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

"Harvard architecture"
### 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 the same memory

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

- **SIMD**

- **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 an MIMD machine can come from one or more physical memory modules.

- 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.

- Problem: bus is a bottleneck.

- MIMD, because each CPU has own I, D streams.

- 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.

- 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.
**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
- Why it has gone away (for now):
  - 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:
  
  \[
  a = \sum_{i} x_i \times y_i
  \]

- The inner loop in C++:
  
  ```
  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 (int 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)

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

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

0 250 500 750
x
y

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

SGI Example (cont’d)

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

loop iterator i is local to each thread

SGI Example (cont’d)

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

Uh-oh…. a is shared….
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:

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

Old value of `a` in both CPUs
```assembly
lw $s0, a
add $s0, $s0, $s1
sw $s0, a
```
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
```

New a may be \( a + x_1 \) or \( a + x_2 \) instead of \( a + x_1 + x_2 \).

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

```
#pragma pfor iterate(T=0; 4; 1)
for (T = 0; T < 3; T++)
    for (int 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:

```
#pragma pfor iterate(T=0; 4; 1)
for (T = 0; T < 3; T++) {
    for (int 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:
  ```
  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)
      send(&a, 0);
  ```
  (continued next slide)
Message Passing Version (cont’d)

```c
// process 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: old (~2005) open source implementation
  - SGI, IBM, others: products optimized for their hardware
- For our class: OpenMPI
  - http://www.open-mpi.org/

MPI Version of Inner Product

```c
#include "mpi.h"

int main( int argc, char *argv[] )
{
    MPI_Init( &argc, &argv );
    ....
    MPI_Finalize();
    return 0;
}
```
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 < 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 on ACISS

- There are several C++ compilers (and Fortran, C, ...)
- There are also several MPI installations on ACISS
- When you first log in you will see a message explaining how to specify which software packages you want to use

```bash
$ module avail
R/2.13.2 (default)
R/2.14.1
...
mpi/mpich2-1.4.1p1_gcc-4.4.5
mpi-tor/openmpi-1.4.4_gcc-4.4.5
...
```

### Select the MPI Implementation

- Use the “module load” command to specify one of the MPI packages
  ```bash
  $ module load mpi-tor/openmpi-1.5.4_gcc-4.5.3
  ```
- Verify the package is loaded:
  ```bash
  $ module list
  Currently Loaded Modulefiles:
  ...
  8) mpi-tor/openmpi-1.5.4_gcc-4.5.3
  ...
  Recommendation: put the “module load” command in your shell initialization file...
  ```
Compiling an MPI Program

- Compiling with OpenMPI is simple
- Use one of the “lightweight wrappers”
  
  ```$ cd 455/hello
  $ ls
  hello.C  Makefile
  $ mpic++ -o hello hello.C
  $ ls
  hello  hello.C  Makefile```

Launching an MPI Program

- Start the program with `mpirun`:
  
  ```$ mpirun -np 8 hello```

Job Queues

- It’s OK to run the “hello, world” program on the head node
- For future projects, we’ll want to run on the compute nodes
  - 128 basic nodes (2 6-core CPUs, 72 GB RAM)
  - 16 “fat nodes” (4 8-core CPUs, 384 GB RAM)
  - 52 GPU nodes (basic node plus 3 GPUs)
- Put the commands that run your job in a script, then submit it to one of the runtime queues:
  
  ```% qsub -q fatnodes myjob.sh```

  *ACISS sysadmins want us to use a queue called “student”*