Parallel Machines and Programming Models
Outline

• Overview of parallel machines (~hardware) and programming models (~software)
  • Shared memory
  • Shared address space
  • Message passing
  • Data parallel
  • Clusters of SMPs or GPUs
  • Grid

• Note: Parallel machine may or may not be tightly coupled to programming model
  • Historically, tight coupling
  • Today, portability is important
A generic 

Interconnection Network

• Where is the memory physically located?
• Is it connect directly to processors?
• What is the connectivity of the network?
Parallel Programming Models

- **Programming model** is made up of the languages and libraries that create an abstract view of the machine

- **Control**
  - How is parallelism created?
  - What orderings exist between operations?

- **Data**
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?

- **Synchronization**
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?

- **Cost**
  - How do we account for the cost of each of the above?
Simple Example

• Consider applying a function $f$ to the elements of an array $A$ and then computing its sum:

$$
\sum_{i=0}^{n-1} f(A[i])
$$

• Questions:
  • Where does $A$ live? All in single memory? Partitioned?
  • What work will be done by each processors?
  • They need to coordinate to get a single result, how?

A = array of all data
fA = f(A)
s = sum(fA)
Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables

![Diagram of shared memory model]

- y = ..s ...
- s = ...
- i: 2
- i: 5
- i: 8
- P0
- P1
- Pn
Simple Example

- **Shared memory strategy:**
  - small number $p \ll n=\text{size}(A)$ processors
  - attached to single memory

- **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.
    - Collect the $p$ partial sums and compute a global sum.

**Two Classes of Data:**

- **Logically Shared**
  - The original $n$ numbers, the global sum.

- **Logically Private**
  - The individual function evaluations.
  - What about the individual partial sums?

\[
\sum_{i=0}^{n-1} f(A[i])
\]
Thread 1

\[
\text{for } i = 0, \ n/2-1 \\
\quad s = s + f(A[i])
\]

Thread 2

\[
\text{for } i = n/2, \ n-1 \\
\quad s = s + f(A[i])
\]

What is the problem with this program?

A race condition or data race occurs when:
- Two processors (or two threads) access the same variable, and at least one does a write.
- The accesses are concurrent (not synchronized) so they could happen simultaneously.
### Shared Memory “Code” for Computing a Sum

#### A=

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

f(x) = x^2

```java
static int s = 0;
```

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>compute f([A[i]]) and put in reg0</td>
<td>compute f([A[i]]) and put in reg0</td>
</tr>
<tr>
<td>reg1 = s</td>
<td>reg1 = s</td>
</tr>
<tr>
<td>reg1 = reg1 + reg0</td>
<td>reg1 = reg1 + reg0</td>
</tr>
<tr>
<td>s = reg1</td>
<td>s = reg1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Assume A = [3,5], f(x) = x^2, and s=0 initially
- For this program to work, s should be $3^2 + 5^2 = 34$ at the end
  - but it may be 34, 9, or 25
- The *atomic* operations are reads and writes
  - Never see $\frac{1}{2}$ of one number, but += operation is *not* atomic
  - All computations happen in (private) registers
Improved Code for Computing a Sum

static int s = 0;
static lock lk;

Thread 1

local_s1 = 0
for i = 0, n/2-1
  local_s1 = local_s1 + f(A[i])
lock(lk);
s = s + local_s1
unlock(lk);

Thread 2

local_s2 = 0
for i = n/2, n-1
  local_s2 = local_s2 + f(A[i])
lock(lk);
s = s + local_s2
unlock(lk);

• Since addition is associative, it’s OK to rearrange order
• Most computation is on private variables
  - Sharing frequency is also reduced, which might improve speed
  - But there is still a race condition on the update of shared s
  - The race condition can be fixed by adding locks (only one thread can hold a lock at a time; others wait for it)
Machine Model 1a: Shared Memory

- Processors all connected to a large shared memory.
  - Typically called Symmetric Multiprocessors (SMPs)
  - SGI, Sun, HP, Intel, IBM SMPs (nodes of Millennium, SP)
  - Multicore chips, except that all caches are shared
- Difficulty scaling to large numbers of processors
  - <= 32 processors typical
- Advantage: uniform memory access (UMA)
- Cost: much cheaper to access data in cache than main memory.

![Diagram showing shared memory architecture with processors P1, P2, Pn connected to a shared memory through a bus and individual caches.]
Machine Model 1b: Multithreaded Processor

- Multiple thread “contexts” without full processors
- Memory and some other state is shared
- Sun Niagra processor (for servers)
  - Up to 64 threads all running simultaneously (8 threads x 8 cores)
  - In addition to sharing memory, they share floating point units
  - Why? Switch between threads for long-latency memory operations

- Cray MTA and Eldorado processors (for HPC)
Eldorado Processor (logical view)

Source: John Feo, Cray
Machine Model 1c: Distributed Shared Memory

- Memory is logically shared, but physically distributed
  - Any processor can access any address in memory
  - Cache lines (or pages) are passed around machine
- SGI is canonical example (+ research machines)
  - Scales to 512 (SGI Altix (Columbia) at NASA/Ames)
  - Limitation is cache coherency protocols – how to keep cached copies of the same address consistent

![Diagram of distributed shared memory model]

Cache lines (pages) must be large to amortize overhead → locality still critical to performance
# Review so far and plan for Lecture 3

## Programming Models

| 1. Shared Memory              | 1a. Shared Memory            |
| 3. Data Parallel              | 1c. Distributed Shared Mem.  |
| 4. Hybrid                     | 2a. Distributed Memory       |
|                               | 2b. Internet & Grid Computing|
|                               | 2c. Global Address Space     |
|                               | 3a. SIMD                     |
|                               | 3b. Vector                   |
|                               | 4. Hybrid                    |
|                               |                               |

## Machine Models

1. Shared Memory
2. Distributed Memory
3. Global Address Space
# Review so far and plan for Lecture 3

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>Machine Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Shared Memory</td>
<td>1a. Shared Memory</td>
</tr>
<tr>
<td>1b. Multithreaded Procs.</td>
<td>1b. Multithreaded Procs.</td>
</tr>
<tr>
<td>1c. Distributed Shared Mem.</td>
<td>1c. Distributed Shared Mem.</td>
</tr>
<tr>
<td>2. Message Passing</td>
<td>2a. Distributed Memory</td>
</tr>
<tr>
<td>2a. Global Address Space</td>
<td>2b. Internet &amp; Grid Computing</td>
</tr>
<tr>
<td>2c. Global Address Space</td>
<td>2c. Global Address Space</td>
</tr>
<tr>
<td>3. Data Parallel</td>
<td>3a. SIMD</td>
</tr>
<tr>
<td>3b. Vector</td>
<td>4. Hybrid</td>
</tr>
<tr>
<td>4. Hybrid</td>
<td>4. Hybrid</td>
</tr>
</tbody>
</table>
Programming Model 2: Message Passing

- Program consists of a collection of named processes.
  - Usually fixed at program startup time
  - Thread of control plus local address space -- NO shared data.
  - Logically shared data is partitioned over local processes.

- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used SW
Machine Model 2a: Distributed Memory

- Cray XT4, XT 5
- PC Clusters (Berkeley NOW, Beowulf)
- Hopper, Franklin, IBM SP-3, Millennium, CITRIS are distributed memory machines, but the nodes are SMPs.
- Each processor has its own memory and cache but cannot directly access another processor’s memory.
- Each “node” has a Network Interface (NI) for all communication and synchronization.

Diagram:
- Processor nodes (P0, P1, Pn) with memory and Network Interface (NI)
- Interconnect connecting the nodes
Machine Model 2b: Internet/Grid Computing

- **SETI@Home**: Running on 500,000 PCs
  - ~1000 CPU Years per Day
  - 485,821 CPU Years so far
- Sophisticated Data & Signal Processing Analysis
- Distributes Datasets from Arecibo Radio Telescope

Next Step—Allen Telescope Array

Google “volunteer computing” or “BOINC”
Programming Model 2a: Global Address Space

- Program consists of a collection of **named** threads.
  - Usually fixed at program startup time
  - Local and shared data, as in shared memory model
  - But, shared data is partitioned over local processes
  - Cost models say remote data is expensive

- Examples: UPC, Titanium, Co-Array Fortran

- Global Address Space programming is an intermediate point between message passing and shared memory

```
s[i] = ...
y = ..s[i] ...
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>s[0]: 26</th>
<th>s[1]: 32</th>
<th>s[n]: 27</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>P0</td>
<td>P1</td>
<td>Pn</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Machine Model 2c: Global Address Space

- Cray T3D, T3E, X1, and HP Alphaserver cluster
- Clusters built with Quadrics, Myrinet, or Infiniband
- The network interface supports RDMA (Remote Direct Memory Access)
  - NI can directly access memory without interrupting the CPU
  - One processor can read/write memory with one-sided operations (put/get)
  - Not just a load/store as on a shared memory machine
    - Continue computing while waiting for memory op to finish
  - Remote data is typically not cached locally

Global address space may be supported in varying degrees
## Review so far and plan for Lecture 3

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>Machine Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Shared Memory</td>
<td>1a. Shared Memory</td>
</tr>
<tr>
<td>1c. Distributed Shared Mem.</td>
<td>1c. Distributed Shared Mem.</td>
</tr>
<tr>
<td>2. Message Passing</td>
<td>2a. Distributed Memory</td>
</tr>
<tr>
<td>2a. Global Address Space</td>
<td>2b. Internet &amp; Grid Computing</td>
</tr>
<tr>
<td>2c. Global Address Space</td>
<td></td>
</tr>
<tr>
<td>3. Data Parallel</td>
<td>3a. SIMD</td>
</tr>
<tr>
<td>3b. Vector</td>
<td></td>
</tr>
<tr>
<td>4. Hybrid</td>
<td>4. Hybrid</td>
</tr>
</tbody>
</table>
Programming Model 3: Data Parallel

• Single thread of control consisting of parallel operations.
  • A = B+C could mean add two arrays in parallel

• Parallel operations applied to all (or a defined subset) of a data structure, usually an array
  • Communication is implicit in parallel operators
  • Elegant and easy to understand and reason about
  • Coordination is implicit – statements executed synchronously
  • Similar to Matlab language for array operations

• Drawbacks:
  • Not all problems fit this model
  • Difficult to map onto coarse-grained machines

A = array of all data
fA = f(A)
s = sum(fA)
Machine Model 3a: SIMD System

- A large number of (usually) small processors.
  - A single “control processor” issues each instruction.
  - Each processor executes the same instruction.
  - Some processors may be turned off on some instructions.

- Originally machines were specialized to scientific computing, few made (CM2, Maspar)

- Programming model can be implemented in the compiler
  - mapping n-fold parallelism to p processors, n >> p, but it’s hard (e.g., HPF)
Machine Model 3b: Vector Machines

- Vector architectures are based on a single processor
  - Multiple functional units
  - All performing the same operation
  - Instructions may specify large amounts of parallelism (e.g., 64-way) but hardware executes only a subset in parallel

- Historically important
  - Overtaken by MPPs in the 90s

- Re-emerging in recent years
  - At a large scale in the Earth Simulator (NEC SX6) and Cray X1
  - At a small scale in SIMD media extensions to microprocessors
    - SSE, SSE2 (Intel: Pentium/IA64)
    - Altivec (IBM/Motorola/Apple: PowerPC)
    - VIS (Sun: Sparc)
  - At a larger scale in GPUs

- Key idea: Compiler does some of the difficult work of finding parallelism, so the hardware doesn’t have to
Vector Processors

- Vector instructions operate on a vector of elements
  - These are specified as operations on vector registers

- A supercomputer vector register holds ~32-64 elts
  - The number of elements is larger than the amount of parallel hardware, called vector pipes or lanes, say 2-4

- The hardware performs a full vector operation in
  - \#elements-per-vector-register / \#pipes

(Actually, performs \#pipes adds in parallel)
Cray X1: Parallel Vector Architecture

Cray combines several technologies in the X1
• 12.8 Gflop/s Vector processors (MSP)
• Shared caches (unusual on earlier vector machines)
• 4 processor nodes sharing up to 64 GB of memory
• Single System Image to 4096 Processors
• Remote put/get between nodes (faster than MPI)
Earth Simulator Architecture

Parallel Vector Architecture

• High speed (vector) processors
• High memory bandwidth (vector architecture)
• Fast network (new crossbar switch)

Rearranging commodity parts can’t match this performance
## Review so far and plan for Lecture 3

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>Machine Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Shared Memory</td>
<td>1a. Shared Memory</td>
</tr>
<tr>
<td>1a. Shared Memory</td>
<td>1b. Multithreaded Procs.</td>
</tr>
<tr>
<td>1b. Multithreaded Procs.</td>
<td>1c. Distributed Shared Mem.</td>
</tr>
<tr>
<td>1c. Distributed Shared Mem.</td>
<td></td>
</tr>
<tr>
<td>2. Message Passing</td>
<td>2a. Distributed Memory</td>
</tr>
<tr>
<td>2a. Global Address Space</td>
<td>2b. Internet &amp; Grid Computing</td>
</tr>
<tr>
<td>2b. Internet &amp; Grid Computing</td>
<td>2c. Global Address Space</td>
</tr>
<tr>
<td>3. Data Parallel</td>
<td>3a. SIMD &amp; GPU</td>
</tr>
<tr>
<td>3a. SIMD &amp; GPU</td>
<td>3b. Vector</td>
</tr>
<tr>
<td>3b. Vector</td>
<td>4. Hybrid</td>
</tr>
<tr>
<td>4. Hybrid</td>
<td>4. Hybrid</td>
</tr>
</tbody>
</table>
Machine Model 4: Hybrid machines

- Multicore/SMPs are a building block for a larger machine with a network
- Common names:
  - CLUMP = Cluster of SMPs
- Many modern machines look like this:
  - Millennium, IBM SPs, NERSC Franklin, Hopper
- What is an appropriate programming model #4 ???
  - Treat machine as “flat”, always use message passing, even within SMP (simple, but ignores an important part of memory hierarchy).
  - Shared memory within one SMP, but message passing outside of an SMP.
- Graphics or game processors may also be building block
Programming Model 4: Hybrids

- Programming models can be mixed
  - Message passing (MPI) at the top level with shared memory within a node is common
  - New DARPA HPCS languages mix data parallel and threads in a global address space
  - Global address space models can (often) call message passing libraries or vice versa
  - Global address space models can be used in a hybrid mode
    - Shared memory when it exists in hardware
    - Communication (done by the runtime system) otherwise

- For better or worse
  - Supercomputers often programmed this way for peak performance
### Review so far and plan for Lecture 3

<table>
<thead>
<tr>
<th>Programming Models</th>
<th>Machine Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. <strong>Shared Memory</strong></td>
<td>1a. <strong>Shared Memory</strong></td>
</tr>
<tr>
<td>1a. <strong>Shared Memory</strong></td>
<td>1b. Multithreaded Procs.</td>
</tr>
<tr>
<td>1b. Multithreaded Procs.</td>
<td>1c. Distributed Shared Mem.</td>
</tr>
<tr>
<td>2. <strong>Message Passing</strong></td>
<td>2a. Distributed Memory</td>
</tr>
<tr>
<td>2a. <strong>Global Address Space</strong></td>
<td>2b. Internet &amp; Grid Computing</td>
</tr>
<tr>
<td>3. <strong>Data Parallel</strong></td>
<td>2c. Global Address Space</td>
</tr>
<tr>
<td>3a. SIMD &amp; GPU</td>
<td>3b. Vector</td>
</tr>
<tr>
<td>4. <strong>Hybrid</strong></td>
<td>4. <strong>Hybrid</strong></td>
</tr>
</tbody>
</table>

**What about GPU? What about Cloud?**
What about GPU and Cloud?

• GPU’s big performance opportunity is data parallelism
  • Most programs have a mixture of highly parallel operations, and some not so parallel
  • GPUs provide a threaded programming model (CUDA) for data parallelism to accommodate both
  • Current research attempting to generalize programming model to other architectures, for portability (OpenCL)
  • Lecture later in the semester

• Cloud computing lets large numbers of people easily share $O(10^5)$ machines
  • MapReduce was first programming model: data parallel on distributed memory
  • More flexible models (Hadoop…) invented since then