Parallel Processing:
Multiprocessor Systems (MIMD)

- MP - A computer system with at least 2 processors:
  - Processor
  - Cache
  - Intercconnection Network
  - Memory
  - I/O

- Q1 – How do they share data?
- Q2 – How do they coordinate?
- Q3 – How many processors can be supported?

Shared Memory Multiprocessor (SMP)

- Q1 – Single address space shared by all processors/cores
- Q2 – Processors coordinate/communicate through shared variables in memory (via loads and stores)
  - Use of shared data must be coordinated via synchronization primitives (locks) that allow access to data to only one processor at a time
- All multicore computers today are SMP

CS10 Review: Higher Order Functions with “CS10: Sum Reduction”

- Useful HOFs (you can build your own!)
  - map Reporter over List
    - Report a new list, every element E of List becoming Reporter(E)
  - keep items such that Predicate from List
    - Report a new list, keeping only elements E of List if Predicate(E)
  - combine with Reporter over List
    - Combine all the elements of List with Reporter(E)
    - This is also known as “reduce”

CS61C Example: Sum Reduction

- Sum 100,000 numbers on 100 processor SMP
  - Each processor has ID: 0 ≤ Pn ≤ 99
  - Partition 1000 numbers per processor
  - Initial summation on each processor [Phase I]
    - Aka, “the map phase”
    - sum[Pn] = 0;
    - for i = 1000*Pn; i < 1000*(Pn+1); i = i + 1
  - Now need to add these partial sums [Phase II]
    - Reduction: divide and conquer in “the reduce phase”
    - Half the processors add pairs, then quarter, ...
    - Need to synchronize between reduction steps
Example: Sum Reduction

```java
half = 100;
repeat
    synch();
    /* Proc 0 sums extra element if there is one */
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

An Example with 10 Processors

```java
sum(P0) sum(P1) sum(P2) sum(P3) sum(P4) sum(P5) sum(P6) sum(P7) sum(P8) sum(P9)
```

Peer Instruction

```java
half = 100;
repeat
    synch();
    /* Proc 0 sums extra element if there is one */
    if (half%2 != 0 && Pn == 0)
        sum[0] = sum[0] + sum[half-1];
    half = half/2; /* dividing line on who sums */
    if (Pn < half) sum[Pn] = sum[Pn] + sum[Pn+half];
until (half == 1);
```

What goes in Shared? What goes in Private?

<table>
<thead>
<tr>
<th></th>
<th>sum</th>
<th>Pn</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>PRIVATE</td>
<td>PRIVATE</td>
</tr>
<tr>
<td>b</td>
<td>PRIVATE</td>
<td>PRIVATE</td>
</tr>
<tr>
<td>c</td>
<td>PRIVATE</td>
<td>SHARED</td>
</tr>
<tr>
<td>d</td>
<td>SHARED</td>
<td>SHARED</td>
</tr>
<tr>
<td>e</td>
<td>SHARED</td>
<td>SHARED</td>
</tr>
</tbody>
</table>

Memory Model for Multi-threading

- All threads have access to the same, globally shared, memory
- Data can be shared or private
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is transparent to the programmer
- Synchronization takes place, but it is mostly implicit

CAN BE SPECIFIED IN A LANGUAGE WITH MIMD SUPPORT – SUCH AS OpenMP

Three Key Questions about Multiprocessors

- Q3 – How many processors can be supported?
- Key bottleneck in an SMP is the memory system
- Caches can effectively increase memory bandwidth/open the bottleneck
- But what happens to the memory being actively shared among the processors through the caches?
Shared Memory and Caches

- **What if?**
  - Processors 1 and 2 read Memory[1000] (value 20)

- Processors 1 and 2 read Memory[1000] with 1000
- Processor 0 writes Memory[1000] with 40

Keeping Multiple Caches Coherent

- Architect’s job: shared memory 🔄 keep cache values coherent
- Idea: When any processor has cache miss or writes, notify other processors via interconnection network
  - If only reading, many processors can have copies
  - If a processor writes, invalidate all other copies
- Shared written result can “ping-pong” between caches

How Does HW Keep $Coherent?$

Each cache tracks state of each block in cache:
- **Shared**: up-to-date data, not allowed to write other caches may have a copy
  - copy in memory is also up-to-date
- **Modified**: up-to-date, changed (dirty), OK to write
  - no other cache has a copy, copy in memory is out-of-date
  - must respond to read request
- **Invalid**: Not really in the cache

2 Optional Performance Optimizations of Cache Coherency via new States

- **Exclusive**: up-to-date data, OK to write (change to modified)
  - no other cache has a copy
  - copy in memory up-to-date
  - Avoids writing to memory if block replaced
  - Supplies data on read instead of going to memory
- **Owner**: up-to-date data, OK to write (if invalidate shared copies first then change to modified)
  - other caches may have a copy (they must be in Shared state)
  - copy in memory not up-to-date
  - So, owner must supply data on read instead of going to memory

Common Cache Coherency Protocol:

- **MOESI** (snoopy protocol)
  - Each block in each cache is in one of the following states:
    - **Modified** (in cache)
    - **Owned** (in cache)
    - **Exclusive** (in cache)
    - **Shared** (in cache)
    - **Invalid** (not in cache)

 Compatibility Matrix:

- Allowed states for a given cache block in any pair of caches
Common Cache Coherency Protocol: MOESI (snoopy protocol)

- Each block in each cache is in one of the following states:
  - Modified (in cache)
  - Owned (in cache)
  - Exclusive (in cache)
  - Shared (in cache)
  - Invalid (not in cache)

```
<table>
<thead>
<tr>
<th></th>
<th>M</th>
<th>O</th>
<th>E</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>E</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>S</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Compatibility Matrix: Allowed states for a given cache block in any pair of caches

```
• Read and Write are by this core.
• Read and Probe Read and Probe Write are reads and writes by others, that must
  probe this core's cache.
```

Cache Coherency and Block Size

- Suppose block size is 32 bytes
- Suppose Processor 0 reading and writing variable X, Processor 1 reading and writing variable Y
- Suppose in X location 4000, Y in 4012
- What will happen?
- Effect called false sharing
- How can you prevent it?

DaMniki’s Laptop? sysctl hw

```
hw.ncpu: 2
hw.byteorder: 1234
hw.memsize: 8589934592
hw.alivecpu: 2
hw.physicalcpu: 2
hw.physicalcpu_max: 2
hw.logicalcpu: 2
hw.logicalcpu_max: 2
hw.cpu: 4
hw.cpusubtype: 4
hw.cpu4bit_capable: 1
hw.cpufamily: 208621756
hw.cachefreq: 2120000000000
hw.cachesize: 4096
hw.bustype: 1046000000
hw.bustype_min: 1064000000
hw.bustype_max: 1064000000
hw.bustype: 306000000
```

And In Conclusion, ...

- Sequential software is slow software
  - SIMD and MIMD only path to higher performance
- Multiprocessor (Multicore) uses Shared Memory
  (single address space)
- Cache coherency implements shared memory
  even with multiple copies in multiple caches
  - False sharing a concern
- Next Time: OpenMP as simple parallel extension
to C