Lecture 2

Address Space Organization
Shared Memory
A first parallel program
Announcements

• Office hours:
  Mondays 2.15p to 3.15p, Thursdays 5p to 6p
• Homework 1 has been posted;
  due next Friday at 9PM
Announcements

• Homework 1 has been posted; due next Tuesday at 8PM

• Section has been changed
  – Friday 2:00 to 3:20 pm, Room TBA
Memory hierarchies and address space organization
Address Space Organization

- We classify the address space organization of a parallel computer according to whether or not it provides global memory.
- When there is a global memory we have a “shared memory” or “shared address space” architecture.
  - multiprocessor
- Where there is no global memory, we have a “shared nothing” architecture, also known as a multicompuler.
Multiprocessor organization

- The address space is global to all processors
- Hardware automatically performs the global to local mapping using address translation mechanisms
- Two types, according to the uniformity of memory access times
  - **UMA**: Uniform Memory Access time
    - In the absence of contention all processors observe the same memory access time
    - Also called *Symmetric Multiprocessors*
    - Usually bus based: not a scalable solution
NUMA

• Non-Uniform Memory Access time
  – Processors see distant-dependent access times to memory
  – Implies physically distributed memory
• We often call these *distributed shared memory architectures*
  – Commercial example: SGI Altix, up to 512 cores
  – Elaborate interconnect with a directory structure to monitor sharers
Architectures without shared memory

• Each processor has direct access to local memory only
• Send and receive messages to obtain copies of data from other processors
• We call this a *shared nothing* architecture, or a *multicomputer*
Hierarchical organization

- Each node is a multiprocessor, usually an SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All multicomputer systems today
- Hybrid programming or message passing
Parallel processing this course

• The course will primarily use SMPs based on multi-core chips
• A brief look at
  – Shared nothing
  – GPUs
Revisiting cache memories
Managing locality with loop interchange

• Data access order affects performance
• The success of caching depends on the ability to *re-use* previously cached data
• Assume a cache with 2 entries, each 2 words wide

```c
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        a[i][j] += b[i][j];
for (j=0; j<N; j++)
    for (i=0; i<N; i++)
        a[i][j] += b[i][j];
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**The 3 C’s**
- Cold Start
- Capacity
- Conflict
Testbed

- 2.7GHz Power PC G5 (970fx)
- Caches: 128 Byte line size
  - 512KB L2 (8-way, 12 CP hit time)
  - 32K L1 (2-way, 2 CP hit time)
- TLB: 1024 entries, 4-way
- gcc version 4.0.1 (Apple Computer, Inc. build 5370), -O2 optimization
- Single precision floating point
Results

for (i=0; i<N; i++)
for (j=0; j<N; j++)
a[i][j] += b[i][j];

for (j=0; j<N; j++)
for (i=0; i<N; i++)
a[i][j] += b[i][j];

<table>
<thead>
<tr>
<th>N</th>
<th>IJ (ms)</th>
<th>JI (ms)</th>
<th># Reps</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>0.007</td>
<td>0.007</td>
<td>$10^4$</td>
</tr>
<tr>
<td>128</td>
<td>0.027</td>
<td>0.083</td>
<td>10</td>
</tr>
<tr>
<td>512</td>
<td>1.1</td>
<td>37</td>
<td>10</td>
</tr>
<tr>
<td>1024</td>
<td>4.9</td>
<td>284</td>
<td>10</td>
</tr>
<tr>
<td>2048</td>
<td>18</td>
<td>2,090</td>
<td>10</td>
</tr>
</tbody>
</table>
Coherence, Consistency, and False Sharing
Cache Coherence

• What happens when processors read and write the same cached memory location
• If one processor writes to the location, all others must *eventually* see the write
• Processors listen ("sniff" or "snoop") bus activity

\[
\begin{array}{c}
X := 1 \\
\text{Memory}
\end{array}
\]
Cache Coherence

- P1 & P2 load X from main memory into cache
- P1 stores 2 into X
- The memory system doesn’t have a coherent value for X
Cache Coherence Protocols

• Ensure that all processors *eventually* see the same value
• Two policies
  – Update-on-write (implies a write-through cache)
  – Invalidate-on-write
Memory consistency and correctness

- Cache coherence tells us that memory will eventually be consistent
- The memory consistency policy tells us when this will happen
- Even if memory is consistent, changes don’t propagate instantaneously
- These give rise to correctness issues involving program behavior
Memory consistency

• A memory system is consistent if the following 3 conditions hold
  – Program order
  – Definition of a coherent view of memory
  – Serialization of writes
Program order

- If a processor writes and then reads the same location X, and there are no other intervening writes by other processors to X, then the read will always return the value previously written.
Definition of a coherent view of memory

- If a processor P reads from location X that was previously written by a processor Q, then the read will return the value previously written, if a sufficient amount of time has elapsed between the read and the write.
Serialization of writes

- If two processors write to the same location X, then other processors reading X will observe the same sequence of values in the order written.
- If 10 and then 20 is written into X, then no processor can read 20 and then 10.
Memory consistency model

- The memory consistency model determines when a written value will be seen by a reader.
- **Sequential Consistency** maintains a linear execution on a parallel architecture that is consistent with the sequential execution of some interleaved arrangement of the separate concurrent instruction streams.
- Expensive to implement.
- **Relaxed consistency**
  - Enforce consistency only at well defined times.
  - Useful in handling false sharing.
False sharing

• Consider two processors that write to different locations mapping to different parts of the same cache line
False sharing

- P0 writes a location
- Assuming we have a write-through cache, memory is updated
False sharing

• P1 reads the location written by P0
• P1 then writes a different location in the same block of memory
False sharing

- P1’s write updates main memory
- Snooping protocol invalidates the corresponding block in P0’s cache
False sharing

Successive writes by P0 and P1 cause the processors to uselessly invalidate one another’s cache
Programming with threads
Shared memory programming with threads

• Program executes a collection of concurrent instruction streams, called *threads*
• We specify # threads ($NT$) when we run the program
• Each thread
  – is usually assigned to a distinct physical processor
  – is initialized with the same code
  – has a *thread index*, a unique integer in [0:$NT-1$]
  – executes instructions at its own rate
• A thread is similar to a procedure call with notable differences
  – A new storage class: shared data
  – A procedure call is “synchronous:” a return indicates completion
  – A spawned thread executes asynchronously until it completes
  – Both share global storage with caller
  – Synchronization is needed when updating shared state (thread safety)
Why threads?

• Processes are “heavy weight” objects scheduled by the OS
  – Protected address space, open files and other state
• A thread, AKA a lightweight process (LWP) is sometimes more appropriate
  – Threads share the address space and open files of the parent, but have their own stack
  – Reduced management overheads
  – Kernel scheduler multiplexes threads
Threads in practice

• A common interface is the POSIX Threads “standard” (pthreads): IEEE POSIX 1003.1c-1995
  – Beware of non-standard features
• Another approach is to use program annotations via openMP
• We will use Kaminsky’s Parallel Java (PJ)
Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads may or may not execute on different processors, and might be interleaved
- Scheduling behavior specified separately
public class HelloSeq {

    // Prevent construction.

    private HelloSeq() {} 

    public static void main (String[] args)
        throws Exception {
            System.out.println("Hello world!");
        }
    }

java HelloSeq
Hello world!
import edu.rit.pj.ParallelRegion;
import edu.rit.pj.ParallelTeam;
public class HelloSmp {
    private HelloSmp() { } // Prevent construction.

    public static void main (String[] args)
    throws Exception {
        int N = 2;
        if (args.length > 0)
            N = Integer.parseInt(args[0]);
        System.out.println ("Starting up " + N + " threads");
        new ParallelTeam(N).execute (new ParallelRegion() {
            public void run() {
                int t = getThreadIndex();
                System.out.println ("Hello world from thread " + t + "!");
            }
        });
        System.out.println ("That's all folks!");
    }
}
Prime number testing (Program1Seq)

prep cs160w
cd $pub/examples/Program1
java Program1Seq 34155013283267 34155013283297 34155013283401 34155013283437

i = 0 call start = 0 msec
i = 0 call finish = 196 msec
i = 1 call start = 196 msec
i = 1 call finish = 389 msec
i = 2 call start = 389 msec
i = 2 call finish = 584 msec
i = 3 call start = 584 msec
i = 3 call finish = 779 msec

private static boolean isPrime (long x) {
    if (x % 2 == 0) return false;
    long p = 3, psqr = p*p;
    while (psqr <= x) {
        if (x % p == 0) return false;
        p += 2;
        psqr = p*p;
    }
    return true;
}
Prime number testing (Program1Smp)

prep cs160w
cd $pub/examples/Program1
java Program1Smp 34155013283267 34155013283297 34155013283401 34155013283437

Seq

i = 0 call start = 16 msec  i = 0 call start = 0 msec
i = 0 call finish = 375 msec  i = 0 call finish = 196 msec
i = 1 call start = 16 msec  i = 1 call start = 196 msec
i = 1 call finish = 375 msec  i = 1 call finish = 389 msec
i = 2 call start = 17 msec  i = 2 call start = 389 msec
i = 2 call finish = 375 msec  i = 2 call finish = 584 msec
i = 3 call start = 16 msec  i = 3 call start = 584 msec
i = 3 call finish = 381 msec  i = 3 call finish = 779 msec
Control flow
Race conditions

• Consider the statement, assuming $x == 0$
  
  $x=x+1$

• Generated code
  
  $r_1 \leftarrow (x)$
  $r_1 \leftarrow r_1 + \#1$
  $r_1 \rightarrow (x)$

• Possible interleaving with two threads
  
  \[
  \begin{array}{c}
  P_1 \\
r_1 \leftarrow x \\
r_1 \leftarrow r_1 + \#1 \\
x \leftarrow r_1
  \end{array}
  \begin{array}{c}
  P_2 \\
r_1 \leftarrow x \\
r_1 \leftarrow r_1 + \#1 \\
x \leftarrow r_1
  \end{array}
  \]

  $r_1(P_1)$ gets 0
  $r_2(P_2)$ also gets 0
  $r_1(P_1)$ set to 1
  $r_1(P_1)$ set to 1
  $P_1$ writes its R1
  $P_2$ writes its R1
Synchronization

- A race condition arises when the timing of accesses to shared memory can affect the outcome
- We say we have a non-deterministic computation
- Memory consistency and cache coherence are necessary but not sufficient conditions for ensuring program correctness
- The Java Memory model (see handout for next lecture) tells us how the memory system will behave
- We avoid race conditions through appropriate synchronization, in our case atomic updates
- PJ class edu.rit.pj.reduction.SharedInt, which is built around Java’s AtomicInteger
- When an operation is atomic, it is indivisible; no other thread can access intermediate state
Programming Lab #1

• You have been provided 1 PJ program and 1 Java Program
• The first program contains various bugs
• Fix the bugs, discuss how you fixed them, and explain your rationale in a report
• The program runs two threads: a Producer and a Consumer
• The Producer generates sequences of integers
• The consumer sums them up
• A counter is used to ensure that
  – the producer cannot generate the next sequence until the consumer has summed up the current sequence
  – The consumer won’t attempt to sum a sequence until the producer has generated it
Programming Lab #1 - continued

• The second program is a serial program that runs a cellular automata (Conway’s Game of Life) and outputs the state of the simulation to a graphical display (via gnuplot)
• Using the techniques employed in the first program, re-implement in PJ to run as two threads, one to run the simulator, the other to handle the display
• Explain your design decisions and verify correctness with the help of provided test cases