Lecture 1
Introduction
Introduction

• Your instructor is Scott B. Baden, 
  baden@cs.ucsd.edu

• Office: room 3244 in EBU3B

• Office hours: Tuesday after class (week 1) 
or by appointment

• The class home page is 
  http://www.cse.ucsd.edu/classes/sp07/cse262
Text and readings

• Assigned class readings will include handouts and on-line material

  On reserve at the library

• See our documentation web page at http://www-cse.ucsd.edu/users/baden/Doc

• Today’s lecture slides (after lecture) available at http://www.cse.ucsd.edu/classes/sp07/cse262/Lectures/Lec01.pdf
Course Requirements

• Class participation and preparation: 30%
  – Come prepared to contribute to class discussions
  – Writeups of the readings for 10 lectures
• In class presentations (2 x 30 minutes): 10%
• Research Project: 60%
  – Weekly progress reports
  – In class presentations
  – Report due Friday June 8, 2007 at 5pm
Policies

• Academic Integrity
  – 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/sp07/cse262/policies.html
Course overview and background

• Latest trends in solving computationally intensive problems on parallel computers
  – Historical retrospective
  – Trends
  – Software and hardware

• Background
  – Graduate standing
  – Recommend: computer architecture (CSE 240A)
  – Students outside CSE are welcome

• Prior experience
  – Parallel computation
  – Numerical analysis?
Topics

• A retrospective
  – Classic architectures and how to program them: Cray-1, Connection Machine 2, Cray T3E
  – Historical developments inevitably become relevant again
  – Foundational models: BSP, dataflow, PMH

• General purpose programming on GPUs
  – Architectures: Cell BE, NVIDIA etc
  – Software issues
  – Higher level run time models, language support

• Alternative programming models and languages
  – PGAS languages, Cilk
  – Architectural cognizance
  – Run time frameworks

• Programming in the large
  – Case studies to develop a repertoire of problem solving techniques, including irregular problems
  – Data structures and their efficient implementation: load balancing and performance
Hardware platforms

- SDSC DataStar and Blue Gene / L
- IBM Cell Broadband Engine
- NVIDIA ?
Today’s laptop would have been yesterday’s supercomputer

- Cray-1 Supercomputer
- 80 MHz processor
- 8 Megabytes memory
- Water cooled
- 6 feet H x 7 feet W
- 4 tons
- Over $10M in 1976

- MacBook
- 2.0GHz Intel Core Duo
- 1 Gigabyte memory
- Air cooled
- 1 x 14.1 x 9.6 inches
- 4 pounds
- $1299 in 2006
What can a parallel computer do for you?

• Solve a problem at a fraction of the cost of the traditional solution
• Enable new capabilities that were previously unachievable
• We “win” if the result justifies the effort
• Collaboration in computational chemistry
Motivation for parallel computers

• Potential for a large performance boost: 100 processors run 100 times faster than one processor
• Or, tackle a problem 100 time the size, accuracy…
• A parallel computer increases memory capacity and bandwidth as well as the computational rate
The Age of Parallel Computing

- Until the early to mid 90’s vector computer were the reigning architecture
- With colleagues, solved a problem on an Intel Paragon parallel computer at 1/6 the hardware cost of the Cray C90 supercomputer
- Clever algorithms played a role, avoiding costly “brute force” methods

![Graph showing Intel Paragon Performance relative to Cray C90]
Processing speeds

Computer Architecture: A Quantitative Approach
The processor-memory gap

![Graph showing the processor-memory gap from 1980 to 2005. The x-axis represents years (1980 to 2005) and the y-axis represents a logarithmic scale from $10^0$ to $10^5$. The graph shows an exponential increase in both processor and memory (DRAM) capacities over time. The processor capacity increases more rapidly than the memory capacity.](image-url)
Parallel processing

• We define parallel processing as the simultaneous execution or overlap of independent processing activities
• Different levels of granularity
• Granularity dictates how often a computation communicates, and what scale
  – Distributed computer: at the level of a whole program
  – Multicomputer: function, a loop nest
  – Multiprocessor: + memory reference (cache line)
  – Instruction level parallelism: instruction, register
A generic parallel computer

• A collection of $P$ processor-memory pairs
• Processors communicate over the interconnect
• We might not know what the interconnect looks like
• There is an option for global memory
Address Space Organization

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

• When there is a global memory we have a “shared memory” architecture, also known as a multiprocessor

• Where there is no global memory we have a “shared nothing” architecture, also known as a multicomputer
Issues

• A parallel computer increases memory capacity and bandwidth together with the computational rate

• **Scalability.** Performance increases as a “nice function” of the number of processors: linear or even $p \ lg \ p$

• In practice scalability can be hard to achieve
  ▶ “Non-productive” work associated with exploiting parallelism, e.g. communication
  ▶ Serial sections: portions of the code that run on only one processor. e.g. initialization
  ▶ Load imbalance: work assigned unevenly to processors

• Some algorithms present intrinsic barriers to realizing scalability and in these cases we seek alternatives

  ```
  \text{forall } i=0:n-1 \ \text{sum} = \text{sum} + x[i]
  ```

• Architecture and software issues, too
Amdahl’s law (1967)

- A serial section limits scalability
- Let $f =$ fraction of $T_1$ that runs serially
- *Amdahl's Law (1967)*: As $P \to \infty$, $S_P \to 1/f$
Scaled Speedup

• Amdahl’s law led many to take a pessimistic outlook on the benefits of parallelism

• Observation: Amdahl’s law assumes that the workload ($W$) remains fixed

• But parallel computers are used to tackle more ambitious workloads

  $W$ increases with $P$

  $f$ often decreases with $W$
Why is parallel programming interesting?

• A well behaved single processor algorithm may behave poorly on a parallel computer, and may need to be reformulated numerically.

• There is no magic compiler that can turn a serial program into an efficient parallel program all the time and on all machines.
  – Performance programming involving low-level details: heavily application dependent.
  – Irregularity in the computation and its data structures forces us to think even harder.
  – Users don’t start from scratch—they reuse old code. Poorly structured code, or code structured for older architectures can entail costly reprogramming.
An important universal: the locality principle

• Programs generally exhibit two forms of locality when accessing memory:
  – Temporal locality: we are more likely to reference more recently accessed memory locations than less recently accessed ones
  – Spatial locality: we are more likely to reference nearby addresses than far away ones

• Memory hierarchies rely on locality to improve memory access times through *re-use*

• Underlying principles for improving cache re-use also apply to local and remote data in a distributed memory computer

• Locality often involves loops in computationally intensive problems
Memory hierarchy pyramid

- **Disk**: 10^6 CP
- **Dram**: 100 to 500 CP
- **L2$**: 10 CP (10 to 100 B)
- **L1$**: 1 CP (10 to 100 B)

16 to 64 KB
256KB to 4 MB
100s of MB to a few GB
10s to 100s GB

Faster, smaller, more expensive
Shared memory organization

• The address space is global to all processors
• Hardware automatically performs the global to local mapping using address translation mechanisms
• More natural for the programmer and compiler writers
  – A natural extension for existing single processor execution model
  – We don’t have to distinguish local from remote data
• We classify shared memory architectures according to the uniformity of memory access times
UMA

• **UMA** = Uniform Memory Access time
• All processors observe the same access time to memory in the absence of contention
• Also called *Symmetric Multiprocessors*
• Usually bus based: not a scalable solution
NUMA: Non-Uniform Memory Access time

- Processors see distance-dependent access times to memory
- Implies physically distributed memory
- Distributed shared memory architectures
  - SGI Origin 3000, up to 512 processors
  - Elaborate interconnect monitors memory sharing
Disadvantages

• An efficient program may end up having to mimic a message passing program!

• For a given level of aggregate memory bandwidth, shared memory appears more costly than shared nothing architectures
Architectures without shared memory

• A collection of $P$ processor-memory pairs
• Processors communicate by sending messages over an interconnect
Hybrid organizations

- Today’s multicomputers generally have a hybrid design
- Hierarchically organized: each node is a multiprocessor
- Nodes communicate by passing messages, processors within a node communicate via shared memory
- Also called a *multi-tier computer*
Locality

• A parallel computer extends the notion of a traditional memory hierarchy

• The notions of locality which apply to virtual memory, cache memory, and registers, also apply to parallel memory hierarchies

• Vital to preserve locality inherent to the application, and to amortize fixed costs
Control mechanisms

• In addition to address space organization, architectures are also classified according to their *control mechanism*

• How do the processors issue instructions?

• Today, most parallel computers execute their instruction streams independently

• Some special purpose machines execute a global instruction stream in lock-step
Flynn’s classification (1966)

**SIMD:**
Single Instruction, Multiple Data

**MIMD:**
Multiple Instruction, Multiple Data
SIMD

- Two classic SIMD designs
  - ILIAC IV (1960s)
  - Connection Machine Model 1 and 2 (1980s)
- Modern designs: SSE, GPUs
- Efficiently compute on regular arrays of data
- Irregular or data dependent computations lead to poor performance

```
forall ( i=0:n-1)
  if ( x[i] < 0) then
    y[i] = x[i]
  else
    y[i] = √x[i]
  end if
```