Today’s Topic

- The goal for today: a brief introduction to parallel programming
  - necessary background for Projects 1 and 2
  - introduces main concepts and two styles of parallel programming
  - both styles will be explored in more detail later

- Reading:
  - W&A text: 1.1 -- 1.5, 2.1 -- 2.2
  - MPICH home page: http://www-unix.mcs.anl.gov/mpi/mpich/
  - MPI tutorials: http://www-unix.mcs.anl.gov/mpi/tutorial

Parallel Machines: Flynn’s Taxonomy

- The parallel programming library we use (MPI) is based on an SPMD model
  - SPMD = “single program multiple data”

- To understand this acronym: a historical note on Flynn’s taxonomy
- Processors deal with two types of information
  - instruction stream: sequence of opcodes
  - data stream: sequence of operands

Instruction and Data Streams

- These “streams” are abstractions
- A typical processor chip will have one or two physical connections to memory but one stream of each type
One or Many Streams?

- Flynn’s taxonomy categorizes machines by the number of instruction and data streams

SISD

- Traditional single processor systems
- One processor, both streams fetched from same memory

MISD

- No commercial products based on the MISD design
- Instruction pipelines, systolic arrays are arguably MISD

MIMD

- A wide range of parallel processors falls into the MIMD category
- Use standard CPU chips, each reads its own instruction and data stream
**Distributed Memory Multiprocessor**

- The data streams of a MIMD machine can come from one or more physical memory modules.

  ![Diagram showing distributed memory multiprocessor](image)

- A "cluster" computer, e.g. Beowulf

- Problem: access to remote data (more on this later)

- Ethernet or high capacity switch (e.g. Myrinet)

**Shared Memory Multiprocessor**

- A simple design for a shared memory parallel processor has all processors access a single memory via a bus.

  ![Diagram showing shared memory multiprocessor](image)

- Problem: bus is a bottleneck

- Topic for a future lecture: improving performance through the use of caches and cache coherence

**Distributed Shared Memory**

- My favorite oxymoron…

- Since the bus is a bottleneck, use more than one memory module

- Use special-purpose logic to route memory references to the correct module

  ![Diagram showing distributed shared memory](image)

- Problem: expensive, special-purpose hardware

**SIMD**

- CPU divided into control and datapath (ALU)

- Control unit fetches, decodes single instruction stream from program memory

- Many datapaths (thousands) work on their own data streams

  ![Diagram showing SIMD](image)
**SIMD (cont’d)**

- Examples:
  - Goodyear MPP
  - TMC Connection Machine
  - Maspar MP-1

- Code fragment:
  ```
  lw $t0 4($s1)
  add $t2, $t1, $t0
  ```

- On MP-1: 4096 loads, followed by 4096 adds…

**SPMD**

- The SIMD paradigm was very effective for a surprisingly wide range of applications

- Problems:
  - special-purpose hardware, limited market
  - long paths between control unit, ALUs limit scalability (~20MHz clocks)
  - tight synchronization limited efficiency

- A generalization: Single Program Multiple Data (SPMD)
  - use MIMD architecture
  - run same application on each node
  - give each node a separate piece of the problem to work on

**The Programmer’s View**

- For programmers writing code with more than one process (thread), the distinction between shared and distributed memory depends on the structure of the address space

- **Shared**
  - single address space
  - all variables accessible by all processes

- **Distributed**
  - many separate spaces
  - process can access own variables, but uses messages or other mechanisms to get values from other processes

**Example: Inner Product**

- A simple program to compute the inner product of two vectors:
  
  \[ a = \sum_i x_i y_i \]

- The inner loop in C++:
  ```cpp
  for (i = 0; i < 1000; i++)
    a += x[i] * y[i];
  ```
Inner Product Using SGI's SMP Macros

- On SGI systems, a C++ programmer can use preprocessor macros to tell the compiler how to create threads.
- This code makes four threads:
  ```
  #pragma pfor iterate(T=0; 4; 1)
  for (T = 0; T < 3; T++)
    for (i = T*250; i < (T+1)*250; i++)
      a += x[i] * y[i];
  ```

- “pfor” means “parallel for loop” -- the body of the next for loop will be executed in parallel by each thread.

SGI Example (cont’d)

```python
#pragma pfor iterate(T=0; 4; 1)
for (T = 0; T < 3; T++)
  for (i = T*250; i < (T+1)*250; i++)
    a += x[i] * y[i];
```

**T** = thread ID
**T** is local var in each thread

<table>
<thead>
<tr>
<th><strong>x</strong></th>
<th>250</th>
<th>500</th>
<th>750</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>y</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPMD: four threads will work independently on separate pieces of x, y

Uh-oh…. a is shared…
To see why there is a potential problem when two processors try to update the same variable, consider the machine level code and a possible order of events:

```
# a += x;
    lw $s0, a
    add $s0, $s0, $s1
    sw $s0, a
```

Old value of $a$ in both CPUs

1. Load $a$ from memory into $s0$
2. Add $s0$ and $x$ to get $s0$
3. Store result back to memory
**Requirement for Synchronization**

- To see why there is a potential problem when two processors try to update the same variable, consider the machine level code and a possible order of events:

  ```
  # a += x;
  lw $s0, a
  add $s0, $s0, $s1
  sw $s0, a
  ```

  - Time --->
  - Old value of a in both CPUs
  - New a may be a + x1 or a + x2 instead of a + x1 + x2...

**Critical Regions**

- A solution, based on techniques developed for operating systems:
  - identify sections of code that update shared variables
  - these sections are called critical regions
  - add special instructions before, after regions to allow only one thread at a time to execute code in a critical region

  - We'll look at implementations of synchronization primitives later in the term....

**Critical Region in the SGI Example**

```c
#pragma pfor iterate(T=0; 4; 1)
for (T = 0; T < 3; T++)
    for (i = T*250; i < (T+1)*250; i++)
        #pragma critical
        a += x[i] * y[i];
```

**Local Accumulators**

- Even though a is accessed properly now, it is a bottleneck
  - high probability a process will block while another updates a
  - probability increases when more threads are used

- A better solution uses local counters:

  ```c
  #pragma pfor iterate(T=0; 4; 1)
  for (T = 0; T < 3; T++) {
    for (i = T*250; i < (T+1)*250; i++)
      a[T] += x[i] * y[i];
  #pragma critical
    a += a[T]
  }
  ```
Distributed Memory Version

- In the shared memory implementation, all threads could access the global accumulator a.
- On a distributed memory multiprocessor, a resides on one node.
- Threads running on other nodes cannot access a directly.
- Other threads must send a message to the “owner” of a:
  - Can request current value
  - Can send a value to add to a

Message Passing

- In a C++ program, sending a message is the same as printing to an output stream:
  - compose a string
  - processor issues command to start i/o transfer
  - link copies bytes from memory
- The receiving processor handles the message the way it would handle input from a stream:
  - kernel identifies receiving process
  - copies incoming data to process memory space
  - wakes process to deal with new data

Message Passing Version of Inner Product

Outline of inner product using a message library:

```c
n = my_id(); // this process ID
np = num_procs(); // number of processes
for (i = 0; i < 250; i++)
a = x[i] * y[i]; // local portion of ip
```

// processes 1..np-1 send their ‘a’ to process 0 if (n > 0)
```c
send(&a, 0);
```
(continued next slide)
Message Passing Version (cont’d)

// processes 0 collects partial products
if (n == 0) {
    for (i = 1; i < np; i++) {
        receive(&b,i);
        a += b;
    }
}

Note SPMD nature of this program -- all processes execute same code but on different parts of the global data

MPI

- Message Passing Interface (MPI) is a widely used standard for distributed memory parallel programs
  - also implemented in DM, DSM architectures
  - send message just by passing pointer to data
- MPI routines can be called from C, C++, Fortran, Java, …
- Versions of MPI
  - MPI-1, MPI-2: standards, defined by committee
  - MPICH: portable, open source implementation
    - our version on p690: MPICH2 / MPI-2
  - SGI, IBM, others: products optimized for their hardware

MPI Version of Inner Product

MPI_comm_rank(&n,X); // process ID
MPI_comm_size(&np,X); // number of processes
for (i = 0; i < 250; i++)
    a = x[i] * y[i];
if (n > 0)
    MPI_Send(&a,1,MPI_DOUBLE,0,X);
if (n == 0)
    for (i = 1; i < np; i++)
        MPI_Recv(&b,1,MPI_DOUBLE,i,tag,X,&s);
a += b;
MPI Version of Inner Product

```c
MPI_comm_rank(&n,X); // process ID
MPI_comm_size(&np,X); // number of processes
for (i = 0; i < np; i++)
    a += x[i] * y[i];
if (n > 0)
    MPI_Send(&a,1,MPI_DOUBLE,0,0,X);
if (n == 0)
    for (i = 1; i < np; i++)
        MPI_Recv(&b,1,MPI_DOUBLE,i,0,X,&s);
a += b;
```

MPI on the p690

- There are several C++ compilers (and Fortran, C, ...)
- There are also several MPI installations on the p690
- When you make an MPI program, make sure you compile your application with the same compiler that made the MPI library
- Recommended settings in Makefiles for CIS 455/555 projects:
  ```
  MPI = /opt/absoft_hpcsdk/mpich2-64-1.0.1
  CXX = xlC
  CXXFLAGS += -I${MPI}/include -q64
  LDFLAGS += -L${MPI}/lib
  ```

MPI on the p690 (cont'd)

- To link the MPI libraries with a C++ program use `-lmpich`
  ```
  hello: hello.o
  ${LINK.cc} -o hello hello.o -lmpich
  ```
**MPD**

- MPI programs are coordinated by a process manager
- The preferred manager for MPI on the p690 is **mpd**
- When you start your program, it looks for an **mpd** daemon to talk to
- To start **mpd**: 
  ```
  % mpd &
  ```
- To start your program: 
  ```
  % hello
  ```

**MPD on an SMP**

- The p690 has 16 processors
- To tell **mpd** to use more than one process for your applications, use the **ncpus** switch: 
  ```
  % mpd --ncpus=4 &
  ```
- To run your program in parallel: 
  ```
  % mpiexec -np 4 ./hello
  ```

**MPD on a Cluster or Network**

- We won’t use it this way in CIS 455/555, but you can configure a “ring” of **mpd** processes across several machines
- Initialize your MPI environment on each host (**.mpd.conf**)
- Create a configuration file on the host you want to start the program from
- Start **mpd** on each host, e.g. 
  ```
  % mpirun -np 3 -f mpd.hosts
  ```

**Other MPD Commands**

- Don’t forget to shut down **mpd** before you log off: 
  ```
  % mpdallexit
  ```
- Other useful things: 
  ```
  % mpdtrace
  ```
- Learn more about **mpd** in the MPICH Installation Guide
MPI/MPD Summary

```bash
[p690:hello] % mpd --ncpus=4 &
[1] 30305
[p690:hello] % make
xlC -I/opt/absoft_hpcsdk/mpich2-64-1.0.1/include -q64 -c -o hello.o hello.C
xlC -I/opt/absoft_hpcsdk/mpich2-64-1.0.1/include -q64 -L/opt/absoft_hpcsdk/mpich2-64-1.0.1/lib -o hello hello.o -lmpich
[p690:hello] % mpiexec -n 4 ./hello
Hello from 0 of 4
Hello from 2 of 4
Hello from 1 of 4
Hello from 3 of 4
[p690:hello] % mpdallexit
```

**Start mpd when you log in**

**Compile and test as often as you want**

**Shut down before you log off**