Multicores, Multiprocessors, and Clusters

P. A. Wilsey

Univ of Cincinnati
## Classification of Parallelism

### Software

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Sequential</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>Some problem written as a sequential program (the MATLAB example from the textbook). Execution on a serial platform.</td>
<td>Some problem written as a concurrent program (the O/S example from the textbook). Execution on a serial platform.</td>
</tr>
<tr>
<td>Parallel</td>
<td>Some problem written as a sequential program (the MATLAB example from the textbook). Execution on a parallel platform.</td>
<td>Some problem written as a concurrent program (the O/S example from the textbook). Execution on a parallel platform.</td>
</tr>
</tbody>
</table>
Flynn’s Classification of Parallelism

CU: control unit
PU: processor unit
MM: memory unit
SM: shared memory
IS: instruction stream
DS: data stream

(a) SISD computer
(b) SIMD computer
(c) MISD computer
(d) MIMD computer

Single program, multiple data (SPMD): Single program run on every node on a MIMD style platform.
Parallelism is Hard/Amdhal’s Law

Amdhal’s Law: Idealized parallelism

\[
\text{Speedup} = \frac{1}{(1 - \text{Fraction}_\text{enhanced}) + \text{Fraction}_\text{enhanced} \cdot \text{Speedup}_\text{enhanced}}
\]

Restating

\[
\text{Speedup} = \frac{1}{(1 - \% \text{ affected} + \% \text{ left after optimization})}
\]

Hence the drive to find scalable problems sizes (see textbook example on pg 506) or embarrassingly parallel applications.
Speed-up Challenges

**Strong Scaling:** increasing parallelism in hardware achieves increased speedup.

**Weak Scaling:** increased parallelism is achieved only by increasing the problem size with the hardware parallelism size increases.
Shared Memory Multiprocessors (SMP)

Parallel hardware that presents the programmer with a single physical address space across all processors.

Uniform Memory Access (UMA, nobody uses this acronym): all memory is accessed in near uniform speeds.

NonUniform Memory Access (NUMA, widely used): memory access speed is dependent on location vis-a-vis the processor/memory positions.

Shared memory programming model requires mechanisms (in the hardware) to enable programming to protect shared objects from simultaneous updates.

- locks other atomic primitives
- non-blocking algorithms and atomic operations
- transactional memory (forthcoming in the Intel Haswell processor)
Locks and Other Atomic Primitives

**mutexes:** A locking mechanism to synchronize access to a shared resource.

**Semaphores:** A signaling mechanism to control access to a shared resource.

**Deadlock:** programming error where a collection of tasks are all waiting for a lock/semaphore but the lock is not available and no tasks are configured so that the locked data structure will be unlocked.
Think of a mutex as a **lock/unlock** mechanism. A task locks an object, accesses it, and then unlocks the object. While different tasks can lock the same object, only the task that receives the lock can unlock the resource.

```plaintext

**task 1:**

loop while (!done1)
  lock (x);
  shared_variable++;
  unlock (x);
end loop;

**task 2:**

loop while (!done2)
  lock (x);
  shared_variable--;
  unlock (x);
end loop;
```

In this example, the lock variable `x` is used to protect access to the shared variable `shared_variable`. 
Semaphores

Most often structured as a **counting semaphore** that is initialized with the number of how many simultaneous accesses are allowed. The counting semaphores are variables that are accessed using `wait (P)` and `signal (V)`.

**producer:**

```plaintext
wait (emptyCount);
wait (useQueue);
addItemInToQueue(item);
signal (useQueue);
signal (fullCount);
```

**consumer:**

```plaintext
wait (fullCount);
wait (useQueue);
removeItemFromQueue();
signal (useQueue);
signal (emptyCount);
```

This example illustrates the classic producer/consumer problem. The tasks communicate through a queue of maximum size $N$. Queue overruns/underuns are not permitted. The solution uses three semaphores: `emptyCount` initially set to $N$, `fullCount` initially set to $0$, and `useQueue` initially set to 1.
Non-blocking Algorithms

**Lock-free**: guarantees system wide progress. Basically the system guarantees that some parallel task will progress, but could conceptually starve one (or more) of the tasks.

**Wait-free**: guarantees that all operations have a bounded number of steps to completion. Basically ensuring that no task starves (much stronger constraint than lock-free).
Transactional memory allows the grouping of collections of operations to memory so that they occur as a single atomic operation. The atomic operation to the memory is committed only if no other operation to that memory subspace is serviced during the transaction. Thus, the collection of operations are either all committed or not and when the transaction fails, the programmer must take some action to attempt to accomplish the operation.

Execution an instruction to denote the beginning of a transaction.
Execute one or more memory operations (e.g., x++; y--).
Execute an instruction to denote the end of the transaction.

Test if the transaction was committed.
Message Passing Multiprocessors

Parallel hardware where each processor has its own private address space.

- Information is shared by transmitting messages between processing nodes.
- Most common form is a Beowulf Cluster or collection (tightly coupled or loosely coupled) of computers.
- Often using a standardized message passing protocol such as MPI, PVM, or Infiniband. Of course TCP/IP and UDP can also be used.