CS 594 Spring 2006
Lecture 4:
Overview of High-Performance Computing

Jack Dongarra
Computer Science Department
University of Tennessee

Top 500 Computers
- Listing of the 500 most powerful Computers in the World
- Yardstick: Rmax from LINPACK MPP
  \[ Ax = b, \text{ dense problem} \]
Updated twice a year
  SC‘xy in the States in November
  Meeting in Germany in June
What is a Supercomputer?

- A supercomputer is a hardware and software system that provides close to the maximum performance that can currently be achieved.

- Over the last 12 years the range for the Top500 has increased greater than Moore’s Law

  - 1993:
    - #1 = 59.7 GFlop/s
    - #500 = 422 MFlop/s

  - 2005:
    - #1 = 280 TFlop/s
    - #500 = 1.64 TFlop/s

Why do we need them?
Almost all of the technical areas that are important to the well-being of humanity use supercomputing in fundamental and essential ways.

Computational fluid dynamics, protein folding, climate modeling, national security, in particular for cryptanalysis and for simulating nuclear weapons to name a few.

Architecture/Systems Continuum

- **Custom processor with custom interconnect**
  - Cray X1
  - NEC SX-8
  - IBM Regatta
  - IBM Blue Gene/L

- **Commodity processor with custom interconnect**
  - SGI Altix
  - Intel Itanium 2
  - Cray XT3, XD1
  - AMD Opteron

- **Commodity processor with commodity interconnect**
  - Clusters
    - Pentium, Itanium, Opteron, Alpha
  - GigE, Infiniband, Myrinet, Quadrics
  - NEC TX7
  - IBM eServer
  - Dawning

Tightly Coupled

Loosely Coupled

[Diagram showing the continuum from Custom to Commod with different processor and interconnect options]
Architectures / Systems

Cluster: Commodity processors & Commodity interconnect
Constellation: # of procs/node ≥ nodes in the system

Performance Development

Sum:
IBM ASCI White
LLNL
Fujitsu "NWT NAL"
Sandia
Intel ASCI Red
NEC Earth Simulator
IBM BlueGene/L
My Laptop

SUM
## 26th List: The TOP10

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Computer</th>
<th>Rmax [TF/s]</th>
<th>Installation Site</th>
<th>Country</th>
<th>Year</th>
<th>#Prc</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM</td>
<td>BlueGene/L eServer Blue Gene</td>
<td>280.6</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
<td>2005</td>
<td>131072</td>
</tr>
<tr>
<td>IBM</td>
<td>BGW eServer Blue Gene</td>
<td>91.29</td>
<td>IBM Thomas Watson</td>
<td>USA</td>
<td>2005</td>
<td>40960</td>
</tr>
<tr>
<td>IBM</td>
<td>ASC Purple Power5 g575</td>
<td>63.39</td>
<td>DOE/NNSA/LLNL</td>
<td>USA</td>
<td>2005</td>
<td>10240</td>
</tr>
<tr>
<td>SGI</td>
<td>Columbia Altix, Ithanium/Infiniband</td>
<td>51.87</td>
<td>NASA Ames</td>
<td>USA</td>
<td>2004</td>
<td>hybrid 10160</td>
</tr>
<tr>
<td>Dell</td>
<td>Thunderbird Pentium/Infiniband</td>
<td>38.27</td>
<td>Sandia</td>
<td>USA</td>
<td>2005</td>
<td>custom 8000</td>
</tr>
<tr>
<td>Cray</td>
<td>Red Storm Cray XT3 AMD</td>
<td>36.19</td>
<td>Sandia</td>
<td>USA</td>
<td>2005</td>
<td>10880</td>
</tr>
<tr>
<td>NEC</td>
<td>Earth-Simulator SX-6</td>
<td>35.86</td>
<td>Earth Simulator Center</td>
<td>Japan</td>
<td>2002</td>
<td>custom 5120</td>
</tr>
<tr>
<td>IBM</td>
<td>MareNostrum PFC 970/Myrtic</td>
<td>27.91</td>
<td>Barcelona Supercomputer Center</td>
<td>Spain</td>
<td>2005</td>
<td>custom 4800</td>
</tr>
<tr>
<td>IBM</td>
<td>eServer Blue Gene</td>
<td>27.45</td>
<td>ASTRON University Groningen</td>
<td>Netherlands</td>
<td>2005</td>
<td>custom 12288</td>
</tr>
<tr>
<td>Cray</td>
<td>Jaguar Cray XT3 AMD</td>
<td>20.53</td>
<td>Oak Ridge National Lab</td>
<td>USA</td>
<td>2005</td>
<td>hybrid 5200</td>
</tr>
</tbody>
</table>

### IBM BlueGene/L #1 131,072 Processors

Total of 18 systems all in the Top100

- 1.6 MWatts (1600 homes)
- 43,000 ops/s/person
- Rack (32 Node boards, 8x8x16)
- 2048 processors
- 131,072 procs

**Peak: 367 Tflop/s**
**Linpack: 281 Tflop/s**

The compute node ASICs include all networking and processor functionality. Each compute ASIC includes two 32-bit superscalar PowerPC 440 embedded cores (note that L1 cache coherence is not maintained between these cores). (13K sec about 3.6 hours; n=1.8M)
Processor Types

91% = 66% Intel
15% IBM
11% AMD
Concurrent Levels of the Top500

Flops per Gross Domestic Product
Based on the November 2005 Top500
KFlop/s per Capita (Flops/Pop)
Based on the November 2005 Top500

Hint: Peter Jackson had something to do with this

WETA Digital (Lord of the Rings)

Has nothing to do with the 47.2 million sheep in NZ

Fuel Efficiency: GFlops/Watt

Top 20 systems
Based on processor power rating only
Top500 Conclusions

- Microprocessor based supercomputers have brought a major change in accessibility and affordability.
- MPPs continue to account of more than half of all installed high-performance computers worldwide.

Distributed and Parallel Systems

- Gather (unused) resources
- Steal cycles
- System SW manages resources
- System SW adds value
- 10% - 20% overhead is OK
- Resources drive applications
- Time to completion is not critical
- Time-shared

- Bounded set of resources
- Apps grow to consume all cycles
- Application manages resources
- System SW gets in the way
- 5% overhead is maximum
- Apps drive purchase of equipment
- Real-time constraints
- Space-shared
Virtual Environments

Do they make any sense?
Performance Improvements for Scientific Computing Problems

Different Architectures

- **Parallel computing**: single systems with many processors working on same problem
- **Distributed computing**: many systems loosely coupled by a scheduler to work on related problems
- **Grid Computing**: many systems tightly coupled by software, perhaps geographically distributed, to work together on single problems or on related problems
Types of Parallel Computers

- The simplest and most useful way to classify modern parallel computers is by their memory model:
  - shared memory
  - distributed memory

Shared vs. Distributed Memory

- Shared memory - single address space. All processors have access to a pool of shared memory. (Ex: SGI Origin, Sun E10000)

- Distributed memory - each processor has its own local memory. Must do message passing to exchange data between processors. (Ex: CRAY T3E, IBM SP, clusters)
Shared Memory: UMA vs. NUMA

**Uniform memory access (UMA):** Each processor has uniform access to memory. Also known as symmetric multiprocessors (Sun E10000)

Non-uniform memory access (NUMA): Time for memory access depends on location of data. Local access is faster than non-local access. Easier to scale than SMPs (SGI Origin)

Distributed Memory: MPPs vs. Clusters

- **Processors-memory nodes are connected by some type of interconnect network**
  - Massively Parallel Processor (MPP): tightly integrated, single system image.
  - Cluster: individual computers connected by **s/w**
Both shared and distributed memory systems have:
1. **processors**: now generally commodity RISC processors
2. **memory**: now generally commodity DRAM
3. **network/interconnect**: between the processors and memory (bus, crossbar, fat tree, torus, hypercube, etc.)

We will now begin to describe these pieces in detail, starting with definitions of terms.

Interconnect-Related Terms

- **Latency**: How long does it take to start sending a "message"? Measured in microseconds.
  (Also in processors: How long does it take to output results of some operations, such as floating point add, divide etc., which are pipelined?)

- **Bandwidth**: What data rate can be sustained once the message is started? Measured in Mbytes/sec.
Interconnect-Related Terms

**Topology**: the manner in which the nodes are connected.
- Best choice would be a fully connected network (every processor to every other). Unfeasible for cost and scaling reasons.
- Instead, processors are arranged in some variation of a grid, torus, or hypercube.

![3-d hypercube](image) ![2-d mesh](image) ![2-d torus](image)

Highly Parallel Supercomputing: Where Are We?

- **Performance**:
  - Sustained performance has dramatically increased during the last year.
  - On most applications, sustained performance per dollar now exceeds that of conventional supercomputers. But...
  - Conventional systems are still faster on some applications.

- **Languages and compilers**:
  - Standardized, portable, high-level languages such as HPF, PVM and MPI are available. But...
  - Initial HPF releases are not very efficient.
  - Message passing programming is tedious and hard to debug.
  - Programming difficulty remains a major obstacle to usage by mainstream scientist.
Highly Parallel Supercomputing: Where Are We?

- Operating systems:
  - Robustness and reliability are improving.
  - New system management tools improve system utilization. But...
  - Reliability still not as good as conventional systems.

- I/O subsystems:
  - New RAID disks, HiPPI interfaces, etc. provide substantially improved I/O performance. But...
  - I/O remains a bottleneck on some systems.

The Importance of Standards - Software

- Writing programs for MPP is hard ...
- But ... one-off efforts if written in a standard language
- Past lack of parallel programming standards ...
  - ... has restricted uptake of technology (to "enthusiasts")
  - ... reduced portability (over a range of current architectures and between future generations)
- Now standards exist: (PVM, MPI & HPF), which ...
  - ... allows users & manufacturers to protect software investment
  - ... encourage growth of a "third party" parallel software industry & parallel versions of widely used codes
The Importance of Standards - Hardware

- **Processors**
  - commodity RISC processors
- **Interconnects**
  - high bandwidth, low latency communications protocol
  - no de-facto standard yet (ATM, Fibre Channel, HPPI, FDDI)
- **Growing demand for total solution:**
  - robust hardware + usable software
- **HPC systems containing all the programming tools / environments / languages / libraries / applications packages found on desktops**

The Future of HPC

- The expense of being different is being replaced by the economics of being the same
- HPC needs to lose its "special purpose" tag
- Still has to bring about the promise of scalable general purpose computing …
- … but it is dangerous to ignore this technology
- Final success when MPP technology is embedded in desktop computing
- Yesterday's HPC is today's mainframe is tomorrow's workstation
Achieving TeraFlops

- In 1991, 1 Gflop/s
- 1000 fold increase
  - Architecture
    - exploiting parallelism
  - Processor, communication, memory
    - Moore’s Law
  - Algorithm improvements
    - block-partitioned algorithms

Future: Petaflops (\(10^{15}\) fl pt ops/s)

Today \(\approx \sqrt{10^{15}}\) flops for our workstations

- A Pflop for 1 second \(\Rightarrow\) a typical workstation computing for 1 year.
- From an algorithmic standpoint
  - concurrency
  - data locality
  - latency & sync
  - floating point accuracy
  - dynamic redistribution of workload
  - new language and constructs
  - role of numerical libraries
  - algorithm adaptation to hardware failure
A Petaflops Computer System

- 1 Pflop/s sustained computing
- Between 10,000 and 1,000,000 processors
- Between 10 TB and 1PB main memory
- Commensurate I/O bandwidth, mass store, etc.
- If built today, cost $40 B and consume 1 TWatt.
- May be feasible and “affordable” by the year 2010

Question:

- Suppose we want to compute using four decimal arithmetic:
  - $S = 1.000 + 1.000 \times 10^4 - 1.000 \times 10^4$
  - What's the answer?
Defining Floating Point Arithmetic

**Representable numbers**
- Scientific notation: \( \pm \cdot d.d...d \times r^{\text{exp}} \)
- Sign bit: +/-
- Radix \( r \) (usually 2 or 10, sometimes 16)
- Significand \( d.d...d \) (how many base- \( r \) digits \( d \)?)
- Exponent \( \text{exp} \) (range?)
- Others?

**Operations:**
- Arithmetic: +, -, x, /, ...
  - How to round result to fit in format
- Comparison (\( <, =, > \))
- Conversion between different formats
  - Short to long FP numbers, FP to integer
- Exception handling
  - What to do for 0/0, \( 2^{\text{largest_number}} \), etc.
- Binary/Decimal conversion
  - For I/O, when radix not 10

IEEE Floating Point Arithmetic Standard 754 (1985) - Normalized Numbers

**Normalized Nonzero Representable Numbers:** \( \pm \cdot 1.d.d...d \times r^{\text{exp}} \)
- \( \text{Macheps} = \text{Machine epsilon} = 2^{-\#\text{significand bits}} = \text{relative error in each operation} \)
- Smallest non-zero number \( c \) such that \( (1 + c) > 1 \)
- \( \text{OV} = \text{overflow threshold} = \text{largest number} \)
- \( \text{UN} = \text{underflow threshold} = \text{smallest number} \)

<table>
<thead>
<tr>
<th>Format</th>
<th># bits</th>
<th>#significand bits</th>
<th>macheps</th>
<th>#exponent bits</th>
<th>exponent range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>32</td>
<td>23+1</td>
<td>(2^{-24} (-10^{-7}))</td>
<td>8</td>
<td>2^{-126} - 2^{127} (-10^{+38})</td>
</tr>
<tr>
<td>Double</td>
<td>64</td>
<td>52+1</td>
<td>(2^{-53} (-10^{-16}))</td>
<td>11</td>
<td>2^{-1022} - 2^{1023} (-10^{+308})</td>
</tr>
<tr>
<td>Double</td>
<td>&gt;=80</td>
<td>&gt;=64</td>
<td>(2^{-64} (-10^{-19}))</td>
<td>&gt;=15</td>
<td>2^{-16382} - 2^{16383} (-10^{+4932})</td>
</tr>
</tbody>
</table>

Extended (80 bits on all Intel machines)

**Zero:** \( \pm 0, \text{significand and exponent all zero} \)
- Why bother with -0 later
IEEE Floating Point Arithmetic Standard 754 - "Denoms"

- **Denormalized Numbers**: \( \pm 0.d\ldots d \times 2^{\text{min-exp}} \)
  - sign bit, nonzero significand, minimum exponent
  - Fills in gap between UN and 0
- **Underflow Exception**
  - occurs when exact nonzero result is less than underflow threshold UN
  - Ex: UN/3
  - return a denorm, or zero

---

IEEE Floating Point Arithmetic Standard 754 - \( \pm \infty \)

- **\( \pm \infty \)**: Sign bit, zero significand, maximum exponent
- **Overflow Exception**
  - occurs when exact finite result too large to represent accurately
  - Ex: 2*OV
  - return \( \pm \infty \)
- **Divide by zero Exception**
  - return \( \pm \infty \) = 1/\( \pm 0 \)
  - sign of zero important!
- **Also return \( \pm \infty \) for**
  - 3+\( \infty \), 2*\( \infty \), \( \infty \)*\( \infty \)
  - Result is exact, not an exception!
IEEE Floating Point Arithmetic
Standard 754 - NAN (Not A Number)

- **NAN**: Sign bit, nonzero significand, maximum exponent
- **Invalid Exception**
  - occurs when exact result not a well-defined real number
  - 0/0
  - sqrt(-1)
  - infinity/infinity, infinity/infinity, 0*infinity
  - NAN + 3
  - NAN > 3?
  - Return a NAN in all these cases
- **Two kinds of NANNs**
  - Quiet - propagates without raising an exception
  - Signaling - generate an exception when touched
    - good for detecting uninitialized data

Error Analysis

- **Basic error formula**
  - \( f(a \text{ op } b) = (a \text{ op } b)*(1 + d) \) where
    - \( d \) one of +,-,*/
    - \(|d| \leq \text{macheps}\)
    - assuming no overflow, underflow, or divide by zero
- **Example: adding 4 numbers**
  - \( f(x_1+x_2+x_3+x_4) = (((x_1+x_2)*(1+d_1) + x_3)*(1+d_2) + x_4)*(1+d_3) \)
    - \( = x_1*(1+d_1)^*(1+d_2)*(1+d_3) + x_2*(1+d_1)^*(1+d_2)*(1+d_3) + x_3*(1+d_2)^*(1+d_3) + x_4*(1+d_3) \)
    - \( = x_1*(1+e_1) + x_2*(1+e_2) + x_3*(1+e_3) + x_4*(1+e_4) \)
    - where each \(|e_i| \leq 3*\text{macheps}\)
  - get exact sum of slightly changed summands \(x_i^*(1+e_i)\)
  - Backward Error Analysis - algorithm called numerically stable if it gives the exact result for slightly changed inputs
  - Numerical Stability is an algorithm design goal
Backward error

- Approximate solution is exact solution to modified problem.
- How large a modification to original problem is required to give result actually obtained?
- How much data error in initial input would be required to explain all the error in computed results?
- Approximate solution is good if it is exact solution to "nearby" problem.

\[ \frac{|x' - x|}{|f(x') - f(x)|} \]  

Sensitivity and Conditioning

- Problem is insensitive or well conditioned if relative change in input causes commensurate relative change in solution.
- Problem is sensitive or ill-conditioned, if relative change in solution can be much larger than that in input data.

\[ \text{Cond} = \frac{|\text{Relative change in solution}|}{|\text{Relative change in input data}|} = \frac{|f(x') - f(x)|/f(x)|}{|(x' - x)/x|} \]

- Problem is sensitive, or ill-conditioned, if cond \( \gg 1 \).
- When function \( f \) is evaluated for approximate input \( x' = x + h \) instead of true input value of \( x \).
- Absolute error \( = f(x + h) - f(x) \approx h f'(x) \)
- Relative error \( \approx \frac{f(x + h) - f(x)}{f(x)} \approx h f'(x) / f(x) \)
Sensitivity: 2 Examples

\( \cos(\pi/2) \) and 2-d System of Equations

- Consider problem of computing cosine function for arguments near \( \pi/2 \).
- Let \( x \approx \pi/2 \) and let \( h \) be small perturbation to \( x \). Then

  \[
  \text{abs: } f(x + h) - f(x) \approx h f'(x)
  \]

  \[
  \text{rel: } \frac{f(x + h) - f(x)}{f(x)} \approx \frac{h f'(x)}{f(x)}
  \]

  \[
  \text{absolute error} = \cos(x+h) - \cos(x) \\
  \approx -h \sin(x) \approx -h,
  \]

  \[
  \text{relative error} \approx -h \tan(x) \approx \infty
  \]

- So small change in \( x \) near \( \pi/2 \) causes large relative change in \( \cos(x) \) regardless of method used.
- \( \cos(1.57079) = 0.63267949 \times 10^{-5} \)
- \( \cos(1.57078) = 1.64267949 \times 10^{-5} \)
- Relative change in output is a quarter million times greater than relative change in input.

\( a^*x_1 + b^*x_2 = f \)

\( c^*x_1 + d^*x_2 = g \)
Example: Polynomial Evaluation
Using Horner’s Rule

- Horner’s rule to evaluate $p = \sum_{k=0}^{n} c_k \cdot x^k$
  - $p = c_n$, for $k=n-1$ down to 0, $p = x \cdot p + c_k$
- Numerically Stable
- Apply to $(x-2)^9 = x^9 - 18x^8 + \ldots - 512$
  - $-2^9 + x^8(2^8 - x^7(2^7 + \ldots )))$
- Evaluated around 2

begin
  p := c[n];
  for $k := n$ to 0 by -1 do
    p := p \cdot x + c[k]
  end { for }
HonerPoly := p;
end { HonerPoly }

Example: polynomial evaluation (continued)

- $(x-2)^9 = x^9 - 18x^8 + \ldots - 512$
- We can compute error bounds using
  - $\text{fl}(a \text{ op } b) = (a \text{ op } b) \times (1+d)$
Exception Handling

What happens when the “exact value” is not a real number, or too small or too large to represent accurately?

5 Exceptions:

- **Overflow** - exact result > OV, too large to represent
- **Underflow** - exact result nonzero and < UN, too small to represent
- **Divide-by-zero** - nonzero/0
- **Invalid** - 0/0, sqrt(-1), ...
- **Inexact** - you made a rounding error (very common!)

Possible responses

- Stop with error message (unfriendly, not default)
- Keep computing (default, but how?)

Summary of Values
Representable in IEEE FP

**+- Zero**

**Normalized nonzero numbers**

**Denormalized numbers**

**+-Infinity**

**NANs**

- Signaling and quiet
- Many systems have only quiet
Assuming $x$ and $y$ are non-negative

\[
\begin{align*}
a &= \max(x, y), \quad b = \min(x, y) \\
z &= \begin{cases} 
    a\sqrt{1 + \left(b/a\right)^2}, & a > 0 \\
    0, & a = 0
\end{cases}
\end{align*}
\]

Hazards of Parallel and Heterogeneous Computing

- **What new bugs arise in parallel floating point programs?**
- **Ex 1: Nonrepeatability**
  - Makes debugging hard!
- **Ex 2: Different exception handling**
  - Can cause programs to hang
- **Ex 3: Different rounding (even on IEEE FP machines)**
  - Can cause hanging, or wrong results with no warning
- **See** [www.netlib.org/lapack/lawns/lawn112.ps](http://www.netlib.org/lapack/lawns/lawn112.ps)
- **IBM RS6K and Java**
Types of Parallel Computers

- The simplest and most useful way to classify modern parallel computers is by their memory model:
  - shared memory
  - distributed memory

Standard Uniprocessor Memory Hierarchy

- Intel Pentium 4 2 GHz processor
- P7 Prescott 478
  - 8 Kbytes of 4 way assoc. L1 instruction cache with 32 byte lines.
  - 8 Kbytes of 4 way assoc. L1 data cache with 32 byte lines.
  - 256 Kbytes of 8 way assoc. L2 cache 32 byte lines.
  - 400 MB/s bus speed
  - SSE2 provide peak of 4 Gflop/s
Shared Memory / Local Memory

- Usually think in terms of the hardware
- What about a software model?
- How about something that works like cache?
- Logically shared memory

Parallel Programming Models

- Control
  - how is parallelism created
  - what orderings exist between operations
  - how do different threads of control synchronize
- Naming
  - what data is private vs. shared
  - how logically shared data is accessed or communicated
- Set of operations
  - what are the basic operations
  - what operations are considered to be atomic
- Cost
  - how do we account for the cost of each of the above
Trivial Example \[ \sum_{i=0}^{n-1} f(A[i]) \]

- **Parallel Decomposition:**
  - Each evaluation and each partial sum is a task
- **Assign n/p numbers to each of p procs**
  - Each computes independent “private” results and partial sum
  - One (or all) collects the p partial sums and computes the global sum

=> **Classes of Data**
- **Logically Shared**
  - The original n numbers, the global sum
- **Logically Private**
  - The individual function evaluations
  - What about the individual partial sums?

### Programming Model 1

- **Shared Address Space**
  - Program consists of a collection of threads of control,
  - Each with a set of private variables
    - E.g., local variables on the stack
  - Collectively with a set of shared variables
    - E.g., static variables, shared common blocks, global heap
  - Threads communicate implicitly by writing and reading shared variables
  - Threads coordinate explicitly by synchronization operations on shared variables
    - Writing and reading flags
    - Locks, semaphores
- **Like concurrent programming on uniprocessor**
Model 1

- A shared memory machine
- Processors all connected to a large shared memory
- "Local" memory is not (usually) part of the hardware
  - Sun, DEC, Intel "SMPs" (Symmetric multprocessors) in Millennium; SGI Origin
- Cost: much cheaper to cache than main memory

Machine model 1a: A Shared Address Space Machine
- replace caches by local memories (in abstract machine model)
- this affects the cost model -- repeatedly accessed data should be copied
- Cray T3E

Shared Memory code for computing a sum

Thread 1

\[
\begin{align*}
[s &= 0 \text{ initially}] \\
local_s1 &= 0 \\
for i &= 0, n/2-1 \\
local_s1 &= local_s1 + f(A[i]) \\
s &= s + local_s1
\end{align*}
\]

Thread 2

\[
\begin{align*}
[s &= 0 \text{ initially}] \\
local_s2 &= 0 \\
for i &= n/2, n-1 \\
local_s2 &= local_s2 + f(A[i]) \\
s &= s + local_s2
\end{align*}
\]

What could go wrong?
Pitfall and solution via synchronization

- Pitfall in computing a global sum \( s = \text{local}_s1 + \text{local}_s2 \)

<table>
<thead>
<tr>
<th>Thread 1 (initially ( s=0 ))</th>
<th>Thread 2 (initially ( s=0 ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>load ( s ) [from mem to reg]</td>
<td>load ( s ) [from mem to reg; initially 0]</td>
</tr>
<tr>
<td>( s = s + \text{local}_s1 ) [( \text{local}_s1 ), in reg]</td>
<td>( s = s + \text{local}_s2 ) [( \text{local}_s2 ), in reg]</td>
</tr>
<tr>
<td>store ( s ) [from reg to mem]</td>
<td>store ( s ) [from reg to mem]</td>
</tr>
</tbody>
</table>

- Instructions from different threads can be interleaved arbitrarily
- What can final result \( s \) stored in memory be?
- Race Condition
- Possible solution: Mutual Exclusion with Locks

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>lock</td>
<td>lock</td>
</tr>
<tr>
<td>load ( s )</td>
<td>load ( s )</td>
</tr>
<tr>
<td>( s = s + \text{local}_s1 )</td>
<td>( s = s + \text{local}_s2 )</td>
</tr>
<tr>
<td>store ( s )</td>
<td>store ( s )</td>
</tr>
<tr>
<td>unlock</td>
<td>unlock</td>
</tr>
</tbody>
</table>

- Locks must be atomic (execute completely without interruption)

Programming Model 2

- Message Passing
  - program consists of a collection of named processes
    - thread of control plus local address space
    - local variables, static variables, common blocks, heap
  - processes communicate by explicit data transfers
    - matching pair of send & receive by source and dest. proc.
  - coordination is implicit in every communication event
  - logically shared data is partitioned over local processes
- Like distributed programming
- Program with standard libraries: MPI, PVM
Model 2

- A distributed memory machine
  - Cray T3E, IBM SP2, Clusters
- Processors all connected to own memory (and caches)
  - cannot directly access another processor's memory
- Each "node" has a network interface (NI)
  - all communication and synchronization done through this

Computing $s = x(1) + x(2)$ on each processor

° First possible solution

Processor 1
send xlocal, proc2
[xlocal = x(1)]
receive xremote, proc2
$s = xlocal + xremote$

Processor 2
receive xremote, proc1
send xlocal, proc1
[xlocal = x(2)]
$s = xlocal + xremote$

° Second possible solution - what could go wrong?

Processor 1
send xlocal, proc2
[xlocal = x(1)]
receive xremote, proc2
$s = xlocal + xremote$

Processor 2
send xlocal, proc1
[xlocal = x(2)]
receive xremote, proc1
$s = xlocal + xremote$

° What if send/receive act like the telephone system? The post office?
Programming Model 3

- **Data Parallel**
  - Single sequential thread of control consisting of parallel operations
  - Parallel operations applied to all (or defined subset) of a data structure
  - Communication is implicit in parallel operators and “shifted” data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this model

- Like marching in a regiment

<table>
<thead>
<tr>
<th>A</th>
<th>fA</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{A: array of all data}]</td>
<td>[\text{fA: } f(A)]</td>
<td>[\text{s: } \text{sum}(fA)]</td>
</tr>
</tbody>
</table>

° Think of Matlab

Model 3

- **Vector Computing**
  - One instruction executed across all the data in a pipelined fashion
  - Parallel operations applied to all (or defined subset) of a data structure
  - Communication is implicit in parallel operators and “shifted” data structures
  - Elegant and easy to understand and reason about
  - Not all problems fit this model

- Like marching in a regiment

<table>
<thead>
<tr>
<th>A</th>
<th>fA</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>[\text{A: array of all data}]</td>
<td>[\text{fA: } f(A)]</td>
<td>[\text{s: } \text{sum}(fA)]</td>
</tr>
</tbody>
</table>

° Think of Matlab
Model 3

- An SIMD (Single Instruction Multiple Data) machine
- A large number of small processors
- A single "control processor" issues each instruction
  - each processor executes the same instruction
  - some processors may be turned off on any instruction

Machines not popular (CM2), but programming model is
- implemented by mapping n-fold parallelism to p processors
- mostly done in the compilers (HPF = High Performance Fortran).

Model 4

- Since small shared memory machines (SMPs) are the fastest commodity machine, why not build a larger machine by connecting many of them with a network?
- CLUMP = Cluster of SMPs
- Shared memory within one SMP, message passing outside
- Clusters, ASCI Red (Intel), ...
- Programming model?
  - Treat machine as "flat", always use message passing, even within SMP (simple, but ignore important part of memory hierarchy)
  - Expose two layers: shared memory (OpenMP) and message passing (MPI) higher performance, but ugly to program.
Programming Model 5

- Bulk Synchronous Processing (BSP) - L. Valiant
- Used within the message passing or shared memory models as a programming convention
- Phases separated by global barriers
  - Compute phases: all operate on local data (in distributed memory)
    - or read access to global data (in shared memory)
  - Communication phases: all participate in rearrangement or reduction of global data
- Generally all doing the "same thing" in a phase
  - all do f, but may all do different things within f
- Simplicity of data parallelism without restrictions

Summary so far

- Historically, each parallel machine was unique, along with its programming model and programming language
- You had to throw away your software and start over with each new kind of machine - ugh
- Now we distinguish the programming model from the underlying machine, so we can write portably correct code, that runs on many machines
  - MPI now the most portable option, but can be tedious
- Writing portably fast code requires tuning for the architecture
  - Algorithm design challenge is to make this process easy
  - Example: picking a blocksize, not rewriting whole algorithm