Lecture 2

Address Space Organization
Shared Memory Hierarchies
Threads Programming Model
Announcements

• Office hours
  ‣ Mondays 2.15p to 3.15p
  ‣ Thursdays 4p to 5p

• Access to ieng6-203.ucsd.edu
  ‣ This is different from ieng6.ucsd.edu

• Homework will be posted later this evening
Today’s lecture

• Address space organization
  ‣ Shared memory
  ‣ Distributed memory

• Control mechanism

• Threads programming model

• Shared memory hierarchy
  ‣ Cache coherence and consistency
  ‣ False sharing
Address Space Organization

• We classify the address space organization of a parallel computer according to whether or not it provides global memory.

• If there is global memory we have a “shared memory” or “shared address space” architecture.
  › multiprocessor vs partitioned global address space

• When there is no global memory, we have a “shared nothing” architecture, also known as a multicomputer.
Multiprocessor organization

• Hardware automatically performs the global to local mapping using address translation mechanisms
• 2 types, according to uniformity of memory access times
  ‣ **UMA**: Uniform Memory Access time
  ‣ **NUMA**: Non-Uniform Memory Access time
UMA shared memory

- 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 scalable
**Intel Clovertown Memory Hierarchy**

- Intel Xeon X5355 (Intro: 2006)
- Two “Woodcrest” dies on a multichip module

---

**Line Size = 64B (L1 and L2)**

<table>
<thead>
<tr>
<th>Access latency (clocks)</th>
<th>Core2</th>
<th>Core2</th>
<th>Core2</th>
<th>Core2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td>32K L1</td>
<td>32K L1</td>
<td>32K L1</td>
</tr>
<tr>
<td></td>
<td>14*</td>
<td>4MB Shared L2</td>
<td>4MB Shared L2</td>
<td>4MB Shared L2</td>
</tr>
</tbody>
</table>

* Software-visible latency will vary depending on access patterns and other factors

---

**Associativity**

<table>
<thead>
<tr>
<th></th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

---

Sam Williams et al.

---

Scott B. Baden / CSE 160 / Winter 2011
The 3 C’s of cache misses

- Cold Start
- Capacity
- Conflict

Access latency (clocks)

3
14*

* Software-visible latency will vary depending on access patterns and other factors

Line Size = 64B (L1 and L2)

Associativity

8
16

Sam Williams et al.
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 Origin Altix, up to 512 cores
  - Dash prototype at San Diego Supercomputer Center
  - Software/hardware support to monitor sharers
Architectures without shared memory

- A 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*
- Similarity to NUMA
Hybrid organizations

- Multi-tier organizations are hierarchically organized
- Each node is a multiprocessor, usually an SMP
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- All clusters and high end systems today
Parallel processing this course

- We will use an SMP based on multi-core processors
- Primary programming model: threads
- Also look at
  - Message passing
  - Accelerators (GPUs)
Today’s lecture

• Address space organization
  ‣ Shared memory
  ‣ Distributed memory

• Control mechanism

• Threads programming model

• Shared memory hierarchy
  ‣ Cache coherence and consistency
  ‣ False sharing
Control Mechanism

Flynn’s classification (1966)
How do the processors issue instructions?

**SIMD:** Single Instruction, Multiple Data
Execute a global instruction stream in lock-step

**MIMD:** Multiple Instruction, Multiple Data
Clusters and servers processors execute instruction streams independently
**SIMD (Single Instruction Multiple Data)**

- Operate on regular arrays of data
- Two landmark SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine 1 and 2 (1980s)
- Vector computer: Cray-1 (1976)
- Intel and others support SIMD for multimedia and graphics
  - SSE
    - Streaming SIMD extensions, Altivec
  - Operations defined on vectors
- GPUs, Cell Broadband Engine
- Reduced performance on data dependent or irregular computations

\[
\begin{align*}
2 & \quad 1 \\
4 & \quad 2 \\
8 & = \quad 3 + 5 \\
7 & \quad 5 \\
\end{align*}
\]

forall  \( i = 0 : n-1 \)
\[
x[i] = y[i] + z \[ K[i] \]
end forall

forall  \( i = 0 : n-1 \)
\[
\text{if} ( x[i] < 0) \quad \text{then} \\
\quad y[i] = x[i] \\
\text{else} \\
\quad y[i] = \sqrt{x[i]}
\end{align*}
\]
end if
end forall
Today’s lecture

• Address space organization
  ‣ Shared memory
  ‣ Distributed memory

• Control mechanism

• Threads programming model

• Shared memory hierarchy
  ‣ Cache coherence and consistency
  ‣ False sharing
SPMD execution model

- Most parallel programming is implemented under the **Same Program Multiple Data** programming model = “SPMD”
  - Threads
  - Message passing
  - Other names: “loosely synchronous” or “bulk synchronous”

- Programs execute as a set of $P$ processes or threads
  - We specify $P$ when we run the program
  - Each thread/process usually assigned to a different physical processor

- Each process or thread
  - Is initialized with the same code
  - Has an associated *index* or *rank*, a unique integer
  - Executes instructions at its own rate

- Threads communicate through shared memory, processes via messages
Threads programming model

- Program executes a collection of independent instruction streams, called *threads*

- 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 may be 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)
  - Threads share the address space and open files of the parent, but have their own stack
  - Reduced management overheads, e.g. thread creation
  - Kernel scheduler multiplexes threads
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
- Threads communicate via shared memory
- Scheduling behavior dealt with separately
Multithreading in Practice

• POSIX Threads “standard” (pthreads):
  IEEE POSIX 1003.1c-1995
  ‣ Low level interface
  ‣ Beware of non-standard features

• Java threads not used in high performance computation

• Compiler based
  ‣ OpenMP – program annotations
  ‣ “Parallel” programming languages
    • Co-array FORTRAN
    • UPC
Coding with pthreads

```c
#include <pthread.h> #include <cassert.h>
void *Hello(void *arg) {
    sleep(1);
    int64_t _tid = reinterpret_cast<int64_t>(arg)
    int TID = _tid;
    cout << "Hello from thread " << TID << endl;
    pthread_exit(NULL);  return 0;
}

int main(int argc, char *argv[]) {
    int NT = 3;                 pthread_t th[NT];
    for(int t=0;t<NT;t++){
        int64_t t64 = t;
        assert(!pthread_create(&th[t],NULL, Hello,
                                reinterpret_cast<void *>(t64)));
    }
    for(int t=0;t<NT;t++)
        assert(!pthread_join(th[t], NULL));
pthread_exit(NULL);
}
```

% g++ hello.cpp -lpthread
% a.out
Hello from thread 0
Hello from thread 1
Hello from thread 2
% a.out
Hello from thread 1
Hello from thread 0
Hello from thread 2
%a.out
Hello from thread 1
Hello from thread Hello from thread 02

$SPUB/Examples/Hello
Today’s lecture

- Address space organization
  - Shared memory
  - Distributed memory
- Control mechanism
- Threads programming model
- Shared memory hierarchy
  - Cache coherence and consistency
  - False sharing
Cache Coherence

• A central design issue in shared memory architectures
• Processors may read and write the same cached memory location
• If one processor writes to the location, all others must eventually see the write

\[ X := 1 \quad \text{Memory} \]
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 \textit{eventually} see the same value
- Two policies
  - Update-on-write (implies a write-through cache)
  - Invalidate-on-write
SMP architectures

• Employ a *snooping protocol* to ensure coherence

• Processors listen to bus activity

![Diagram of SMP architectures](image-url)
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 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
Eliminating false sharing

- Cleanly separate locations updated by different processors
  - Manually assign scalars to a pre-allocated region of memory using pointers
  - Spread out the values to coincide with a cache line boundaries
 Programming Lab #1

• Given a serial C++ program that tests lists of integers for primality …
• Crate and multithread version in pthreads
• Observe linear speedups, meeting performance goals we specify
• Assignment will be automatically graded
  ‣ Tested for correctness
  ‣ Performance measured
• Code will appear in $PUB/HW/A1$ later today
Fin