CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 1: February 1

Lecturer: Emery Berger *Scribe: Jacob Sorber*

1.1 Course Mechanics

- No exams
- Homework Problems (Parallel Languages, Libraries, etc)
- Term Project

1.2 Parallel Machines

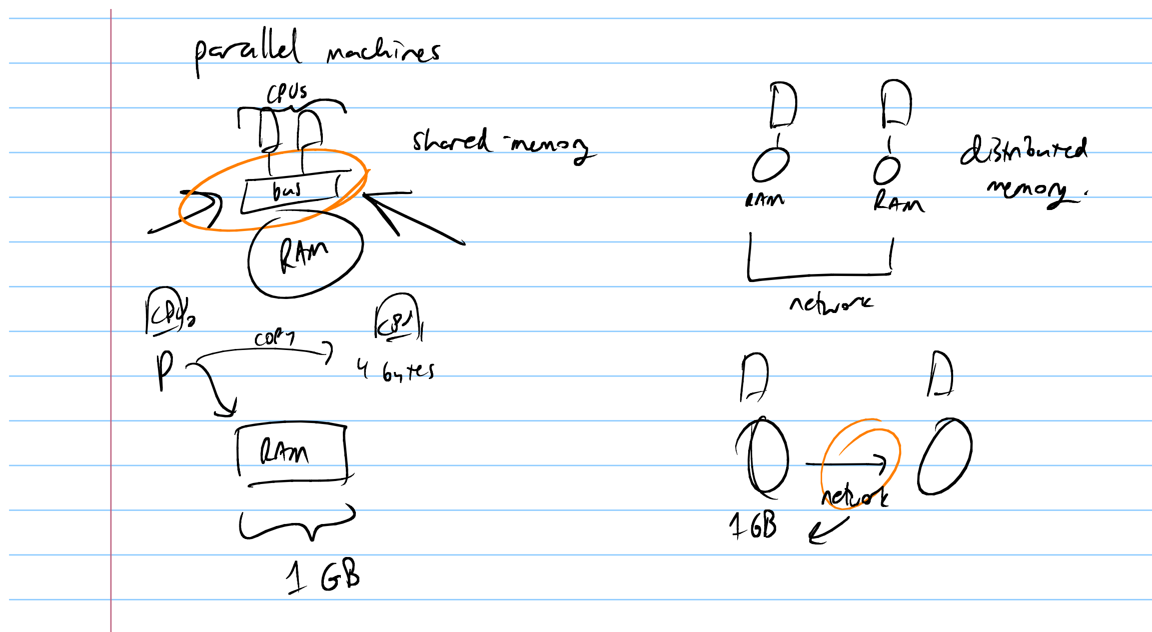

Figure 1.1: Common Parallel Machine Configurations

1.2.1 Shared Memory Systems(Symmetric Multi-Processor)

- + Data passing is very fast. Just pass a pointer.
- + Memory access times same for all accesses.
- - Doesn't scale well. Bus contention typically limits systems to less than 64 processors (on the very high end).

1.2.2 NUMA (Non Uniform Memory Access)

- + Removes the single bottleneck of a shared bus.
- - Memory access times are not predictable
- - Multiple levels of cache (some shared) require complicated coherence protocols.

1.2.3 GRID computing

Grid computing involves a large number of independent systems connected by a network. Message passing typically serves as the mode of communication. Comes with high latency and relatively low bandwidth.

Examples: SETI@home, Folding@home

- Master/slave model
- Master farms out work to slaves
- No inter-slave communication
- Embarrassingly parallel!

1.3 Exploiting Parallelism

There are lots of tasks that are not embarrassingly parallel.

Scientific computing examples:

- Matrix Multiply
- FFT
- N-Body problem
- Successive over-relaxation

1.3.1 Extracting Parallelism

For years people have been extracting parallelism automatically from Fortran code.

Why Fortran?

- Defacto standard for scientific computing
- No recursion
- No pointers

Depends heavily on the architecture it is running on and the topology (see lecture sketch of supercomputer topologies).

1.3.2 Expressing Parallelism

Automatic extraction of parallelism doesn't work for all languages. So, people have proposed using the following:

- Parallel libraries.
- PVM/MPI - Message Passing Interface(effective, but an artificial style for many applications).
- Programming Language extensions.

  HPF - High Performance Fortran

  OpenMP - Standardized loop annotations (i.e. `#pragma OpenMP stripe(10)`)

Problem. How do we know this is right? We don't! Programmers will screw this up.

Why is this interesting now? Multi-core processors have reached the desktop.

Problem. Conventional sequential code runs slower on a multi-core CPU than on a single core chip. Also, conventional apps are not written in Fortran, but C/C++/Java and are parallelized using threads.

1.3.3 Threads

Threads typically offer good performance but poor reliability. Threading is typically supported (C/C++ has POSIX threads, Java has built-in synchronization and threading support).

Why are threads hard to get right?

- Synchronizations problems
- Race conditions (outcome depends on the order of thread execution)
- Hard to debug

1.3.4 Parallel Programming Languages

DARPA High Performance Computing Initiative - Competition between CRAY, SUN, and IBM to provide high performance and high reliability at the language level.

Parallel Language Types

- Explicit (Java w/Threads)
- Implicit

1.3.5 Computation Model

Represent programs as directed acyclic graphs. Nodes represent inherently sequential program segments. Edges represent parallelizable segments.

T_1 = Time to run on 1 CPU = total work to be done

With p CPUs the optimal speedup $= \frac{T_1}{p}$. The best possible speedup (T_∞) is the length of the longest critical path.

If we assume a greedy schedule (Brent schedule),

$T_p = O(T_1/p) + cT_\infty$

Amdahl's Law - Speedup is limited to the parts you can parallelize.

1.3.6 Cilk

Annotate C programs using keywords `spawn` and `sync`, to tell the compiler which code segments can be executed concurrently.

Proof:

```
int fib(int v)
{
    if (v ≤ 1)
    {
        return 1;
    } else {
        int v1 = spawn fib(v-1);
        int v2 = spawn fib(v-2);
        sync;
        return v1+v2;
    }
}
```

■

Cilk implements a greedy schedule using work stealing. Each CPU has a queue and can steal work from another's queue when it finishes all of its work.

Cilk is provably optimal as long as you don't use locks.