CS 221 – May 10

- Course objectives
  - Assemble computer cluster
  - Linux and C
  - Practice basic parallelizing technique

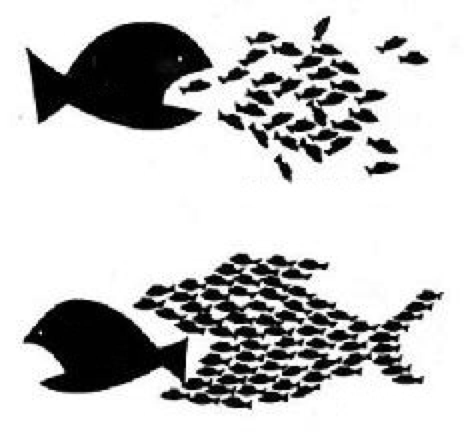
- Stay tuned: I will provide some useful files for you on

  http://cs.furman.edu/~chealy/cs221

- Please read chapter 1 in book.

Motivation

- Stove?
- Raspberry Pi is a very inexpensive, fully functional computer.
  - Similar power to a Pentium 2 (Vintage 1998 PC) but costs about $50.
  - I did an experiment yesterday to compare a pi with my research server. The pi was 28 times slower, but cost 140 times less.
  - So, more bang for the buck!



First steps

- We can't do any parallel programming yet until we
  - Have multiple machines hooked up
  - Understand how to use the operating system.
  - Can write a simple program in C
- Why C?
  - Most compiler research is done for this language.
  - Small, efficient language; similar to Java.
- Today's priority
  - Get one Raspberry pi up and running.

Cast of characters

- A cluster will consist of
  - n Raspberry Pi units
  - Some way to encase the bare boards
  - n SD cards to store OS, other software, and user files
  - n power cords
  - n power adapters
  - n network (Ethernet) cords
  - 1 keyboard, monitor and mouse
- We'll use the setup checklist from the University of Southampton. Set up 1 machine first! That takes about 2 hours.

CS 221 – May 11

- Operating system: Linux
  - What is an OS?
  - Where does Linux come from?
- C language
  - Begin looking at the overview
  - Tomorrow we'll study this subject in earnest.
- Handouts
  - Lab on Linux activities
  - Overview of C

CS 221 – May 15

- Review chapter 1
- Lab
  - Show me your C programs
  - Black spaghetti – connect remaining machines
  - Be able to ping, ssh, and transfer files among nodes

- For tomorrow, please read sections 2.4, 3.1 and 3.2 to get ready to write real parallel programs!
  - Chapter 3 contains nuts and bolts
  - Chapter 2 has background material

Chapter 1 ideas

- Why is parallel computing necessary in the long run?
  - Moore's law
  - Speed of light → size of chip
- It's a problem of HW as well as SW: how?
- How can we tell if a problem could benefit well from a parallelized solution?

Problem solving

- Generally, we begin by writing an ordinary “serial” program.
  - Benefits?
- Then, think of ways to redo the program to take advantage of parallelism
  - There is no automatic tool like a compiler to do all the work for us!
  - Often, the data needs to be partitioned among the individual processors.
  - Need some background software (e.g. MPI) to handle low-level communication details.

Simple example

- Suppose we wanted to sum a large set of values.
  - You are an investment firm, and your fund owns shares in 10,000 securities. What is today's total asset value?
  - Serial solution is straightforward, but a bit slow.
- Suppose we have 5 computers working on this task at once
  - Design 1 program that can shared among all 5 machines.
  - What information is passed to each machine?
  - What happens when they are finished?

Delegate data collection

- It's tempting to just have the "master" node add up all the data.
- If you have many cores, this is not efficient
  - Suppose you have 1 million securities to look up and 1,000 cores sharing the workload. Then, the master node is receiving messages from 999 nodes (in series!), and must add up the 1,000 values.
  - As much as possible, we want results to be communicated simultaneously.
  - Let's work out a simple scenario with 8 cores. Each contains a number that needs to be added. How would you do it? How would the pseudocode look?

Major principles

- A good parallel solution pays close attention to 3 concepts. What do they mean? What could go wrong?

- Communication
- Load balancing
- Synchronization – each core is working at its own pace, for example reading input…

CS 221 – May 16

- Overview of 2.4, 3.1, 3.2
- Lab
  - Do all of your machines work?
  - Let's work through simple examples in book

Parallel software

- Not much software today is currently done with parallelism in mind. Basically limited to:
  - Operating system (!)
  - Databases: allowing you to modify 1 record or table, while someone else can print out some other table
  - Web browser: multimedia doesn't cause machine to hang; multiple tabs

Flynn's taxonomy

- A computer system can be classified based on how many instruction and data streams it can handle simultaneously.
  - SISD (basic computing model)
  - SIMD: the same program is used on a wide stream of data, such as a vector processor
  - MIMD ☺: several cores or processors running *independently* at the same time. Can run same program, but not executing identical statements in lockstep.

MIMD flavors

- Shared memory model
  - You can write a Java program with multiple threads
  - The OS tries to put a new thread on another core.
  - If not enough cores, OS performs multitasking by default.

- Distributed memory model
  - Writing a program that will be run on many computers at once, each with its own memory system, architecture and OS!
  - For convenience we have chosen to have a cluster with the same architecture & OS on each. ☺

SPMD

- Not to confuse with SISD, SIMD, MIMD…
- Single program multiple data: this is the way we will write our parallel programs
  - If-statement condition asks which machine we are on
- A program needs to:
  - Divide computational work evenly
  - Arrange for processes to synchronize (wait until done)
  - Communicate parameters and results.
- How? By passing messages between the processes!
  - We’ll use MPI software (Chapter 3)

MPI

- Message Passing Interface is a library of functions to help us write parallel C programs.
- Once you have written your *parallel* program, compile and run it.
  - (p. 85) mpicc –g –Wall –o mpi_hello mpi_hello.c
  - (p. 86) mpiexec –n 8 ./mpi_hello

    assuming you have 8 machines. You can even give it a larger number, since it's the number of processes you want. (multitasking)

Code features

- In lab I'd like you to type in programs in sections 3.1 and 3.2. Pay attention to details we're seeing for first time.
- What's new?
  - mpi.h
  - MPI_Init() at beginning and MPI_Finalize() at end
    (allocate & deallocate resources needed)
  - MPI_Comm_size() – how many processes are running?
  - MPI_Comm_rank() – which process am I? By convention 0 is the master, and the rest are 1, 2, … n – 1.
  - MPI_Send() and MPI_Recv() ☺

Lab

- Purpose: to be able to use basic MPI functions for the first time.

- Section 3.1
  - Type in mpi_hello.c program
  - Compile & run
- Section 3.2
  - integral.c given on pages 98-99. Note that you also need include files and a function to integrate.
- Answer questions 3.1 – 3.3 on page 140.

CS 221 – May 22

- Warning about possibly overwriting your source code if you call mpicc incorrectly
- What would you do if the size of an array is not a multiple of the # of processes?
- Reminder on programming assignments
- Section 3.5: derived data types

CS 221 – May 24

Timing (sections 2.6 and 3.6)

- Speedup ☺
- Amdahl's law
  - What happens if you can't parallelize everything
- Complexity
- Commands to put in your program to measure time

Amdahl's law

- A factor of n improvement to some aspect of your program doesn't improve total performance this much. Only applies to the feature being improved.
  - Improving half of your program → you can't expect even to double performance.
  - Ex. A factor of 10 improvement on a computation that originally took 40% of the time. The other 60% was unaffected. The "40" becomes 4, so the total time is 4+60 rather than 40+60. So the speedup is only 100/64 = 1.56, not 10!
- Loss leader doesn't bankrupt a store

Complexity

- You have 8 processors working for you. This means up to an 8x speedup
- What works against you? Algorithm complexity
- Nested loops
  - If the problem/input size is n, we must perform n^2 steps
  - Or even n^3 steps if we have 3 nested loops.
  - What does this mean if we double or quadruple the input size?
- Square matrix operations: add and multiply

2-D as 1-D

- To help with the process communication, it helps to represent our 2-D arrays as 1-D.
- Much easier to dynamically allocate 1-D array. ☺

- In our example, we'll assume the matrix is square, so size = # rows = # columns.
- If you want to refer to the element at row i,column j, then say a[i * size + j].

Timing your code

- Insert a call to MPI_Wtime()
  - Returns a double, representing current time in seconds
- But we actually want elapsed time
  - Early in program: start = MPI_Wtime()
  - End of program: finish = MPI_Wtime() – start
  - Any other place also: milestone = MPI_Wtime() – start
  - At end, print values of all timings (from each process)
- Where should we insert these timing probes?
- Re-run your program with:
  - 1 or multiple processes on 1 processor
  - Multiple processes on multiple processors

CS 221 – May 25

- Sorting in parallel (section 3.7)
- Special algorithm: parallel version of bubble sort.
- Lab:
  - Please implement a serial version of this algorithm.
  - Save the parallelizing of it until tomorrow

Sort

- Arrange elements of an array in order
- Let’s assume we have an array of integers, to sort in ascending order
- At some point, sorting requires some elements to be swapped
  - It would be nice if these elements are “near” each other. Why?
  - Bubble sort sounds like a good starting point.
  - But pure bubble sort only looks at adjacent elements. Need a way to look “a little” farther away

Parallel sorting

- We'll play with a special version of bubble sort that is specially designed to be parallelizable.
- We'll treat the 1-D array as if it's 2-D
  - Ability to compare to neighbor on right and below ☺
- It's most convenient for the number of entries to be an even perfect square.
  - Eventually, we'd want each process to get an even # of rows.
  - Thus, we'll assume the size of the array is of the form $(2pk)^2$, where k = 1,2,3,… (even perfect square)
  - For parallelizing, p = # processes. So, if p = 8, it would be good to try 16^2, 32^2, 48^2, etc.

Overview

- Handout example assumes 16 entries, so we can arrange as 2-D array (4x4).
- We'll perform several passes over the array.
  - Odd number pass: Sort the rows. Even rows ascending; odd rows descending
  - Even number pass: Sort the columns. All columns sorted in descending order.
  - After each pass, see if array completely sorted. If so, quit.
  - You only need about n passes, where n = # rows.

Example

31	2	7	5
12	26	31	3
16	20	23	19
18	4	13	32

1. Sort by rows:

2	5	7	31
31	26	12	3
16	19	20	23
32	18	13	4

2. Sort by columns:

32	26	20	31
31	19	13	23
16	18	12	4
2	5	7	3

3. Sort by rows:

20	26	31	32
31	23	19	13
4	12	16	18
7	5	3	2

4. Sort by columns:

31	26	31	32
20	23	19	18
7	12	16	13
4	5	3	2

5. Just one more pass and we’re done! Where is our answer?

Handing each row & col

- The "inner" part of the algorithm looks at individual rows and columns
- Odd number pass: look at rows only
  - Compare elements [0] with [1]; [2] with [3]; [4] with [5]…
  - Compare elements [1] with [2]; [3] with [4]
  - This corresponds to handout: people looking to their right
  - Careful: alternating sense
- Even number pass: look at columns only
  - As with rows, look at [0] and [1], [2] with [3] etc.
  - And then look at [1] with [2], [3] with [4], etc.

Lab

- Functions you may need besides main:
  - Randomize – to initialize array to random data
  - Sort – overview of algorithm
  - Swap two integers
  - Print_array_2d – to see progress as program runs
  - Is_already_sorted – needed after every pass of algorithm
    - The even numbered rows must be ascending, left to right
    - Odd rows must be descending from left to right
    - Also, lowest # on each row must be >= highest number on next row
  - Snake – to convert 2-d back into 1-d answer.

CS 221 – May 26

- What is it good for?
  - Many trials or iterations needed for maximum precision
  - Large input
  - Intractable problems
- Please note:
  - H2 due 5pm today
  - Q2 at start of Tuesday
  - H3 due next Wednesday

Redistricting

- Redraw congressional district boundaries every 10 years.
- The 2010 census showed 4,625,364 people in SC. State is entitled to 7 congressional districts. But they must be (nearly) identical in population.
- 660,766 per district
- Precision of census is the block.
- Ex. Start in one corner of state, and add up blocks until you reach target. Continue with next district.
- (Another stipulation: Voting Rights Act)

Intractable problems

- Problems with no known efficient solution
- See Ron Graham video, excerpt 23:45-33, 47:55-51
- Subset sum problem: Given a list of numbers, is there some subset that adds up to a target value?
- Partition problem is similar: can the list be split into 2 parts that add up to the same total?

Intractable problems (3)

- String matching ("Post Correspondence Problem")
- Given a set of dominoes
  - Each contains a string on the top and bottom
  - Use the dominoes so that the strings on the top and bottom match.
  - You may use each domino as many times as you like. But there must be one domino. ☺
  - The solution is the sequence of dominoes (e.g. 1,2,3)

11	100	111
111	001	11

String matching, cont'd

- Can you find a solution to this one?

1	10111	10
111	10	0

Or this one?

10	011	101
101	11	011

Sudoku

- Methodically try all possible solutions one at a time
- Along the way, can create some ad hoc heuristics to help rule out cases, but still a lot of searching!

Good luck!

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	