Introduction to Parallel Processing

Shared vs Distributed Memory

Introduction to MPI

Running MPI Applications on the p655
Today’s Topic

- The goal for today: a brief introduction to parallel programming
  - necessary background for Project 1
  - 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

![Diagram showing unified cache and split cache architectures]
One or Many Streams?

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

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
</tr>
</tbody>
</table>
SISD

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

- CPU divided into control and datapath
- Control unit fetches, decodes single instruction stream from program memory
- Many (thousands) datapaths work on their own data streams

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>SISD</td>
<td>SIMD</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
</tr>
<tr>
<td>MISD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
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…
**MISD**

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

<table>
<thead>
<tr>
<th>Instruction Streams</th>
<th>Data Streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>MISD</td>
<td>SISD</td>
</tr>
<tr>
<td>Many</td>
<td>Many</td>
</tr>
<tr>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
MIMD

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

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

A “cluster” computer, e.g. Beowulf

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.

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
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++:

```cpp
for (i = 0; i < 1000; i++)
    a += x[i] * y[i];
```
Inner Product Using SGI’s SMP Macros

- On the 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)

```
#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
ID is local var in each thread

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 (i = T*250; i < (T+1)*250; i++)
        a += x[i] * y[i];

loop iterator i is local to each thread

```
x
```
```
y
```
SGI Example (cont’d)

```c
#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];
```

loop iterator `i` is local to each thread

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:

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

```
memory        cpu  cpu
```
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:

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

```
memory  cpu  cpu
lw
```

Time -->
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:

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

Diagram:
```
memory    cpu    cpu
lw         lw
            add
```
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:

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

Old value of `a` in both CPUs

Time ---

```
lw
add
sw
```
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:

```c
# 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 (i = T*250; i < (T+1)*250; i++)
        #pragma critical
        a += x[i] * y[i];
```

Only one thread at a time accesses a
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:

\[
\begin{align*}
n &= \text{my\_id}(); & \text{// this process ID} \\
np &= \text{num\_procs}(); & \text{// number of processes} \\
\text{for} & \quad (i = 0; i < 250; i++) \\
a &= x[i] * y[i]; & \text{// local portion of ip}
\end{align*}
\]
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)
    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;
    }
}

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
  - SGI, IBM, others: products optimized for their hardware
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;
```

“communicator”
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,0,0,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 IBM p655

- There are several copies of MPICH on the IBM system
- Recommended:
  
  ```
  /opt/osshpc/mpich-1.2.5/64/ch_shmem
  ```

  64-bit compiler

  Use shared memory to pass messages
MPI on the IBM p655

- There are several copies of MPICH on the IBM system
- Recommended:
  
  /opt/osshpc/mpich-1.2.5/64/ch_shmem

- Add to C++ project Makefile:

  MPI = /opt/osshpc/mpich-1.2.5/64/ch_shmem
  CXX = /opt/cross/bin/powerpc64-linux-g++
  CXXFLAGS = -I${MPI}/include
  LDFLAGS = -L${MPI}/lib
MPI on the IBM p655 (cont’d)

- To link the MPI libraries with a C++ program:

```bash
hello: hello.o

${LINK.cc} -o hello hello.o -lpmpich
```

- To run the program using 4 processes:

```bash
% setenv PATH
   /opt/oss/hpc/mpich-1.2.5/64/ch_shmem/bin:$PATH

% mpirun -np 4 hello
```