CS 475: Parallel Programming Introduction

Sanjay Rajopadhye
(with updates by Wim Bohm, Cathie Olschanowski)
Colorado State University
Fall 2016

Name card info (inside)

- **Name:** Sanjay Rajopadhye
- **Pronunciation (optional):** In Indian names, "a" is almost always pronounced as a short "u" sound as in gun, fun, etc., or a long "aa" sound as in calm, bard, etc.
  - Sanjay is pronounced as Sun-juy
  - Rajopadhye Raaj-Oh-paath-yay (in the "paath" make the t sound like a d). Don't worry if you don't get it right, it's almost always mispronounced, even in India).
  - These pronunciation rules are used in many parts of Asia, e.g., pronounce "Bagdad?"
- **Major dept:** CS/ECE
- **Status:** Associate Professor (e.g., 3rd year Ph.D.)
- **Interesting fact:** I only recently discovered how thin my hair was in the back

Bio sketch

- **Education**
  - undergrad: B. Tech, IIT Kharagpur, India, 1980
  - grad: Ph.D. University of Utah, 1986
- **Professional:**
  - Assistant Prof, CS University of Oregon (86-91)
  - Assistant Prof, ECE Oregon State University (91-92)
  - Researcher/Prof Associé IRISA, Rennes (1993-2001)
  - Colorado State University (2001-…)
- **Research contributions/interests:**
  - High-performance computing, embedded systems, VLSI, algorithms, compilers, programming languages, …
  - Polyhedral model: a mathematical framework for describing, transforming and "compiling" massively parallel computations

Teaching Assistant

- **Name:** Swetha Varadarajan
- **What you want to be called:** Swetha
- **Major:** ECE
- **Status:** Grad (MS)
- **Interesting fact:** I can play candy crush for 15 hours straight

Topics

- Answer: Why parallel computing?
- Overview the types of parallel computing
- Overview of speedup and efficiency
- Review the plan for this class
- Interactive/Illustrative example

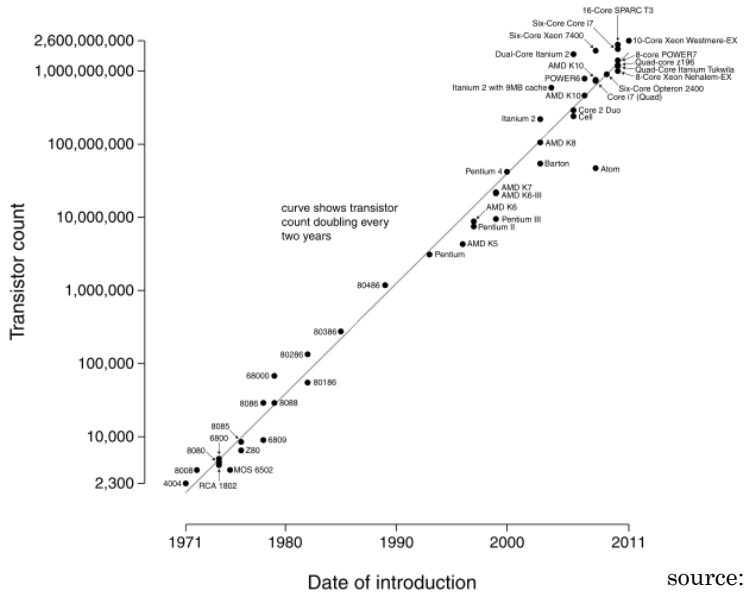
Why Parallel Programming

- Need for **speed**
  - Many applications require orders of magnitude more compute power than we have right now. Speed came for a long time from technology improvements, mainly increased clock speed and chip density.
  - The technology improvements have slowed down, and the only way to get more speed is to exploit **parallelism**. All new computers are now parallel computers.

Technology: Moore's Law

- Empirical observation (1965) by Gordon Moore:
  - The chip density: number of transistors that can be inexpensively placed on an integrated circuit, is increasing **exponentially**, doubling approximately every two years.
  - en.wikipedia.org/wiki/Moore's_law
- This has held true until now and is expected to hold until at least 2015 (news.cnet.com/New-life-for-Moores-Law/2009-1006_3-5672485.html).

Microprocessor Transistor Counts 1971-2011 & Moore's Law



source: Wikipedia

Corollary of exponential growth

- When two quantities grow exponentially, but at different rates, their ratio also grows exponentially.

  $y_1 = a^x$, and $y_2 = b^x$ for $a \geq b \geq 1$, *therefore* $\left(\frac{a}{b}\right)^x = r^x$, r ≥ 1

- $1.1^n \neq O(2^n)$
  - or 2^n grows a lot faster than $(1.1)^n$
- Consequence for computer architecture: growth rate for e.g. memory is not as high as for processors, therefore, memory gets slower and slower (in terms of clock cycles) as compared to processors.

  This gives rise to so called **gaps** or **walls**

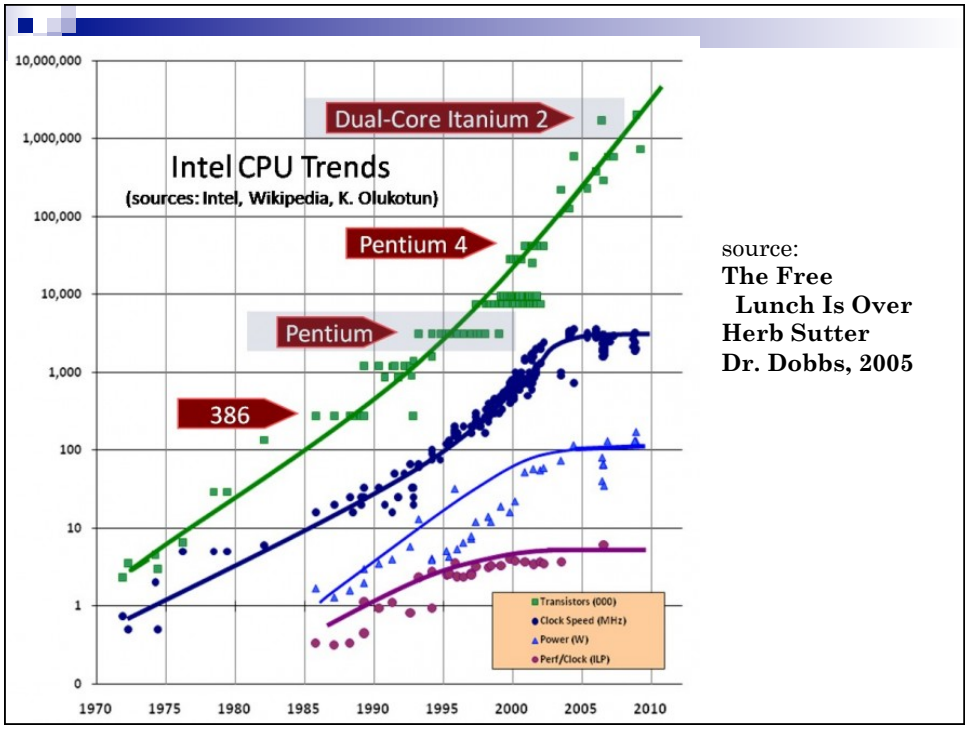
"Gaps or walls" of Moore's Law

- Memory gap/wall:

  Memory bandwidth and latency improve much slower than processor speeds (since mid 80s, this was addressed by ever increasing on-chip caches).
- (Brick) Power Wall

  Higher frequency means more power, i.e., more heat. So chips are getting exceedingly hot

  Consequence: We can no longer increase the clock frequency of our processors.



source:
The Free Lunch Is Over
Herb Sutter
Dr. Dobbs, 2005

Enter **Parallel Computing**

- The goal is to deliver **performance**. The only solution to the power wall is to have multiple processor cores on a chip, because we can still increase the chip **density**.
- Nowadays, there are no more processors with one core.
  - "The processor is the new transistor."
    - 2005 prediction: Number of cores will continue to grow at an exponential rate (at least until 2015), has not happened as much as we thought it would. We are still in the ~16 cores per micro-processor range.

Implicit Parallel Programming

- Let the compiler / runtime system detect parallelism, do task and data allocation, and scheduling.
- This has been a "holy grail" of compiler and parallel computing research for a long time and, after >40 years, still requires research on automatic parallelization, languages, OS, fault tolerance, etc.
- We still don't have a general purpose high performance implicit parallel programming language that compiles to a general purpose parallel machine.

Explicit Parallel Programming

Let the programmer express parallelism, task and data partitioning, allocation, synchronization, and scheduling, using programming languages extended with explicit parallel programming constructs.

- This makes the programming task harder, but a lot of fun and a large source of income
- (very large actually ☺)

Explicit parallel programming

- Explicit parallel programming
  - Multithreading: OpenMP
  - Message Passing: MPI
  - Data parallel programming: **CUDA**
- Explicit Parallelism complicates programming
  - creation, allocation, scheduling of processes
  - data partitioning
  - Synchronization (semaphores, locks, messages)

Computer Architectures

- Sequential: John von Neumann
  - Stored Program, single instruction stream, single data stream
- Parallel: Michael Flynn
  - SIMD:Single instruction multiple data
    - data parallel
  - MIMD: Multiple instruction multiple data

SIMD

- One control unit
- Many processing elements
  - All executing the same instruction
  - Local memories
  - Conditional execution
    - Where ocean perform ocean step
    - Where land perform land step
    - Gives rise to idle processors (optimized in GPUs)
- PE interconnection
  - Usually a 2D mesh

SIMD machines

- Bit level
  - ILLIAC IV (very early research machine, Illinois)
  - CM2 (Thinking Machines)
- now **GPU:** Cartesian grid of SMs
  (Streaming Multiprocessors)
  - each SM has a collection of Processing Elements
  - each SM has a **shared memory** and a register file
  - one **global memory**

MIMD

- Multiple Processors
  - Each executing its own code
- Distributed memory MIMD
  - Complete 'PE+Memory' connected to network
    - Cosmic Cube, N-Cube, our Cray
    - Extreme: Network of Workstations, Data centers: racks of processors with high speed bus interconnects
    - Programming model: Message Passing

Shared memory MIMD

- Shared memory
  - CPUs or cores, bus, shared memory
  - All our processors are now multi-core
  - Programming model: **OpenMP**
- NUMA: Non Uniform Memory Access
  - Memory Hierarchy (registers, cache, local, global)
  - Potential for better performance, but problem: memory (cache) coherence (resolved by architecture)

Clarifying Reading assignments

1. Inside Front Cover
2. Chapter 3 Parallel Algorithm Design

Speedup

- Why do we write parallel programs again?
  to get speedup: go faster than sequential
- What is speedup?
  T_1 = sequential time to execute a program
  sometimes called T, or S
  T_p = time to execute the same program with
  p processors (cores, PEs)
  $S_p = T_1 / T_p$ speedup for p processors

Ideal Speedup, linear speedup

- Ideal speedup: p fold speedup: **$S_p = p$**
  - Ideal not always possible. **WHY?**
    - Certain parts of the computation are inherently sequential
    - Tasks are data dependent, so not all processors are always busy, and need to synchronize
    - Remote data needs communication
      - Memory wall PLUS Communication wall
- Linear speedup: $S_p = \beta p$

  β is usually less than 1

Efficiency

- Speedup is usually not ideal, nor linear
- We express this in terms of efficiency E_p:

  $E_p = S_p / p$

  - E_p defines the average utilization of p processors
  - Range?
    - What does $E_p = 1$ signify?
    - What does $E_p = \beta$ ($0<\beta<1$) signify?

More realistic speedups

- $T_1 = 1, \quad T_p = \sigma + (o+\pi)/p$
- $S_p = 1 / (\sigma + (o+\pi)/p)$

  σ is sequential fraction of the program

  π is parallel fraction of the program

  $\sigma+\pi = 1$

  o is parallel overhead (does not occur in sequential execution)

Draw speedup curves for

p=1, 2, 4, 8, 16, 32 σ= ¼, ½ o= ¼, ½

When p goes to ∞, S_p goes to?

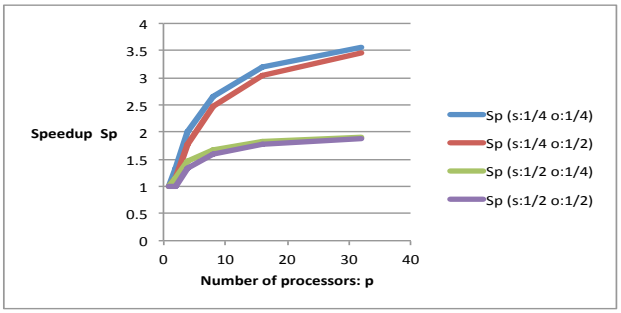
Amdahl's Law (from your reading)

Plotting speedups

$T_1 = 1, \quad T_p = \sigma + (o + \pi)/p$

$S_p = 1/(\sigma + (o + \pi)/p)$

p	Sp (s:1/4 o:1/4)	Sp (s:1/4 o:1/2)	Sp (s:1/2 o:1/4)	Sp (s:1/2 o:1/2)
1	1	1	1	1
2	1.33	1.14	1.14	1.00
4	2.00	1.78	1.45	1.33
8	2.67	2.46	1.68	1.60
16	3.2	3.05	1.83	1.78
32	3.56	3.46	1.91	1.88



This class: explicit parallel programming

Class Objectives

- Study the foundations of parallel programming
- Write explicit parallel programs
  - In C plus libraries
  - Using OpenMP (shared memory), MPI (message passing) and CUDA (data parallel)
  - Execute and measure, plot speedups, interpret / try to understand the outcomes (σ, π, o), document performance

Class Format

- Hands on: write efficient parallel code. This often starts with writing efficient sequential code.
- four versions of a code: naïve sequential, naïve parallel, smart sequential, smart parallel
- Lab covers many details, discussions follow up on what's learned in labs.
- PAs: Write effective and scalable parallel programs:
  - report and interpret their performance.

Parallel Programming steps

- Partition: break program in (smallest possible) parallel tasks and partition the data accordingly.
- Communicate: decide which tasks need to communicate (exchange data).
- Agglomerate: group tasks into larger tasks
  - reducing communication overhead
  - taking **LOCALITY** into account
- High Performance Computing is

  all about locality

Locality

A process (running on a processor) needs to have its data as close to it as possible. Processors have non uniform memory access (NUMA):

- registers 0 (extra) cycles
- cache 1,2,3 cycles

  there are multiple caches on a multicore chip
- local memory 50s to 100s of cycles
- global memory, processor-processor interconnect, disk …

Process

- COARSE in OS world: a program in execution
- FINER GRAIN in HPC world: (a number of) loop iterations, or a function invocation

Exercise: summing numbers

10x10 grid of 100 processors. Add up 4000 numbers initially at the corner processor[0,0], processors can only communicate with NEWS neighbors

Different cost models:

- Each communication "event" may have an arbitrarily large set of numbers (cost independent of volume).
  - Alternate: Communication cost is proportional to volume
- Cost: Communication takes 100x the time of adding
  - Alternate: communication and computation take the same time

Outline

- **"scatter"**: the numbers to processors row or column wise
  - Each processor along the edge gets a stack
    - Keeps whatever is needed for its row/column
    - Forwards the rest to the processor to south (or east)

      Initiates an independent "scatter" along its row/column
- each processor adds its local set of sums
- **"gather"**: sums are sent back in reverse order of scatter and added up along the way

Broadcast / Reduction

- One processor has a value, and we want every processor to have a copy of that value.
- Grid: Time for a processor to receive the value must be at least equal to its distance (number of hops in the grid graph) from the source.

  For the opposite corner, this is the perimeter of the grid, ~ 2sqrt(p) hops
- Reduction: dual of broadcast
  - First add n/p numbers on each processor
  - Then do the broadcast "in reverse" and add along the way

Analysis

- Scatter, add local, communicate back to the corner
  - Intersperse with one addition operation
  - problem size N=4000, machine size P=100, machine topology: square grid, comms cost C=100)
  - "Scatter:" 18 steps (get data, keep what you need, send the rest onwards) (2(sqrt(P)-1)))
  - Add 40 numbers
  - "Reduce" the local answers: 18 steps (one send, add 2 numbers)
- Is P=100 good for this problem? Which P is optimal, assuming a square grid)?

Weekly Roundup

- Moore's law (AKA why parallel?)
- SIMD vs MIMD
- 3 major types of parallel computing
- Explicit versus implicit parallelism
- Locality
- Calculation speedup
  - Measured
  - Theoretical ideal
  - Cost of communication