AMSC/CMSC 662
Computer Organization and Programming for Scientific Computing

Fall 2011

Introduction to Parallel Programming

Dianne P. O'Leary

Introduction to Parallel Programming

Why parallel computing:

- Get results faster than we can with a single processor.
- Take advantage of hardware that is in our laptop, or lab, or cloud.

You've already had an informal (but important) nontechnical introduction.

Now we have to program parallel computers using what you have learned.

Parallel Computing Models

How to communicate:

Two basic approaches

- Shared-memory computing: a set of processes operate independently on their own data and shared data, communicating using the shared memory.
  - Overhead of sharing ≈ overhead of moving data to and from memory.
- Distributed-memory computing: a set of processes operate independently on their own data communicating by passing messages to other processes.
  - Overhead of sharing ≈ overhead of doing input/output (maybe 10,000 times the cost of memory access) + delay of passing the information over a bus or wireless network.

How to organize the work:

Two basic approaches

- Master/slave processes:
  - One process spawns and controls all the others, handing out work and collecting the results.
  - Vulnerable to the health of the master process. If it fails, the job fails.
  - The master process becomes a bottleneck, and often the spawned processes have to wait for attention.
  - Even in computing, slavery is not a good idea.
  - But often there is a parent process that spawns the child processes, and this can be a convenient way to organize the program.
- Communicating processes:
  - Each process operates somewhat independently, grabbing undone work and producing and communicating the results.
  - More robust to hardware failures and load balancing done dynamically.

How to do the work:

Two basic approaches

- SIMD: Single Instruction, Multiple Data.
  - Each process has the same set of instructions (the same computer code), and it operates on a set of data determined by the ID of the process.
  - Much easier than writing a different program for every process!
  - Processors in lockstep synchronization – this leads to inefficiency when one process has to wait while others do something that it doesn't need to do.
  - May be able to use a single fetch/decode hardware component, so hardware tends to be cheaper and faster (smaller).

- MIMD: Multiple Instruction, Multiple Data.
  - Each process can have a different set of instructions and it operates on a set of data determined by the ID of the process.
  - Writing the program can be much more complicated.
  - Processors in much looser synchronization – can work freely until they need to get or give information to another process.
  - Tends to be much more efficient, with less waiting.
  - Processors take more space, so communication is slower.

How to write parallel programs

There are a few competing models:

- MPI: Message Passing Interface.
  This system provides functions in C, Fortran, etc. to allow operations such as:
  - spawning processes.
  - passing a message to a process.
  - passing a message to all processes.
  - gathering information from all processes.

  It does not support shared memory.
- Pthreads: Parallel Threads.
  This system provides low-level support so that shared memory parallel computing can be programmed in C.

- OpenMP: Open MultiProcessing.
  - This system provides somewhat higher-level support for shared memory parallel computing in C.
  - It can also be used for message passing.
- Cuda: Compute Unified Device Architecture
  - This functions as a virtual machine model for a set of graphical processing units (GPUs).
  - Hardware that satisfies these assumptions can be programmed using an extension to C or other languages.
- Parallel computing can also be done from high-level languages such as MATLAB.
  - See `http://www.mathworks.com/help/toolbox/distcomp/distcomp_product_page.html`
  - The university does not have enough licenses to support us playing with this as a class.

The long established custom of lying about parallel computing

- “My new algorithm....”
- “My algorithm gets 90% of peak performance....” (running an unacceptable algorithm)
- “On n processors, my speed-up over a single processor is $n + 1$...” (running an unacceptable algorithm)
- “My algorithm has 90% efficiency....” (on a problem of size 10,000 using 8 processors)
- “Of course my program will run after I leave....”

Fact 1: Most “new” parallel algorithms are “old”.

- Oldest algorithms I know of: developed for rooms of women (called “computers”) computing tables for ballistics and other applications.

http://en.hnf.de

- In the 1970s and 1980s there was a very large number of experimental and commercial parallel computers

  SIMD: ILLIAC IV, DAP, Goodyear MPP, Connection Machine, CDC 8600, etc.

  MIMD: Denelcor HEP, IBM RP3, etc.

  MIMD Message passing: Caltech Cosmic Cube, nCUBE, Transputer (multiple processors on a single chip), etc.

  Vector pipelining: Cray-1, Alliant, etc.

  and an even larger number of researchers publishing parallel algorithms.

If you think of it, they quite possibly thought of it.

Why almost all of these companies failed

- Hardware issues: The mean-time-to-failure for the hardware was too short.
- Software issues: The computers were very, very difficult to program.
- Cost: The computers were very expensive and consumed a tremendous amount of power.

Fact 2: Efficiency can be deceiving.

All you need to do is perform a lot of operations that don't interfere with one another.

A program like this can get a large fraction of peak efficiency – say, perform 90% of the maximum GFLOPS.

It is much harder to get high-efficiency programs to do useful work.

So don't be impressed by graphs showing large GFLOPS.

Example: If I want to solve a linear system $\mathbf{Ax} = \mathbf{b}$ for a typically used "model problem" where $\mathbf{A}$ is the 5-point finite difference approximation to $-u_{xx} - u_{yy}$, I can easily get 90% of peak performance on SIMD machines using the Jacobi algorithm. (To be continued.)

Fact 3: Speed-ups can be deceiving.

Speed-up is (time on a single processor) / (time on the parallel computer).

Example (continued) The speedup of Jacobi's method will also be terrific – almost a factor of n if we use n processors on a problem with n unknowns.

But we are not computing the right performance measures.

- Jacobi's method is not the best sequential algorithm, so it is much more reasonable to define speed-up to be (time for best algorithm on a single processor) / (your algorithm's time on the parallel computer).
- Similarly, efficiency should really be measured using (time for best algorithm on a single processor)/(number of processors * your algorithm's time on the parallel computer).

Fact 4: Scalability should not be neglected.

Efficiency and speed-up need to be measured over a whole range of problem sizes and number of processors.

Great debates about whether we should plot efficiency for various problem sizes for a fixed number of processors, or whether we should increase the number of processors when we increase the problem size.

Fact 4: Portability should not be neglected.

- Don't invest a year of your effort into designing for hardware that will disappear in another 6 months.
- Use programming constructs from MPI, OpenMP, etc., even if they slow your program down, because then your code can be used on other computers, even after your computer disappears.

Our study of parallel programming

- We will program in OpenMP.
- We will talk about GPU programming and do an in-class exercise.

An excellent reference

Peter S. Pacheco,
An Introduction to Parallel Programming
Morgan Kaufmann
2011

The Pacheco book's website:
`http://textbooks.elsevier.com/web/product_details.aspx?isbn=9780123742605`

We will cover Chapter 5.

You can find notes on other chapters and all of the code from the book, on the book's website.