CSE 160

Introduction to parallel computing Fundamentals

Scott B. Baden
Welcome to Parallel Computation!

• Your instructor is Scott B. Baden
  ‣ Office hours week 1: Tue (10/1), 3-5p, rest TBA
• Your TAs: Jing Zheng & Yashodhan Karandikar
• Your Tutors: Gordon Wong, Parry Wilcox, Ivan Tham
• Lab/office hours: TBA
• Section: Wednesdays in CSB 002
  2:00 to 2:50 pm
  5:00 to 5:50 pm
• Scheduling
  ‣ SDSC tour October 22nd
  ‣ Lecture on 10/8 Moves to Mon 10/7, time and location TBA
Reading

• Two required texts
cseweb.ucsd.edu/classes/fa13/cse160-a/texts.html

• Complete the readings before class:
  readings→in class problems→ quizzes→exams

• All class announcements will be made on-line so check frequently
  ▶ Course home page
  http://www-cse.ucsd.edu/classes/fa13/cse160-a
  ▶ Moodle

• On-line materials
  http://www-cse.ucsd.edu/users/baden/Doc
Background

• Pre-requisite: CSE 100
• C/++ programming experience
• Are you familiar with the following?
  ‣ Threads or other form of parallel computation
  ‣ Cache memory hierarchies
Course Requirements

• [4] Programming assignments (50%)
  ‣ Includes a lab writeup
  ‣ Assignments shall be done in teams of 2

• Exams (40%)
  ‣ Midterm (15%)
  ‣ Final (25%)

• [5] quizzes in section (10%)
  ‣ Lowest score will be dropped
Policies

• Academic Honesty
  ‣ Do you own work
  ‣ Plagiarism and cheating will not be tolerated

• By taking this course, you implicitly agree to abide by the following the course polices:

http://www-cse.ucsd.edu/classes/fa13/cse160–a/Policies.html
Programming Labs

• Bang cluster
• Ieng6
• Make sure your accounts work (Ieng6 coming)

• Software
  ‣ C++11 threads
  ‣ MPI
Have you ever had any of these experiences in lecture?

- The instructor went too quickly, you got missed an important point, and got lost
- You were texting on your phone and lost your place
- You got bored and then something exciting was discussed in class, but you were last
- After getting lost about 10 minutes into lecture, the entire rest of the lecture is completely incomprehensible…
- You wish you had just 30 more seconds to write down that note …
- 2 slides later you realize that you didn’t get the material 3 slides ago…
- You are completely sure you got it…
  ‣ But later on that midterm question is really hard
  ‣ Or you weren’t sure why your implementation was inefficient
Introducing the Two-Minute Pause

• Research has shown that learning increases when
  ‣ After every 8-12 minutes of lecture
  ‣ Turn to the person next to you and discuss/review what was just discussed in lecture

• You can
  ‣ Double check your notes
  ‣ Discuss how you understand something
  ‣ Discuss what is important to know
  ‣ Ask a question

Please pay attention and quickly return to “lecture mode” so we can keep moving!
Group Discussion Questions

• **Opportunity in class to develop understanding, make sure you “grok” it**
  ‣ By trying to explain to someone else
  ‣ Getting your brain actively working on it

• **What will happen**
  ‣ I pose a challenging question
  ‣ You discuss with 1-2 people around you
    • Most important is your understanding of why the answer is correct
  ‣ After most people seem to be done
    • I’ll ask for quiet
    • A few will share what their group talked about
      – Good answers are those where you were wrong, then realized…
Lecture technique

- Learning is not a passive process
- Active lecture to keep us engaged
- Different modalities
  - The 2 minute pause
  - In class problem solving
Group Discussion #1
What is your Background?

- C/C++
- Java
- Fortran?
- TLB misses
- Multithreading
- MPI
- RPC
- C++11 Async
- CUDA, GPUs
- Abstract base class

\[ \nabla \cdot u = 0 \]

\[ \frac{D\rho}{Dt} + \rho(\nabla \cdot v) = 0 \]

\[ f(a) + \frac{f'(a)}{1!}(x - a) + \frac{f''(a)}{2!}(x - a)^2 + \ldots \]
The rest of the lecture

- Syllabus
- Intro to parallel computation
What you will learn in this class

• How to solve computationally intensive problems on parallel computers effectively
  ‣ Theory and practice
  ‣ Software techniques
  ‣ Performance tradeoffs

• Emphasize multi-core implementations, threads programming

• CSE 160 will build on what you learned earlier in your career about programming, algorithm design & analysis and generalize them
Syllabus

• Fundamentals
  Motivation, system organization, hardware execution models, limits to performance, program execution models, theoretical models

• Software and programming
  ‣ Programming models and techniques: multithreading and message passing
  ‣ Architectural considerations: multicore primarily
  ‣ C++11 threads and MPI

• Parallel algorithm design and implementation
  ‣ Case studies to develop a repertoire of problem solving techniques, enabling you to recognize the right implementation technique for the problem at hand
  ‣ Data structures and their efficient implementation: load balancing and performance
  ‣ Performance tradeoffs, evaluation, and tuning
The rest of the lecture

• Syllabus

• Intro to parallel computation
The progress of technological disruption

- **Transformational**: modelling, healthcare…
- **New capabilities**
- Changes the common wisdom for solving a problem including the implementation
Today’s mobile computer would have been yesterday’s supercomputer

• Cray-1 Supercomputer
• 80 MHz processor
• 240 Mflops/sec peak
• 3.4 Mflops Linpack
• 8 Megabytes memory

• Water cooled
• 1.8m H x 2.2m W
• 4 tons
• Over $10M in 1976

• iPad 4
• Apple A6X SoC
  • 1.4GHz dual-core Apple swift processor (4600 Megaflops)
  • Quad-core PowerVR SGX 554 GPU
• 988 Megabytes of memory, 32KB L1 and 1MB L2 caches
• 64GB Flash storage
• Color display
• Wireless or phone Networking
• Air cooled
• ~ 9.4 x 186 x 241 mm, 652 g
• $699 in September 2013

©2013 Scott B. Baden / CSE 160 / Fall 2013
The age of the multi-core processor

- On-chip parallel computer
- IBM Power4 (2001), Intel, AMD …
- First dual core laptops (2005-6)
- GPUs (nVidia, ATI):
  a supercomputer on a desktop
- Everyone has a parallel computer at their fingertips
- If we don’t use parallelism, we lose it!
Why is parallel computation inevitable?

- Physical limits on processor clock speed and heat dissipation
- A parallel computer increases memory capacity and bandwidth as well as the computational rate

Christopher Dyken, SINTEF

http://www.neowin.net/
What is parallel processing?

• Compute a workload on simultaneously executing physical resources
• Multiple processor cores co-operate to process a related set of tasks – tightly coupled
• Improve some aspect of performance
  ‣ Speedup: 100 processors run $\times 100$ faster than one
  ‣ Capability: Tackle a larger problem, more accurately
  ‣ Algorithmic, e.g. search
  ‣ Locality: more cache memory and bandwidth
Parallel Processing, Concurrency & Distributed Computing

• Parallel processing
  ‣ Performance (and capacity) is the goal
  ‣ Utilize the available resources effectively, avoiding high overheads

• Concurrency
  ‣ Correctness is the goal
  ‣ Ensure that shared resources are used appropriately, e.g. database transactions

• Distributed computation
  ‣ Geographically distributed
  ‣ Multiple resources computing & communicating unreliably
  ‣ “Cloud” or “Grid” computing, large amounts of storage
  ‣ Looser, coarser grained communication and synchronization
Have you written a parallel program?

- Threads
- MPI
- RPC
- C++11 Async
- CUDA
A Motivating Application - TeraShake

Simulates a 7.7 earthquake along the southern San Andreas fault near LA using seismic, geophysical, and other data from the Southern California Earthquake Center

epicenter.usc.edu/cmeportal/TeraShake.html
How TeraShake Works

• Divide up Southern California into blocks

• For each block, get all the data about geological structures, fault information, …

• Map the blocks onto processors of the supercomputer

• Run the simulation using current information on fault activity and on the physics of earthquakes

©2013 Scott B. Baden / CSE 160 / Fall 2013
Animation
The payoff

• Capability
  ‣ We solved a problem that we couldn’t solve before, or under conditions that were not possible previously

• Performance
  ‣ Solve the same problem in less time than before
  ‣ This can provide a capability if we are solving many problem instances

• The result achieved must justify the effort
  ‣ Enable new scientific discovery
  ‣ Software costs must be reasonable
Ok, we increased performance, but what’s the catch?

- If we don’t use the parallelism, we lose it
  - Amdahl’s law - serial sections
  - Von Neumann bottleneck
  - Load imbalances
- A well behaved single processor algorithm may behave poorly on a parallel computer, and may need to be reformulated
- There is no magic compiler that can turn a serial program into an efficient parallel program all the time and on all machines
What is involved?

• Performance programming
  ‣ Low-level details: heavily application dependent
  ‣ Irregularity in the computation and its data structures forces us to think even harder

• Parallelism introduces many new tradeoffs
  ‣ Redesign the software
  ‣ Rethink the problem solving technique
Remainder of today’s lecture

• Address space organization
  ‣ Shared memory
  ‣ Distributed memory

• Threads programming model
Address Space Organization

• We classify the address space organization of a parallel computer according to whether or not it provides global memory

• If there is global memory we have a “shared memory” or “shared address space” architecture
  ‣ multiprocessor vs partitioned global address space

• When there is no global memory, we have a “shared nothing” architecture, also known as a multicompiler
Multiprocessor organization

- Hardware automatically performs the global to local mapping using address translation mechanisms
- 2 types, depends on uniformity of memory access times
  - **UMA**: *Uniform* Memory Access time
    - Also called a Symmetric Multiprocessor (SMP)
  - **NUMA**: *Non-Uniform* Memory Access time
NUMA

• Non-Uniform Memory Access time
  ‣ Processors see distant-dependent access times to memory
  ‣ Implies physically distributed memory

• We often call these *distributed shared memory architectures*
  ‣ Commercial example: SGI Origin Altix, up to 512 cores
  ‣ Gordon system at San Diego Supercomputer Center
  ‣ Software/hardware support to monitor sharers
Architectures without shared memory

- A core has direct access to local memory only
- Send and receive messages to obtain copies of data from other nodes
- We call this a *shared nothing* architecture, or a *multicomputer*
- Similarity to NUMA
Parallel processing this course

• We will start by programming at the SMP level: multicore programming with threads
• We will then use multiple nodes: message passing
Today’s lecture

• Address space organization
  ‣ Shared memory
  ‣ Distributed memory

• Threads programming model
SPMD execution model

• Most parallel programming is implemented under the **Same Program Multiple Data** programming model = “SPMD”
  ‣ Threads
  ‣ Message passing
  ‣ Other names: “loosely synchronous” or “bulk synchronous”

• Programs execute as a set of P processes or threads
  ‣ We specify P when we run the program
  ‣ Each thread/process usually assigned to a different physical processor

• Each process or thread
  ‣ Is initialized with the same code
  ‣ Has an associated *index* or *rank*, a unique integer
  ‣ Executes instructions at its own rate

• Threads communicate through shared memory, processes via messages
Threads programming model

• Program executes a collection of independent instruction streams, called **threads**

• A thread is similar to a procedure call with notable differences
  ▶ A new storage class: shared data
  ▶ A procedure call is “synchronous:”
    a return indicates completion
  ▶ A spawned thread executes asynchronously until it completes
  ▶ Both share global storage with caller
  ▶ Synchronization may be needed when updating shared state (thread safety)
Why threads?

• Processes are “heavy weight” objects scheduled by the OS
  ‣ Protected address space, open files, and other state

• A thread AKA a lightweight process (LWP)
  ‣ Threads share the address space and open files of the parent, but have their own stack
  ‣ Reduced management overheads, e.g. thread creation
  ‣ Kernel scheduler multiplexes threads
Threads Programming model

- Start with a single root thread
- Fork-join parallelism to create concurrently executing threads
- Threads communicate via shared memory
- A spawned thread executes asynchronously until it completes
- Threads may or may not execute on different processors

©2013 Scott B. Baden / CSE 160 / Fall 2013
Multithreading in Practice

- C++11
- POSIX Threads “standard” (pthreads):
  IEEE POSIX 1003.1c-1995
    - Low level interface
    - Beware of non-standard features
- Java threads not used in high performance computation
- OpenMP – program annotations
- Parallel programming languages
  - Co-array FORTRAN
  - UPC
C++11 Threads

- Via `<thread>`, C++ supports a threading interface similar to pthreads, though a bit more user friendly
- Async is a higher level interface suitable for certain kinds of applications
- New memory model
- Atomic template
Hello world with <Threads>

```cpp
#include <thread>

void Hello(int TID) {
    cout << "Hello from thread " << TID << endl;
}

int main(int argc, char *argv[ ]) {
    thread *thrds = new thread[NT];

    // Spawn threads
    for(int t=0; t<NT; t++) {
        thrds[t] = thread(Hello, t);
    }

    // Join threads
    for(int t=0; t<NT; t++)
        thrds[t].join();
}
```

$ ./hello_th 3
Hello from thread 0
Hello from thread 1
Hello from thread 2

$ ./hello_th 3
Hello from thread 1
Hello from thread 0
Hello from thread 2

$ ./hello_th 4
Running with 4 threads
Hello from thread 0
Hello from thread 3
Hello from thread Hello from thread 21

$PUB/Examples//Threads/Hello-Th

PUB = /share/class/public/cse160-wi13
Fin