HW 1 updated and reposted
Updated all lectures, reposted
check that you are getting emails from class mailing list: par-comp-f14@mailman.sdsu.edu
if not, send me an email.
Interconnection networks

- Affects performance of both distributed and shared memory systems.

- Two categories:
  - Shared memory interconnects
  - Distributed memory interconnects
Shared memory interconnects

- Bus interconnect
  - A collection of parallel communication wires together with some hardware that controls access to the bus.
  - Communication wires are shared by the devices that are connected to it.
  - As the number of devices connected to the bus increases, contention for use of the bus increases, and performance decreases.
Shared memory interconnects

- Switched interconnect
  - Uses switches to control the routing of data among the connected devices.

- Crossbar –
  - Allows simultaneous communication among different devices.
  - Faster than buses.
  - But the cost of the switches and links is relatively high.
Figure 2.7

(a)
A crossbar switch connecting 4 processors (P1) and 4 memory modules (M1)

(b)
Configuration of internal switches in a crossbar

(c) Simultaneous memory accesses by the processors
Distributed memory interconnects

- Two groups
  - Direct interconnect
    - Each switch is directly connected to a processor memory pair, and the switches are connected to each other.
  - Indirect interconnect
    - Switches may not be directly connected to a processor.
Direct interconnect

Figure 2.8

(a) ring

(b) toroidal mesh
Bisection width

- A measure of “number of simultaneous communications” or “connectivity”.

- How many simultaneous communications can take place “across the divide” between the halves?
Two bisections of a ring

(a)

(b)

Figure 2.9
A bisection of a toroidal mesh

Figure 2.10
Definitions

- **Bandwidth**
  - The rate at which a link can transmit data.
  - Usually given in megabits or megabytes per second.

- **Bisection bandwidth**
  - A measure of network quality.
  - Instead of counting the number of links joining the halves, it sums the bandwidth of the links.
Fully connected network

- Each switch is directly connected to every other switch.

![Diagram showing impractical fully connected network]

bisection width = $p^2/4$

Figure 2.11
Hypercube

- Highly connected direct interconnect.
- Built inductively:
  - A one-dimensional hypercube is a fully-connected system with two processors.
  - A two-dimensional hypercube is built from two one-dimensional hypercubes by joining "corresponding" switches.
  - Similarly a three-dimensional hypercube is built from two two-dimensional hypercubes.
Hypercubes

Figure 2.12

(a) one-dimensional

(b) two-dimensional

(c) three-dimensional
Indirect interconnects

- Simple examples of indirect networks:
  - Crossbar
  - Omega network

- Often shown with unidirectional links and a collection of processors, each of which has an outgoing and an incoming link, and a switching network.
A generic indirect network

Figure 2.13
Crossbar interconnect for distributed memory

Figure 2.14
An omega network

Figure 2.15
A switch in an omega network

Figure 2.16
More definitions

- Any time data is transmitted, we’re interested in how long it will take for the data to reach its destination.

  - **Latency**
    - The time that elapses between the source’s beginning to transmit the data and the destination’s starting to receive the first byte.

  - **Bandwidth**
    - The rate at which the destination receives data after it has started to receive the first byte.
Message transmission time = $l + \frac{n}{b}$

- latency (seconds)
- length of message (bytes)
- bandwidth (bytes per second)
38,400-processors, 307,200 CPU cores

20-petaflop [ AMD Opteron 6200 ]

Cray Gemini Interconnect

- processor-to-processor
- Optical network of 2x2 switches
- Banyan: O( Nlog N )

Cache coherence

- Programmers have no control over caches and when they get updated.

Figure 2.17

A shared memory system with two cores and two caches
Cache coherence

y0 privately owned by Core 0
y1 and z1 privately owned by Core 1

x = 2; /* shared variable */

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>y0 = x;</td>
<td>y1 = 3*x;</td>
</tr>
<tr>
<td>1</td>
<td>x = 7;</td>
<td>Statement(s) not involving x</td>
</tr>
<tr>
<td>2</td>
<td>Statement(s) not involving x</td>
<td>z1 = 4*x;</td>
</tr>
</tbody>
</table>

y0 eventually ends up = 2
y1 eventually ends up = 6
z1 = ???
Snooping Cache Coherence

- The cores share a bus.
- Any signal transmitted on the bus can be “seen” by all cores connected to the bus.
- When core 0 updates the copy of $x$ stored in its cache it also broadcasts this information across the bus.
- If core 1 is “snooping” the bus, it will see that $x$ has been updated and it can mark its copy of $x$ as invalid.
Directory Based Cache Coherence

- Uses a data structure called a directory that stores the status of each cache line.

- When a variable is updated, the directory is consulted, and the cache controllers of the cores that have that variable’s cache line in their caches are invalidated.
Parallel Software is Essential for HPC

- Hardware and compilers must keep up the pace as needed.
- In shared memory programs:
  - Start a single process and fork threads.
  - Threads carry out tasks.
- In distributed memory programs:
  - Start multiple processes.
  - Processes carry out tasks.
A SPMD program consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches.

```c
if ( I am thread process i )
    do something;
else
    do more interesting things;
```
Writing Parallel Programs

1. Divide the work among the processes/threads
   (a) so each process/thread gets roughly the same amount of work
   (b) and communication is minimized.

2. Arrange for the processes/threads to synchronize.

3. Arrange for communication among processes/threads.

```c
double x[n], y[n];
...
for (i = 0; i < n; i++)
    x[i] += y[i];
```
Shared Memory

- **Dynamic threads**
  - Master thread waits for work, forks new threads, and when threads are done, they terminate
  - Efficient use of resources, but thread creation and termination is time consuming.

- **Static threads**
  - Pool of threads created and are allocated work, but do not terminate until cleanup.
  - Better performance, but potential waste of system resources.
Nondeterminism

```c
... printf ( "Thread %d > my_val = %d\n", my_rank , my_x ) ;
... Thread 0 > my_val = 7
Thread 1 > my_val = 19
Thread 1 > my_val = 19
```
Nondeterminism

my_val = Compute_val ( my_rank ) ;

x += my_val ;

<table>
<thead>
<tr>
<th>Time</th>
<th>Core 0</th>
<th>Core 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Finish assignment to my_val</td>
<td>In call to Compute_val</td>
</tr>
<tr>
<td>1</td>
<td>Load x = 0 into register</td>
<td>Finish assignment to my_val</td>
</tr>
<tr>
<td>2</td>
<td>Load my_val = 7 into register</td>
<td>Load x = 0 into register</td>
</tr>
<tr>
<td>3</td>
<td>Add my_val = 7 to x</td>
<td>Load my_val = 19 into register</td>
</tr>
<tr>
<td>4</td>
<td>Store x = 7</td>
<td>Add my_val to x</td>
</tr>
<tr>
<td>5</td>
<td>Start other work</td>
<td>Store x = 19</td>
</tr>
</tbody>
</table>
Nondeterminism

- Race condition
- Critical section
- Mutually exclusive
- Mutual exclusion lock (mutex, or simply lock)

```c
my_val = Compute_val ( my_rank ) ;
Lock(&add_my_val_lock ) ;
x += my_val ;
Unlock(&add_my_val_lock ) ;
```
busy-waiting

my_val = Compute_val ( my_rank )
if ( my_rank == 1)
    while (!ok_for_1) /* Busy-wait loop */
x += my_val; /* Critical section */
if ( my_rank == 0)
    ok_for_1 = true; /* Let thread 1 update x */
message-passing

```c
char message[100];
...
my_rank = Get_rank();
if (my_rank == 1) {
    sprintf(message,
            "Greetings from process 1" );
    Send(message, MSG_CHAR, 100, 0);
} else if (my_rank == 0) {
    Receive(message, MSG_CHAR, 100, 1);
    printf("Process 0 > Received: %s\n", message);
}
```
Partitioned Global Address Space Languages

```c
shared int n = ...;
shared double x[n], y[n];
private int i, my_first_element, my_last_element;
my_first_element = ...;
my_last_element = ...;
/* Initialize x and y */
...
for (i = my_first_element; i <= my_last_element; i++)
    x[i] += y[i];
```
Input and Output

- In distributed memory programs, only process 0 will access `stdin`. In shared memory programs, only the master thread or thread 0 will access `stdin`.

- In both distributed memory and shared memory programs all the processes/threads can access `stdout` and `stderr`.
Input and Output

- However, because of the indeterminacy of the order of output to `stdout`, in most cases only a single process/thread will be used for all output to `stdout` other than debugging output.

- Debug output should always include the rank or id of the process/thread that’s generating the output.
Input and Output

- Only a single process/thread will attempt to access any single file other than stdin, stdout, or stderr. So, for example, each process/thread can open its own, private file for reading or writing, but no two processes/threads will open the same file.
Fosters Methodology: The PCAM Method

- **Partitioning**: Decompose computation and data operations into small tasks. Focus on identifying tasks that can be executed in parallel.
- **Communication**: Define communication structures and algorithms for the tasks defined above.
- **Agglomeration**: Tasks are combined into larger tasks to improve performance or to reduce development costs.
- **Mapping**: Maximize processor utilization and minimizing communication costs by distributing tasks to processors or threads.
Foster Algorithm 4.1

Example

1D finite difference problem (FD), in which there is a vector, $X^{(0)}$ of size $N$ that must compute $X^T$, where

$$0 < i < N - 1 : \quad X_i^{(t+1)} = \frac{X_{i-1}^{(t)} + 2X_i^{(t)} + X_{i+1}^{(t)}}{4}$$
A parallel algorithm for this problem creates $N$ tasks, one for each point in $X$. The $i$ th task is given the value $X^{(0)}_i$ and is responsible for computing, in $T$ steps, the values $X^{(1)}_i, X^{(2)}_i, \ldots, X^{(T)}_i$. Hence, at step $t$, it must obtain the values $X^{(t)}_{i-1}$ and $X^{(t)}_{i+1}$ from tasks $i-1$ and $i+1$. We specify this data transfer by defining channels that link each task with `left'' and `right'' neighbors, as shown in Figure 1.11, and requiring that at step $t$, each task $i$ other than task 0 and task $N-1$

1. sends its data $X^{(t)}_i$ on its left and right outports,
2. receives $X^{(t)}_{i-1}$ and $X^{(t)}_{i+1}$ from its left and right inports, and
3. uses these values to compute $X^{(t+1)}_i$.

Notice that the $N$ tasks can execute independently, with the only constraint on execution order being the synchronization enforced by the receive operations. This synchronization ensures that no data value is updated at step $t+1$ until the data values in neighboring tasks have been updated at step $t$. Hence, execution is deterministic.
Foster’s Methodology: PCAM

Figure Ref: Foster, Designing and Building Parallel Programs
Partitioning/Decomposition

1-D  2-D  3-D

Domain decomposition

Atmospheric Model

Hydrology Model

Land Surface Model

Ocean Model

Figure Refs: Foster, Designing and Building Parallel Programs
Partitioning Design Checklist

1. Size of partition $\gg$ # of processors (10x)
2. Partition should avoid redundant computation and storage requirements
3. Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
4. #Tasks must scale with probsize: increase in probsize should increase #tasks rather not size
Communication

- **Local**: task communicates with a small set of other tasks (*neighbors*);
- **Global**: requires each task to communicate with many tasks.
- **Structured**: task & neighbors form a regular structure, such as a tree or grid/matrix
- **Unstructured**: networks may be arbitrary graphs.
- **Static**: identity of communication partners does not change over time.
- **Dynamic**: identity of communication partners determined at runtime
- **Synchronous**: producers & consumers are coordinated (e.g. data xfers)
- **Asynchronous**: consumer obtains data without cooperation of producer.
Communication: 2D Stencil

Jacobi finite difference method

\[ X_{i,j}^{(t+1)} = \frac{4X_{i,j}^{(t)} + X_{i-1,j}^{(t)} + X_{i+1,j}^{(t)} + X_{i,j-1}^{(t)} + X_{i,j+1}^{(t)}}{8} \]
Communication: 2D Stencil Algorithm

Jacobi finite difference method

for $t = 0$ to $T - 1$

send $X_{i,j}^{(t)}$ to each neighbor

receive $X_{i-1,j}^{(t)}$, $X_{i+1,j}^{(t)}$, $X_{i,j-1}^{(t)}$, $X_{i-1,j+1}^{(t)}$

compute $X_{i,j}^{(t+1)}$
Communication: Local

Two finite difference update strategies, applied on a two-dimensional grid with a five-point stencil. Shaded grid points have already been updated to step \( t+1 \). Arrows show data dependencies for one of the latter points. Figure on left is Gauss-Seidel, on right is red-black.
Communication: Global

Centralized summation algorithm
Communication: Global

Figure 2.8: Tree structure for divide-and-conquer summation algorithm with $N=8$. The $N$ numbers located in the tasks at the bottom of the diagram are communicated to the tasks in the row immediately above; these each perform an addition and then forward the result to the next level. The complete sum is available at the root of the tree after $\log_2 N$ steps.

In summary, we observe that in developing an efficient parallel summation algorithm, we have distributed the $N-1$ communication and computation operations required to perform the summation and have modified the order in which these operations are performed so that they can proceed concurrently. The result is a regular communication structure in which each task communicates with a small set of neighbors.

2.3.3 Unstructured and Dynamic Communication

Tree structure for divide-and-conquer summation algorithm with $N=8$. 
Communication Checklist

1. Do all tasks perform about the same number of communication operations?
2. Does each task communicate only with a small number of neighbors?
3. Are communication operations able to proceed concurrently?
4. Is the computation associated with different tasks able to proceed concurrently?
Next class: 09/09/14
HW #1 Due: 09/09/14 at start of class
Finish Parallel Software Overview, look at Unix, Performance Basics