Lecture 13: Parallel Programming

CS178: Programming Parallel and Distributed Systems

March 14, 2001
Steven P. Reiss

I. Overview

A. Previously we have looked at

1. Multithreaded programming -- what you can do with one box with a handful of CPUs
2. Client-server programming -- how to deal with systems that have a shared resource that needs to be accessed by a variety of clients
3. Internet programming -- how to do client-server programming over the Internet

B. Today we want to start looking at parallel computing

1. How to do more, faster
2. How to tackle large-scale problems

II. Why Parallel Computing

A. There are hard problems (grand challenge)

1. Weather forecasting
2. Flow analysis (airplane wings; boat hulls; car shapes)
3. Planetary motion (simulating solar system; galaxy; ...)
4. Simulating nuclear explosions
5. Simulating the chemical reactions in a cell
6. Simulating the human brain
7. Categorizing the web

B. The only way of solving these is multiple processors

1. Impossible to do with a single processor
   a) Speed of light and access to data

b) Energy consumption and heat dissipation

c) Switching time for a single transistor

2. Need to have large numbers of correlated processors

a) Small numbers won't gain enough

b) Most problems lack lots of totally independent parts

c) Communications becomes an issue

(1) Must be fast

(2) From and to any processor

3. Multiple threads won't hack it

a) Too few processors

4. Distributed computing won't hack it

a) Communication is too slow

5. Shared memory won't hack it (right now)

a) Too much overhead involved -- slows things down too much

b) Little control over communications

c) This is future research

6. Parallel architecture is then based on message passing

a) Lots of computers (1000+)

b) Connected by a black box network

III. Parallel Architectures : Control

A. Control structure

1. Lots of computers makes things difficult

a) Need to keep them busy

b) Need to do lots of synchronization

c) Lots of complexity to deal with coordination, instruction decoding, ...

2. Control and data can be either single/multiple

a) SIMD

b) SISD

c) MISD

d) MIMD

B. Early parallel systems were SIMD

1. **Single CPU dedicated to control**
2. **Other CPUs are slaves -- all issue the same instructions at the same time**
3. **Conditionals turn on/off cpus where needed**
4. **This turned out to be a difficult model to program for many applications**

C. Other early machines were vector machines

1. **Specialized instructions to deal with vectors**
2. **Parallel hardware to do vector ops quickly**
3. **This worked fine for some applications**
4. **But was hard to use effectively in general**

D. Most current parallel machines are MIMD

1. **Each processor is independent**
   a) Often run the same code, but run it independently
   b) Often 2 or 3 different types of codes are run
2. **Easier to build**
   a) Processors are cheap
   b) Boxes are cheap
3. **Problem that remains is how to connect them**

E. Actual architectures are a mixed bag

1. **Not single processors communicating**
2. **Rather allow multiple processor machines communicating**
   a) SP2 -- has single, dual, and qual processor nodes
   b) This is more standard
3. **Allows a mix of multithreaded and parallel computations**

F. Memory connections with multiple processors

IV. Parallel Architectures :: Communication

A. Problem -- how to connect the processors

1. **Objectives**

a) Communications between any pair of processors should be fast

(1) High bandwidth -- can send lots of data quickly

(2) Low latency -- little overhead in sending any data

b) Communications between A and B should not interfere with communications between C and D

(1) Bisection width -- the number of links that must be cut to divide the network in half

(2) Getting data from all nodes to all nodes

c) Number of links between processors should be small

(1) This is the diameter of the network

2. Limitiations

a) Hardware should be cheap and tractable

b) Cost is proportional to the number of links

B. Network options

1. Completely connected network

a) Fast, efficient communication

b) Doesn't scale -- impossible to implement for any reasonable number of machines

2. Line/Ring network

a) Each processor is connected to two neighbors

b) Communications can be slow -- requires lots of links

c) Need a routing algorithm

(1) Simple -- left or right

d) This is also impractical for high-speed, but is an example

3. Mesh -- can extend a line to an array

a) Each processor is connected to 4 neighbors

b) Practical to build

c) Diameter is 2*sqrt(n)

d) Still need routing algorithms that don't interfere

e) Bisection width is sqrt(n)

4. Torus -- extend the mesh by connecting outside nodes

a) Cuts diameter in half with a few extra connections

b) Routing algorithms become a bit more complex

5. Tree

a) Suppose the nodes are organized in a tree

b) Diameter then becomes 2*log(n)

c) Number of connections per node is small however

d) But there are serious communications bottlenecks -- bisection width is 1

e) Still good for divide and conquer

6. Hypercube

a) Extend the idea of a mesh into higher dimensions

b) Show 3D hypercube

c) Each element has 2K neighbors

d) Diameter of the network is log(n)

e) There are good routing algorithms

f) Bisection width is high as well

7. Butterfly

a) Basic idea:

(1) 4 nodes, 2x2 (A-B / C-D)

(2) Connections AC, AD, BC, BD (butterfly pattern)

(3) Repeat this with the 4 nodes being a unit (recurse)

b) Result is log(n) diameter with small set of connections

c) Fast routing and high bisection width

8. Ethernet routines -- using networks of workstations

a) Original ethernet

b) Ethernet with routers and hubs

c) Star networks, nested star networks

d) These are more like n-ary trees

C. Embedding

1. Hardware is generally fixed in architecture

2. Problems might require different architectures

a) Want to think of the architecture differently than it actually is

b) This can be done by embedding

3. Exampe: embedding a tree in a mesh

a) Start with root at the center

b) Spacer node in X+, X- direction, then next sons

c) Spacer node in Y+, Y- direction, then next sons

d) Nodes in X+, X- direction

e) Nodes in Y+, Y- direction

4. This is often done to map the problem to the hardware

V. Routing Algorithms

A. This is an interesting area

1. But not of too much interest to the programmer, more to the hardware designer

B. Problems to deal with

1. Each node needs to maintain a queue of messages

2. Need to send messages along connection to get it there appropriately

3. Need to minimize queue sizes, wait times for messages

4. Want to optimize traffic separation between nodes

a) Static guarantees are nice

b) Might want to do dynamic scheduling

5. Deadlock and livelock a problem with buffers

a) Circularly full buffers

VI. Input/Output

A. This is an essential part of most computations

1. Can provide shared disk to all processors

a) This can become a bottleneck

2. Can provide each processor with a disk

a) How to distribute the data initially

3. Can have dedicated I/O processors

B. No fixed solutions

VII. Potential versus Promise

A. Suppose we have k processors, what type of speed up can we get

1. How to measure effectiveness of the program

2. How to measure effectiveness of the hardware

B. Communications costs

1. Computation/communication ratio

2. How to maximize this

a) Fewer processors

b) More computation before communication

c) Often problem-specific

C. Speedup factor

1. Execution time on 1 processor / time w k processors

2. This can be viewed in actuality (real timings)

3. Or it can be viewed algorithmically

a) Best uniprocessor solution / best n-processor solution

b) Sorting :: unary is n log n; kary is 4n, speed up is 1/4 logn

c) Some problems can't be sped up much, others can

D. Overhead

1. Factors that limit speedup but don't affect computation

a) Periods where not all processors have something to do

b) Extra computations needed for parallelization

c) Communication time

d) Periods where processors have to wait for others to compolete

2. Efficiency of a system

a) Exec(1) / Exec(n) * n

E. Amdahl's law

1. Let f be the fraction of the computation that cannot be divided into concurrent tasks

2. **Then Time is f*t + (1-f)*t/n**
3. **Then Speedup is n / (1+(n-1)f)**
4. **As n goes to infinity, max speedup is 1/f**

F. Scalability

1. **What happens if the problem size increases**
2. **What happens if the processor size increases**
3. **Look at scaled speedup**
   a) Let s be the sequential time of the program
   b) Let p be the parallel time part of the program
   c) Then SS(n) = (s+np)/(s+p)
   d) If s+p == 1, then this becomes n + (1-n)s
   e) This is Gustafson's law

VIII. The future

A. We'll start looking at MPI for message passing

1. **What are the primitives, how is it used**
2. **Examples, etc.**

B. Designing parallel applications with MPI

1. **How to achieve best performance**
2. **Both algorithmically and practically**