i=0; i<iter; i++)
     {
      a[i] = i;
      b[i] = i;
     }
    if (myid == 0)
     {
      MPI_Recv(c,10,MPI_INT,1,0,MPI_COMM_WORLD,&status);
      MPI_Recv(d,10,MPI_INT,2,0,MPI_COMM_WORLD,&status);
      e = (int *) malloc(iter*sizeof(int));
      for (i=0; i<iter; i++)
       {
        e[i] = d[i] - c[i];
        printf("e[%d] = %d\n",i,e[i]);
       }
     }
    else if (myid == 1)
     {
      for (i=0; i<iter; i++) { c[i] = a[i] + b[i]; }
      MPI_Send(c,10,MPI_INT,0,0,MPI_COMM_WORLD);
     }
    else if (myid == 2)
     {
      for (i=0; i<iter; i++) { d[i] = a[i] * b[i]; }
      MPI_Send(d,10,MPI_INT,0,0,MPI_COMM_WORLD);
     }
    else
     {
      printf("Process id %d not needed\n",myid);
     }
   }
  MPI_Finalize();
 }
```

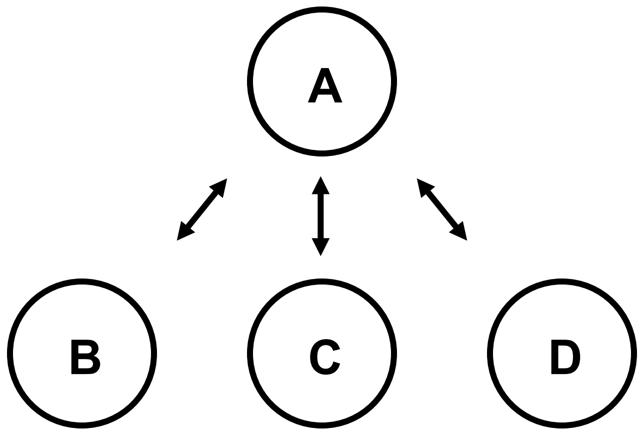
Task Parallelism

Definition: when independent "Worker" tasks can perform functions that do not need to communicate with each other, only with a "Master" or "Manager" process.

Such tasks are often called "Embarrassingly Parallel" because they can be parallelized with little extra work or thought.

Examples:

- Independent Monte Carlo Simulations
- ATM Transactions



Task Parallelism
Independent Tasks are Distributed as Workers are Available

```
// initialize values 0-99
    a[]=i;
    b[]=i;

// Send each idle worker an index of a[] & b[] to add and return the sum
// and continue while there is still work to be done

    while (i<100)
     {
      // find an idle worker & send it value of i (index) to add
      // receive back summed values
     }
```

Pipeline Parallelism

Definition: each task works on one stage in a sequence of stages. The output of one stage is the input of the next. (Note: This works best when each stage takes the same amount of time to complete)

Examples:

- Assembly lines
- Computing partial sums

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
A	i	i+1	i+2	i+3	i+4	i+5	i+6			
↓ B		i	i+1	i+2	i+3	i+4	i+5	i+6		
↓ C			i	i+1	i+2	i+3	i+4	i+5	i+6	
				i	i+1	i+2	i+3	i+4	i+5	i+6

Pipeline Parallelism

```
// Process 0 initializes a[] & b[]
   for (i=0; i<=3; i++)
     {
      a[i] = i;
      b[i] = 0;
     }
   b[0] = a[0];
// Process 0 sends a & b to Process 1

// Process 1 receives a & b from Process 0
   b[1] = b[0] + a[1];
// Process 1 sends a & b to Process 2

// Process 2 receives a & b from Process 1
   b[2] = b[1] + a[2];
// Process 2 sends a & b to Process 3

// Process 3 receives a & b from Process 2
   b[3] = b[2] + a[3];
```

Tightly vs. Loosely Coupled Parallelism

Tightly Coupled Parallel Approaches

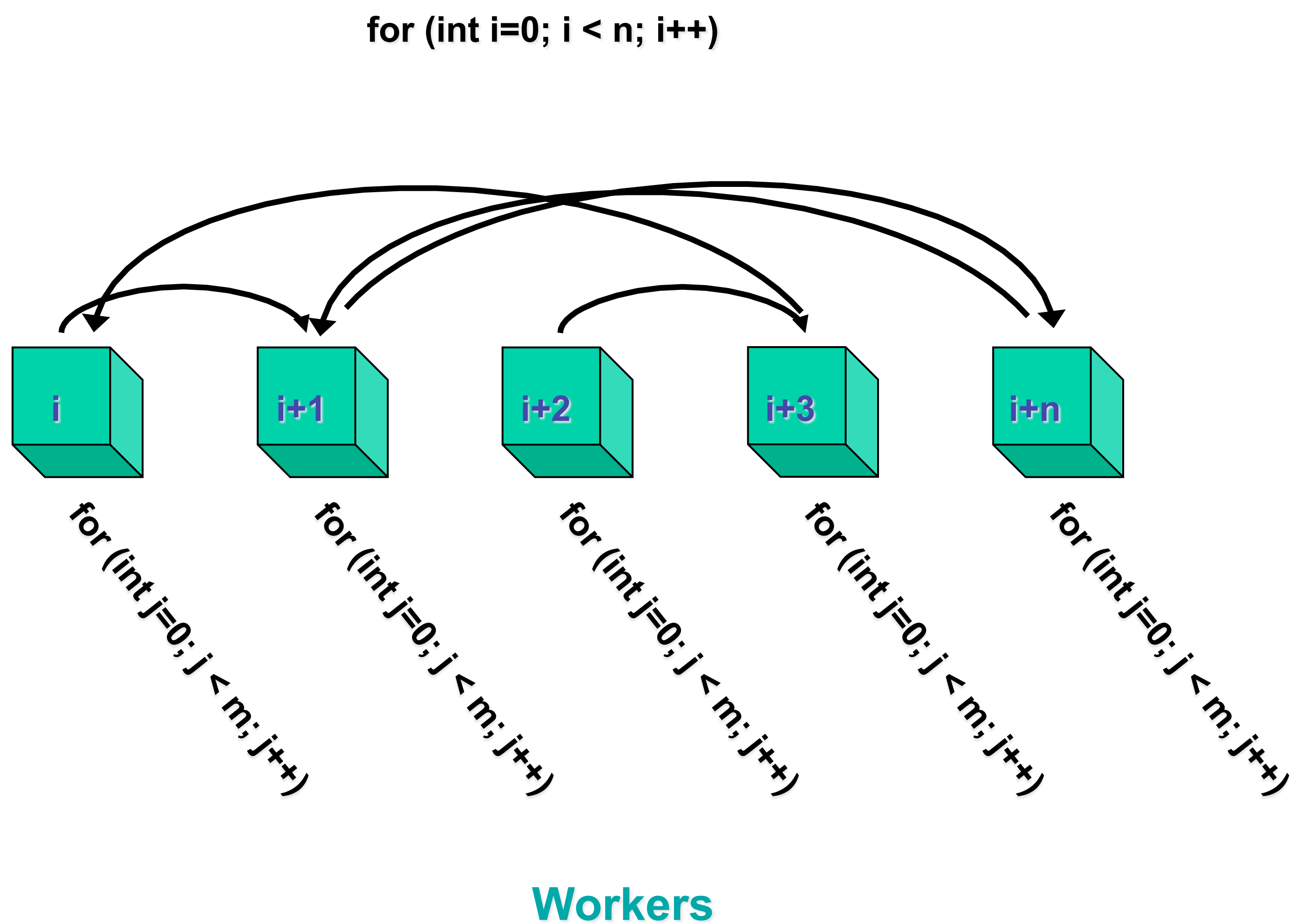
Parallel tasks must exchange data during the computation

Loosely Coupled Parallel Approaches

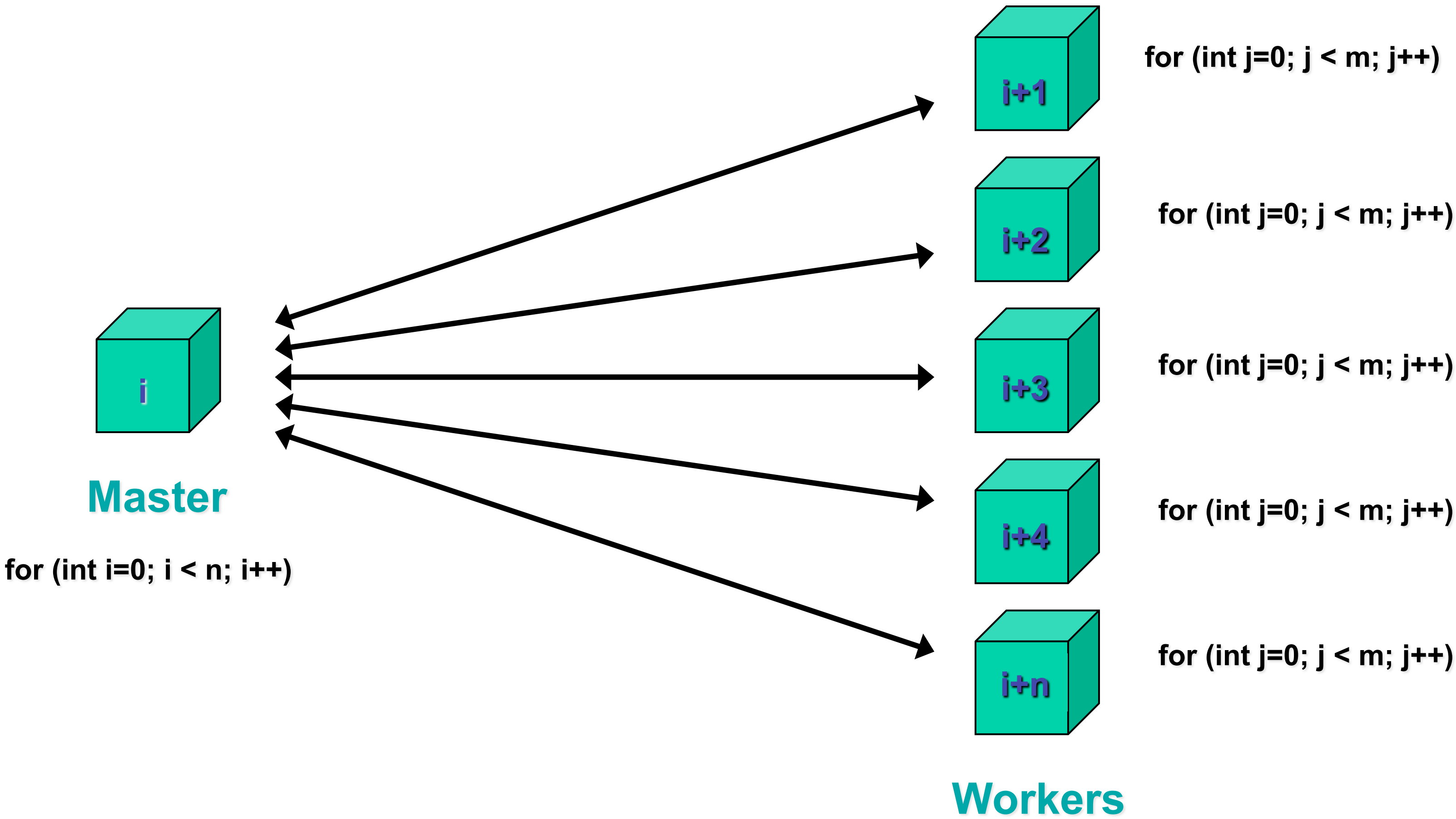
Parallel tasks can complete independent of each other

```
for (int i=0; i < n; i++)
 {
  for (int j=0; j < m; j++)
   {
    //Perform Calculation Here
   } // for j
 } // for i
```

Tightly Coupled Example: Strong Data Dependencies

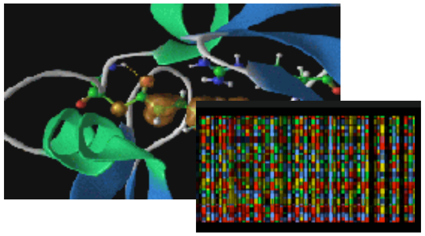


Loosely Coupled Example: Master-Worker Codes

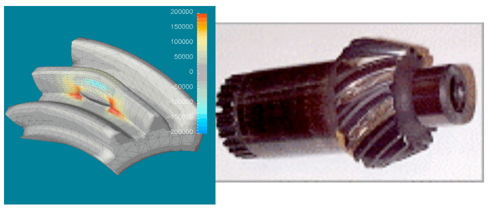


HPC Examples at Cornell: Parallel Computing and Data Intensive Computing

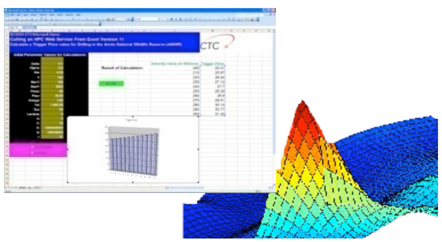
Parallel Computing Examples



Computational Biology Service Unit
http://cbsu.tc.cornell.edu/index.htm



Cornell Fracture Group
http://www.cfg.cornell.edu



Computational Finance
http://www.orie.cornell.edu/orie/manhattan/

Cornell Institute for Social and Economic Research
http://www.ciser.cornell.edu/

Data Intensive Computing Applications

Modern Research is Producing Massive Amounts of Data

- Microscopes
- Telescopes
- Gene Sequencers
- Mass Spectrometers
- Satellite & Radar Images
- Distributed Weather Sensors
- High Performance Computing (especially HPC Clusters)

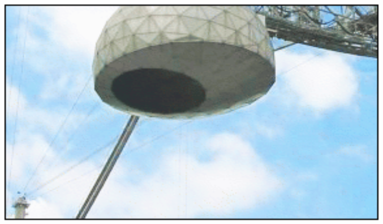
Research Communities Rely on Distributed Data Sources

- Collaboration
- Virtual Laboratories
- Laboratory Information Management Systems (LIMS)

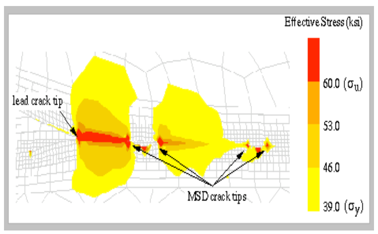
New Management and Usage Issues

- Security
- Reliability/Availability
- Manageability
- Data Locality – You can't ftp a petabyte to your laptop….

Data Intensive Computing Examples



Arecibo - World's Largest Radiotelescope
Johannes Gehrke, Jim Cordes, David Lifka
Serving Astronomy Data via SQL Server and Web Services
http://arecibo.tc.cornell.edu/PALFA
http://www.cs.cornell.edu/johannes/



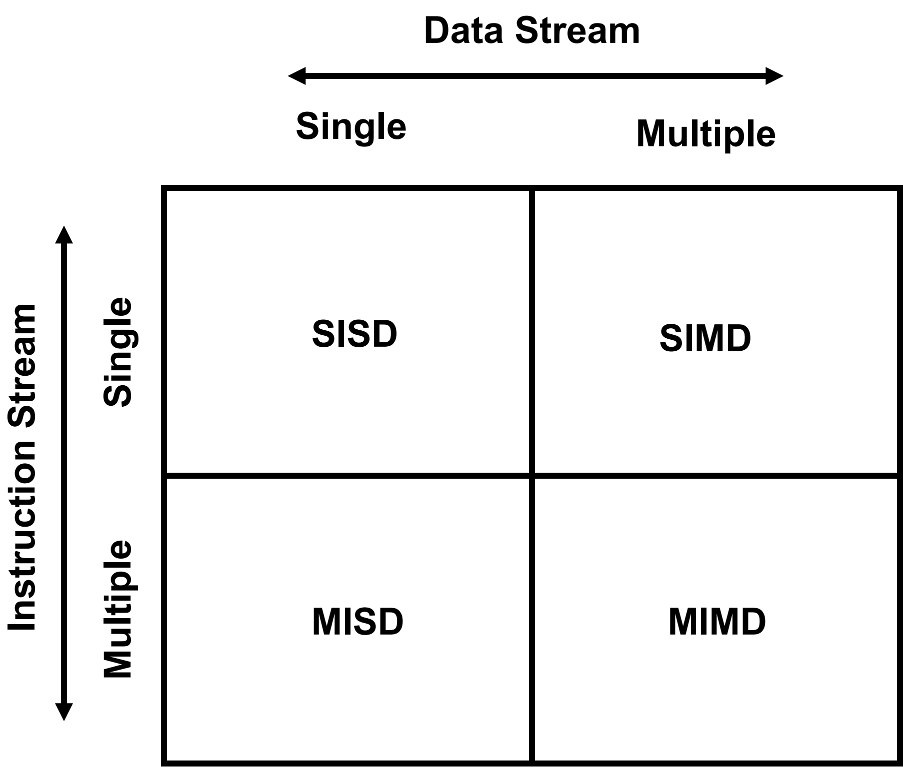
Cornell Fracture Group
Tony Ingraffea
Serving Finite Element Models via SQL Server & Web Services
http://www.cfg.cornell.edu/

Physically Accurate Imagery
Steve Marschner
http://www.cs.cornell.edu/~srm/

The Structure and Evolution of the Web
William Arms
http://www.cs.cornell.edu/wya/

High Performance Computing Architectures

Flynn's Taxonomy
Classification Scheme for Parallel Computers

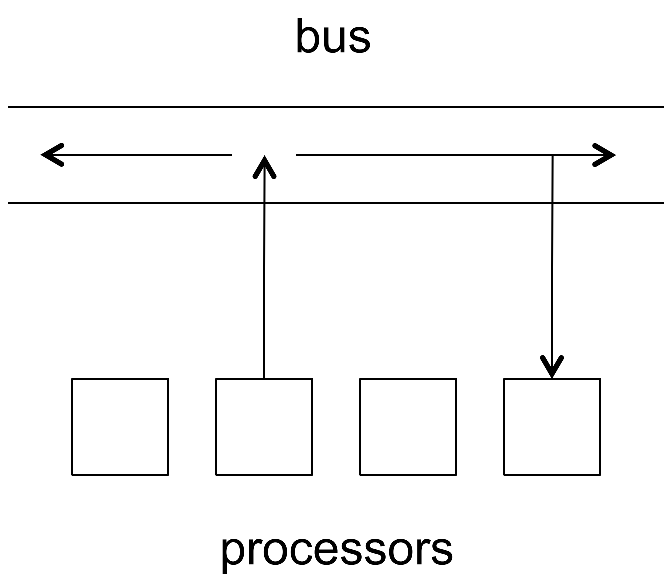


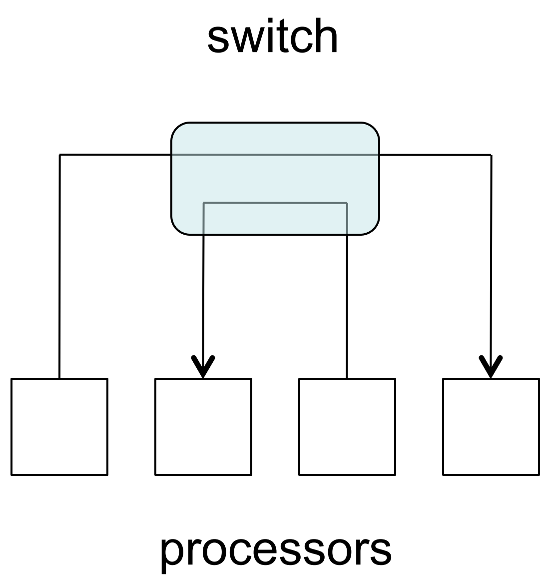
Examples from Flynn's Categories

- **SISD – Single Instruction Stream, Single Data Stream**
  - Common uniprocessor machines
- **SIMD – Single Instruction Stream, Multiple Data Streams**
  - Processor arrays (including GPUs) & pipelined vector processors
- **MISD – Multiple Instruction Streams, Single Data Stream**
  - Systolic arrays: think data pump or pumping-heart model (not many built)
- **MIMD – Multiple Instruction Streams, Multiple Data Streams**
  - Multiprocessors and multicomputers

- **Multiprocessor: multi-CPU computer with shared memory**
  - SMP: Symmetric MultiProcessor (uniform memory access)
  - NUMA: Non Uniform Memory Access multiprocessor
- **Multicomputer: team of computers with distributed CPUs and memory**
  - Must have external *interconnect* between "nodes"

Shared vs. Switched Media





SMP: not scalable,
bus becomes congested
as processors are added

NUMA, Multicomputer:
switch can grow with
number of processors

2D Mesh and Torus Topologies

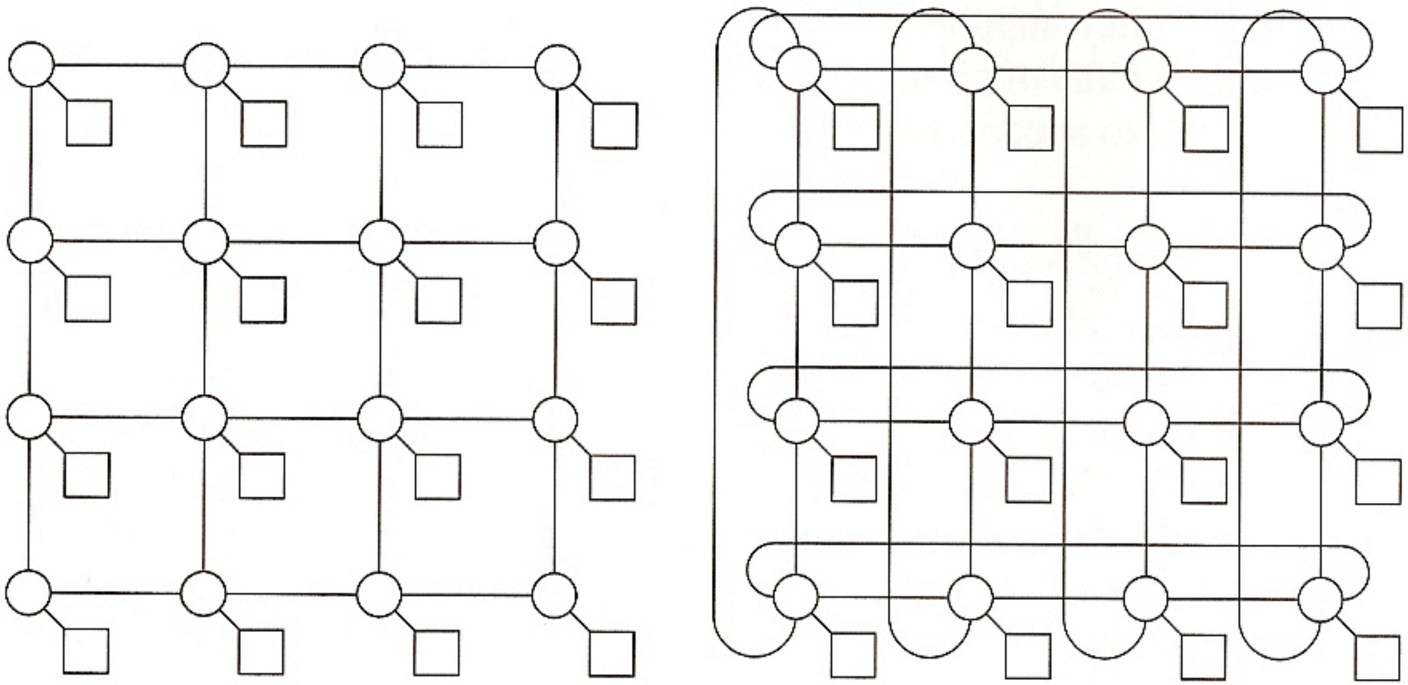


Figure 2.2 Variants of the 2-D mesh network. Circles represent switches, while squares represent processors. (a) Without wraparound connections. (b) With wraparound connections.

Hypercube Topology

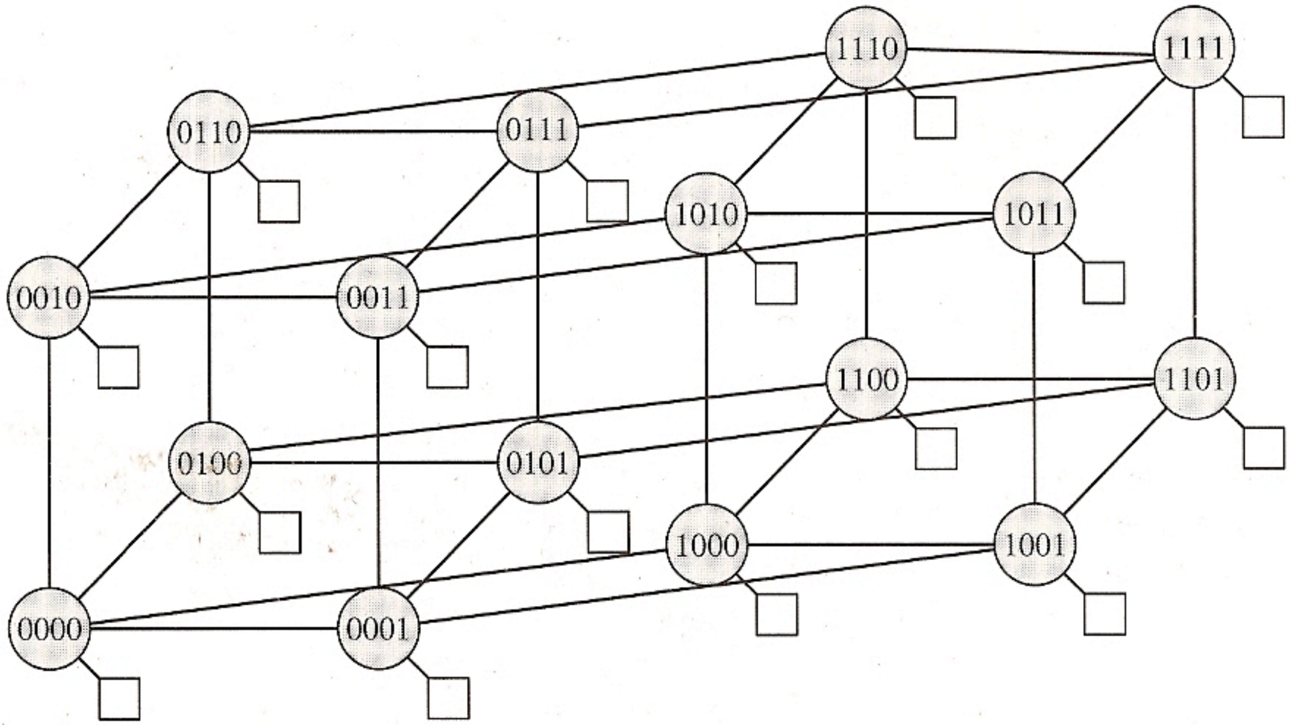


Figure 2.7 A hypercube network with 16 processor nodes and an equal number of switch nodes. Circles represent switches, and squares represent processors. Processor/switch pairs share addresses.

Tree, Fat Tree, and Hypertree Topologies

Fat tree if links get wider toward the top…

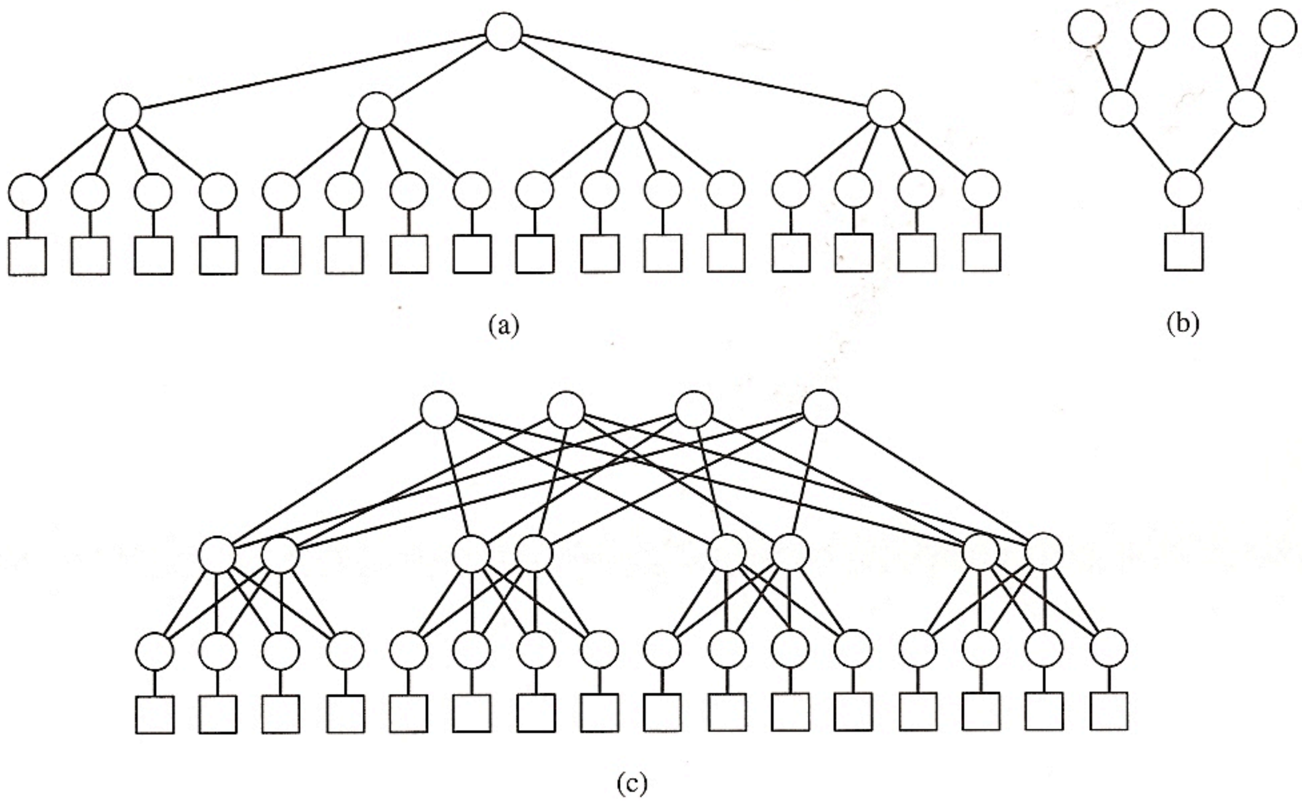
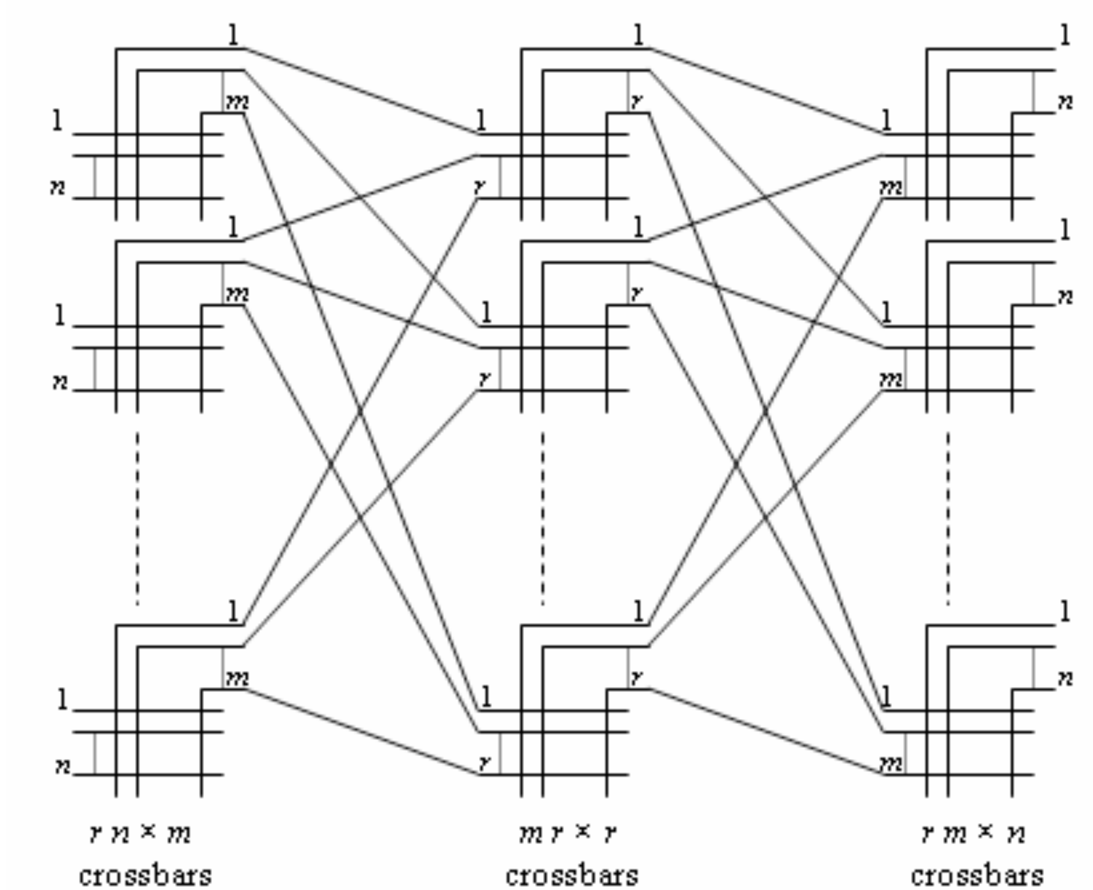


Figure 2.4 Hypertree network of degree 4 and depth 2. Circles represent switches, and squares represent processors. (a) Front view. (b) Side view. (c) Complete network.

Clos Network: Equal to a Full Crossbar Switch, Better Than a Hypertree (Fewer Hops)



Generally *n* = *m*, so inputs and outputs can be bundled into the same cable and plug into a single *switch port*

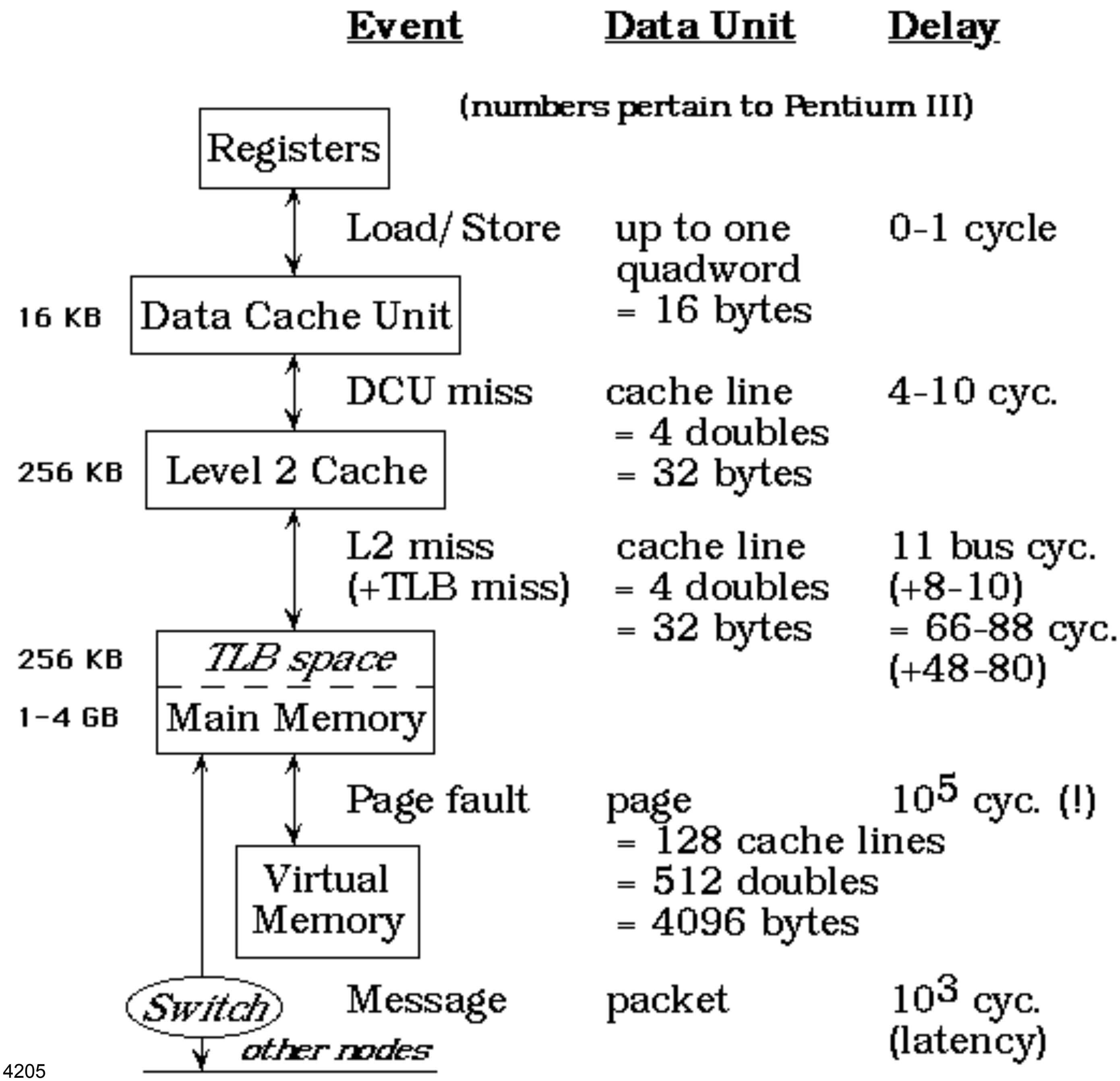
Comparison of Switched Media

Type	Latency	Bandwidth	Cost
Gigabit Ethernet	~1 msec	0.1 gigabyte/sec	$
10 Gigabit Ethernet	~100 μsec	1.0 gigabyte/sec	$$
QDR InfiniBand	~1μsec	3.6 gigabyte/sec	$$$

Mellanox 36-port InfiniBand switch

Computing Concepts

Single Processor Memory Hierarchy



Creating and Running Software

- **Compiler**
  - Produces object code: from myprog.c, creates myprog.o (Windows: .obj)
- **Linker**
  - Produces complete executable, including object code imported from libraries
- **Shared Objects (.so) and Dynamic Load Libraries (Windows DLLs)**
  - These are loaded at runtime: the link step inserts instructions on what to import
  - If a shared object is loaded, a single copy can be used by multiple processes
- **Process**
  - A running executable: the OS controls multitasking of processes via scheduling
- **Virtual Memory**
  - "Address space" available to a running process, addresses can be 32- or 64-bit
- **Paging (to Disk)**
  - Physical RAM has been exceeded: requested data are not in any cache (*cache miss*) or in RAM (*page fault*) must be loaded from swap space on a hard drive