Parallel Computation

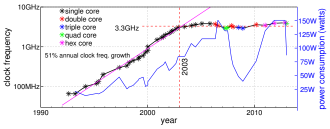
Mark Greenstreet

CpSc 418 – Jan. 4, 2017

Outline:

- Why Parallel Computation Matters
- Course Overview
- Our First Parallel Program
- The next month
- Table of Contents

Why Parallel Computation Matters



Clock Speed and Power of Intel Processors vs. Year Released
[Wikipedia CPU-Power, 2011]

- In the good-old days, processor performance doubled roughly every 1.5 years.
- Single thread performance has seen small gains in the past 14 years.
  - Too bad. If it had, we would have 1000GHz CPUs today. ☺
- Need other ways to increase performance.

Why Sequential Performance Can't Improve (much)

Power

- CPUs with faster clocks use more energy per operation than slower ones.
- For mobile devices: high power limits battery life.
- For desktop computers and gaming consoles: cooling high-power chips requires expensive hardware.
- For large servers and clouds, the power bill is a large part of the operating cost.

More Barriers to Sequential Performance

- The memory bottleneck.
  - Accessing main memory (i.e. DRAM) takes hundreds of clock cycles.
  - But, we can get high bandwidth.
- Limited instruction-level-parallelism.
  - CPUs already execute instructions in parallel.
  - But, the amount of this "free" parallelism is limited.
- Design complexity.
  - Designing a chip with 100 simple processors is **way** easier than designing a chip with one big processor.
- Reliability.
  - If a chip has 100 processors and one fails, there are still 99 good ones.
  - If a chip has 1 processor and it fails, then the chip is useless.
- See [Asanovic et al., 2006].

Parallel Computers

- Mobile devices:
  - multi-core to get good performance on apps and reasonable battery life.
  - many dedicated "accelerators" for graphics, WiFi, networking, video, audio, . . .
- Desktop computers
  - multi-core for performance
  - separate GPU for graphics
- Commercial servers
  - multiple, multi-core processors with shared memory.
  - large clusters of machines connected by dedicated networks.

Outline

- Why Does Parallel Computation Matter?
- Course Overview
  - Topics
  - Syllabus
  - The instructor and TAs
  - The textbook(s)
  - Grades

Homework:	35%	roughly one HW every two weeks
Midterm:	25%	March 1, in class
Final:	40%	
Mini-Assignments:	see description on slide 19	
Bug Bounties:	see description on slide 20	

  - Plagiarism – please don't
  - Learning Objectives
- Our First Parallel Program

Topics

- Parallel Architectures
- Parallel Performance
- Parallel Algorithms
- Parallel Programming Frameworks

Parallel Architectures

- **There isn't one, standard, parallel architecture for everything.**
  We have:
  - Multi-core CPUs with a shared-memory programming model. Used for mobile device application processors, laptops, desktops, and many large data-base servers.
  - Networked clusters, typically running linux. Used for web-servers and data-mining. Scientific supercomputers are typically huge clusters with dedicated, high-performance networks.
  - Domain specific processors
    - □ GPUs, video codecs, WiFi interfaces, image and sound processing, crypto engines, network packet filtering, and so on.
- As a consequence, **there isn't one, standard, parallel programming paradigm**.

Parallel Performance

The incentive for parallel computing is to do things that wouldn't be practical on a single processor.

- Performance matters.
- We need good models:
  - Counting operations can be very misleading – "adding is free."
  - Communication and coordination are often the dominant costs.
- We need to measure actual execution times of real programs.
  - There isn't a unified framework for parallel program performance analysis that works well in practice.
  - It's important to measure actual execution time and identify where the bottlenecks are.
- Key concepts with performance:
  - Amdahl's law, linear speed up, overheads.

Parallel Algorithms

- We'll explore some old friends in a parallel context:
  - Sum of the elements of an array
  - matrix multiplication
  - dynamic programming.
- And we'll explore some uniquely parallel algorithms:
  - Bitonic sort
  - mutual exclusion
  - producer consumer

Parallel Programming Frameworks

- Erlang: functional, message passing parallelism
  - Avoids many of the common parallel programming errors: races and side-effects.
    You can write Erlang programs with such bugs, but it takes extra effort (esp. for the examples we consider).
  - Allows a simple presentation of many ideas.
  - But it's slow, for many applications, when compared with C or C++.
  - OTOH, it finds real use in large-scale distributed systems.
- CUDA: your graphics card is a super-computer
  - Excellent performance on the "right" kind of problem.
  - The data-parallel model is simple, and useful.

Syllabus

- January: Erlang
  Jan. 4– 9: Course overview, intro. to Erlang programming.
  Jan. 11–18: Parallel programming in Erlang, reduce and scan.
  Jan. 20–27: Parallel architectures
  Jan. 29–Feb. 6: Performance analysis
- February: Erlang, Midterm
  Feb. 8–17: Sorting
  Feb. 20–19: Midterm break.
  Feb. 27: Midterm Review
  Mar. 1: Midterm
- March: CUDA and other topics
  Mar. 3–10: Introduction to SIMD and CUDA.
  Mar. 13–24: More algorithms in CUDA (and a bit of Erlang)
  Mar. 27–Apr. 6: Map-Reduce, Mutual Exclusion, & More Fun.
- Note: I'll make adjustments to this schedule as we go.

Administrative Stuff – Who

- The instructors
  - **Mark Greenstreet,** mrg@cs.ubc.ca
    - □ ICICS/CS 323, (604) 822-3065
    - □ Office hours: Tuesdays, 1pm – 2:30pm, ICICS/CS 323
  - **Ian Mitchell,** mitchell@cs.ubc.ca
    - □ ICICS/CS 217, (604) 822-2317
    - □ Office hours: TBD
- The TAs

Devon Graham,	drgraham@cs.ubc.ca
Chenxi Liu,	chenxil@cs.ubc.ca
Carolyn Shen,	shen.carolyn@gmail.com
Brenda Xiong,	krx.sky@gmail.com

- Course webpage: http://www.ugrad.cs.ubc.ca/~cs418.
- Online discussion group: on piazza.

Textbook(s)

- For Erlang: *Learn You Some Erlang For Great Good*, Fred Hébert,
  - Free! On-line at http://learnyousomeerlang.com.
  - You can buy the dead-tree edition at the same web-site if you like.
- For CUDA: *Programming Massively Parallel Processors: A Hands-on Approach* (2nd **or** 3rd ed.), D.B. Kirk and W-M.W. Hwu.
  - Please get a copy by late February – I'll assign readings starting after the midterm. It's available at amazon.ca and many other places.
- I'll hand-out copies of some book chapters:
  - *Principles of Parallel Programming* (chap. 5), C. Lin & L. Snyder – for the reduce and scan algorithms.
  - *An Introduction to Parallel Programming* (chap. 2), P.S. Pacheco – for a survey of parallel architectures.
  - Probably a few journal, magazine, or conference papers.

Why so many texts?

- There isn't one, dominant parallel architecture or programming paradigm.
- The Lin & Snyder book is a great, paradigm independent introduction,
- But, I've found that descriptions of real programming frameworks lack the details that help you write real code.
- So, I'm using several texts, but
  - You only have to buy one! ☺

Grades

Homework:	35%	roughly one assignment every two weeks
Midterm:	25%	March 1, in class
Final:	40%	
Mini-Assignments:	see description on slide 19	
Bug Bounties:	see description on slide 20	

Homework

- Collaboration policy
  - You are welcome and encouraged to discuss the homework problems with other students in the class, with the TAs and me, and find relevant material in the text book, other books, on the web, etc.
  - You are expected to work out your own solutions and write your own code. Discussions as described above are to help understand the material. Your solutions must be your own.
  - You must properly cite your collaborators and any outside sources that you used. You don't need to cite material from class, the textbooks, or meeting with the TAs or instructor. See slide 22 for more on the plagiarism policy.
- Late policy
  - Each assignment has an "early bird" date before the main date. Turn in you assignment by the early-bird date to get a 5% bonus.
  - **No late homework accepted.**

Exams

- Midterm, in class, on March 1.
- Final exam will be scheduled by the registrar.
- Both exams are open book, open notes, open homework and solutions – open anything printed on paper.
  - You can bring a calculator.
  - No communication devices: laptops, tablets, cell-phones, etc.

Mini-Assignments

- Mini-assignments
  - Worth 20% of points missed from HW and exams.
    - □ If your raw grade is 90%, you can get at most 2% from the minis. Missing one or two isn't a big deal.
    - □ If your raw grade is 70%, you can get 6% from the minis. This can move your letter grade up a notch (e.g. C+ to B−).
    - □ If your raw grade is 45%, you can get up to 11% from the minis. Do the mini-assignments – I hate turning in failing grades.
  - The first is at
    http://www.ugrad.cs.ubc.ca/~cs418/2016-2/mini/1/mini1.pdf,
    and due Jan. 9.
  - **If you are on the course waitlist,** we will select from the students who submit acceptable solutions to **Mini Assignment 1** to fill any slots that open up.

Bug Bounties

- If I make a mistake when stating a homework problem, then the first person to report the error gets extra credit.
  - If the error would have prevented solving the problem, then the extra credit is the same as the value of the problem.
  - Smaller errors get extra credit in proportion to their severity.
- Likewise, bug bounties are awarded (as homework extra credit) for finding errors in mini-assignments, lecture slides, the course web-pages, code I provide, etc.
- The midterm and final have bug bounties awarded in midterm and final exam points respectively.
- If you find an error, report it.
  - Suspected errors in homework, lecture notes, and other course materials should be posted to piazza.
  - The first person to post a bug gets the bounty.
  - Bug-bounties reward you for looking at the HW when it first comes out, and not waiting until the day before it is due.

Grades: the big picture

$$\begin{aligned}
\mathit{RawGrade} &= 0.35 * \mathit{HW} + 0.25 * \mathit{MidTerm} + 0.40 * \mathit{Final}\\
\mathit{MiniBonus} &= 0.20 * (1 - \min(\mathit{RawGrade}, 1)) * \mathit{Mini}\\
\mathit{BB} &= 0.35 * \mathit{BB}_{\mathit{HW}} + 0.25 * \mathit{BB}_{\mathit{MT}} + 0.40 * \mathit{BB}_{\mathit{FX}}\\
\mathit{CourseGrade} &= \min(\mathit{RawGrade} + \mathit{MiniBonus} + \mathit{BB}, 1) \times 100\%
\end{aligned}$$

Plagiarism

- I have a very simple criterion for plagiarism:
  Submitting the work of another person, whether that be another student, something from a book, or something off the web and representing it as your own is plagiarism and constitutes academic misconduct.
- If the source is clearly cited, then it is not academic misconduct.
  If you tell me "This is copied word for word from Jane Foo's solution" that is not academic misconduct. It will be graded as one solution for two people and each will get half credit. I guess that you could try telling me how much credit each of you should get, but I've never had anyone try this before.
- I encourage you to discuss the homework problems with each other.
  If you're brainstorming with some friends and the key idea for a solution comes up, that's OK. In this case, add a note to your solution that lists who you collaborated with.
- More details at:
  - http://www.ugrad.cs.ubc.ca/~cs418/plagiarism.html
  - http://learningcommons.ubc.ca/guide-to-academic-integrity/

Learning Objectives (1/2)

- Parallel Algorithms
  - Familiar with parallel patterns such as reduce, scan, and tiling and can apply them to common parallel programming problems.
  - Can describe parallel algorithms for matrix operations, sorting, dynamic programming, and process coordination.
- Parallel Architectures
  - Can describe shared-memory, message-passing, and SIMD architectures.
  - Can describe a simple cache-coherence protocol.
  - Can identify how communication latency and bandwidth are limited by physical constraints in these architectures.
  - Can describe the difference between bandwidth and inverse latency, and how these impact parallel architectures.

Learning Objectives (2/2)

- Parallel Performance
  - Understands the concept of "speed-up": can calculate it from simple execution models or measured execution times.
  - Can identify key bottlenecks for parallel program performance including communication latency and bandwidth, synchronization overhead, and intrinsically sequential code.
- Parallel Programming Frameworks
  - Can implement simple parallel programs in Erlang and CUDA.
  - Can describe the differences between these paradigms.
  - Can identify when one of these paradigms is particularly well-suited (or badly suited) for a particular application.

Lecture Outline

- Why Does Parallel Computation Matter?
- Course Overview
- Our First Parallel Program
  - Erlang quick start
  - Count 3s
  - Counting 3's in parallel
    - □ The root process
    - □ Spawning worker processes
    - □ The worker processes
    - □ Running the code

Erlang Intro – very abbreviated!

- Erlang is a functional language:
  - Variables are given values when declared, and the value never changes.
  - The main data structures are lists, `[Head | Tail]`, and tuples (covered later).
  - Extensive use of pattern matching.
- The source code for the examples in this lecture is available at:
  http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-06/code.html

Lists

- `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]` is a list of 10 elements.
- If `L1` is a list, then `[0 | L1]` is the list obtained by prepending the element `0` to the list `L1`. In more detail:

```
1> L1 = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
2> L2 = [0 | L1].
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
3> L3 = [0 , L1].
[0, [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]]
```

- Of course, we traverse a list by using recursive functions:

Lists traversal example: sum

```
sum(List) ->
   if (length(List) == 0) -> 0;
      (length(List) > 0)  -> hd(List) + sum(tl(List))
   end.
```

- `length(L)` returns the number of elements in list `L`.
- `hd(L)` returns the first element of list `L` (the head), and throws an exception if `L` is the empty list.
  `hd([1, 2, 3]) = 1`. `hd([1]) = 1` as well.
- `tl(L)` returns the list of all elements after the first (the tail).
  `tl([1, 2, 3]) = [2, 3]`. `tl([1]) = []`.
- See `sum_wo_pm` ("sum without pattern matching") in simple.erl

Pattern Matching – first example

We can use Erlang's pattern matching instead of the `if` expression:

```
sum([]) -> 0;
sum([Head | Tail]) -> Head + sum(Tail).
```

- `sum([Head | Tail])` matches any non-empty list with `Head` being bound to the value of the first element of the list, and `Tail` begin bound to the list of all the other elements.
- More generally, we can use patterns to identify the different cases for a function.
- This can lead to very simple code where function definitions follow the structure of their arguments.
- See `sum` in simple.erl

Count 3's: a simple example

Given an array (or list) with `N` items, return the number of those elements that have the value `3`.

```
count3s([]) -> 0;
count3s([3 | Tail]) -> 1 + count3s(Tail);
count3s([_Other | Tail]) -> count3s(Tail).
```

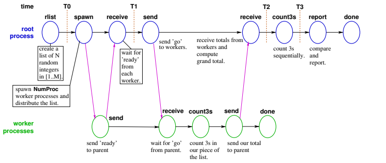
- We'll need to put the code in an erlang module. See `count3s` in count3s.erl for the details.
- To generate a list of random integers, count3s.erl uses the function `rlist(N, M)` from course Erlang library that returns a list of `N` integers randomly chosen from `1..M`.

Running Erlang

```
bash-3.2$ erl
Erlang/OTP 18 [erts-7.0] [source] ...
Eshell V7.0 (abort with ^G)

1> c(count3s).
{ok,count3s}
2> L20 = count3s:rlist(20,5).
[3,4,5,3,2,3,5,4,3,3,1,2,4,1,3,2,3,3,1,3]
3> count3s:count3s(L20).
9
4> count3s:count3s(count3s:rlist(1000000,10)).
99961
5> q().
ok
6> bash-3.2$
```

A Parallel Version


time
T0
T1
T2
T3
root process
rlist
spawn
receive
send
receive
count3s
report
done
create a list of N random integers in [1..M].
spawn NumProc worker processes and distribute the list.
wait for 'ready' from each worker.
send 'go' to workers.
receive totals from workers and compute grand total.
count 3s sequentially.
compare and report.
worker processes
send
receive
count3s
send
done
send 'ready' to parent
wait for 'go' from parent.
count 3s in our piece of the list.
send our total to parent

The code is in

http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-04/src/count3s.erl

Preview of the next month

January 6: Introduction to Erlang Programming
Reading: *Learn You Some Erlang*, the first eight sections – Introduction through Recursion. Feel free to skip the stuff on bit syntax and binary comprehensions.

January 9: Processes and Messages
Reading: *Learn You Some Erlang*, Higher Order Functions and The Hitchhiker's Guide... through More on Multprocessing
Homework: **Homework 1 goes out (due Jan. 18)** – Erlang programming
Mini-Assignment: Mini-Assignment 1 due **10:00am**
Mini-Assignment 2 goes out (due Jan. 13)

January 11: Reduce
Reading: *Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures

January 13: Scan
Reading: Lin & Snyder, chapter 5, pp. 112–125
Mini-Assignment: Mini-Assignment 2 due **10:00am**

January 16: Generalized Reduce and Scan
Homework: Homework 1 deadline for early-bird bonus (11:59pm)
Homework 2 goes out (due Feb. 1) – Reduce and Scan

January 18: Reduce and Scan Examples
Homework: Homework 1 due **11:59pm**

January 20–27: Parallel Architecture

January 29–February 6: Parallel Performance

Review Questions

- Name one, or a few, key reasons that parallel programming is moving into mainstream applications.
- How does the impact of your mini assignment total on your final grade depend on how you did on the other parts of the class?
- What are bug-bounties?
- What is the count 3's problem?
- How did we measure running times to compute speed up?
  - Why did one approach show a speed-up greater than the number of cores used?
  - Why did the other approach show that the parallel version was slower than the sequential one?

Supplementary Material

- Erlang Resources
- Bibliography
- Table of Contents – at the end!!!

Erlang Resources

- Learn You Some Erlang
  http://learnyousomeerlang.com
  An on-line book that gives a very good introduction to Erlang. It has great answers to the "Why is Erlang this way?" kinds of questions, and it gives realistic assessments of both the strengths and limitations of Erlang.
- Erlang Examples:
  http://www.ugrad.cs.ubc.ca/~cs418/2012-1/lecture/09-08.pdf
  My lecture notes that walk through the main features of Erlang with examples for each. Try it with an Erlang interpretter running in another window so you can try the examples and make up your own as you go. This will cover everything you'll need to make it through all (or most) of what we'll do in class, but it doesn't explain how to think in Erlang as well as "Learn You Some Erlang" or Armstrong's Erlang book (next slide).

More Erlang Resources

- The erlang.org tutorial
  http://www.erlang.org/doc/getting_started/users_guide.html
  Somewhere between my "Erlang Examples" and "Learn You Some Erlang."
- Erlang Language Manual
  http://www.erlang.org/doc/reference_manual/users_guide.html
  My go-to place when looking up details of Erlang operators, etc.
- On-line API documentation:
  http://www.erlang.org/erldoc.
- The book: *Programming Erlang: Software for a Concurrent World*, Joe Armstrong, 2007,
  http://pragprog.com/book/jaerlang/programming-erlang
  Very well written, with lots of great examples. More than you'll need for this class, but great if you find yourself using Erlang for a big project.
- More resources listed at http://www.erlang.org/doc.html.

Getting Erlang

- You can run Erlang by giving the command `erl` on any departmental machine. For example:
  - Linux: bowen, thetis, lin01, ..., lin25, ...,

  all machines above are .ugrad.cs.ubc.ca, e.g. bowen.ugrad.cs.ubc.ca, etc.
- You can install Erlang on your computer
  - Erlang solutions provides packages for Windows, OSX, and the most common linux distros
    https://www.erlang-solutions.com/resources/download.html
  - Note: some linux distros come with Erlang pre-installed, but it might be an old version. You should probably install from the link above.

Starting Erlang

- Start the Erlang interpretter.

```
theis % erl
Erlang/OTP 18 [erts-7.0] [source] ...
Eshell V7.0 (abort with ^G)

1> 2+3.
5
2>
```

- The Erlang interpreter evaluates expressions that you type.
- Expressions end with a "`.`" (period).

Bibliography

Krste Asanovic, Ras Bodik, et al.
The landscape of parallel computing research: A view from Berkeley.
Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Science Department, University of California, Berkeley, December 2006.
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

Microprocessor quick reference guide.
http://www.intel.com/pressroom/kits/quickrefyr.htm, June 2013.
accessed 29 August 2013.

List of CPU power dissipation.
http://en.wikipedia.org/wiki/List_of_CPU_power_dissipation, April 2011.
accessed 26 July 2011.

Table Of Contents (1/2)

Table Of Contents (2/2)

Introduction to Erlang

Mark Greenstreet

CpSc 418 – January 6, 2016

Outline:

- Erlang Basics
- Functional programming
- Example, sorting a list
- Functions
- Supplementary Material
- Table of Contents

Objectives

- Learn/review key concepts of functional programming:
  - Referential transparency.
  - Structuring code with functions.
- Introduction to Erlang
  - Basic data types and operations.
  - Program design by structural decomposition.
  - Writing and compiling an Erlang module.

Erlang Basics

- Numbers:
  - Numerical Constants: `1`, `8#31`, `1.5`, `1.5e3`, but not: `1.` or `.5`
  - Arithmetic: `+`, `-`, `*`, `/`, `div`, `band`, `bor`, `bnot`, `bsl`, `bsr`, `bxor`
- Booleans:
  - Comparisons: `=:=`, `=/=`, `==`, `/=`, `<`, `=<`, `>`, `>=`
  - Boolean operations (strict): `and`, `or`, `not`, `xor`
  - Boolean operations (short-circuit): `andalso`, `orelse`
- Atoms:
  - Constants: `x`, `'big DOG-2'`
  - Operations: tests for equality and inequality. Therefore pattern matching.

Lists and Tuples

- Lists:
  - Construction: `[1, 2, 3]`, `[Element1, Element2, ..., Element_N | Tail]`
  - Operations: `hd`, `tl`, `length`, `++`, `--`
  - Erlang's list library, http://erlang.org/doc/man/lists.html: `all`, `any`, `filter`, `foldl`, `foldr`, `map`, `nth`, `nthtail`, `seq`, `sort`, `split`, `zipwith`, and **many** more.
- tuples:
  - Construction: `{1, dog, "called Rover"}`
  - Operations: `element`, `setelement`, `tuple_size`.
  - Lists vs. Tuples:
    - ★ **Lists** are typically used for an **arbitrary** number of elements of the same "type" – like arrays in C, Java, ...
    - ★ **Tuples** are typically used for an **fixed** number of elements of the varying "types" – likes a `struct` in C or an object in Java.

Strings

What happened to strings?!

- Well, they're lists of integers.
- This can be annoying. For example,

```
1> [102, 111, 111, 32, 98, 97, 114].
"foo bar"
2>
```

- By default, Erlang prints lists of integers as strings if every integer in the list is the ASCII code for a "printable" character.
- *Learn You Some Erlang* discusses strings in the "Don't drink too much Kool-Aid" box for lists.

Functional Programming

- Imperative programming (C, Java, Python, ...) is a programming model that corresponds to the von Neumann computer:
  - A program is a sequence of statements.
    In other words, a program is a recipe that gives a step-by-step description of what to do to produce the desired result.
  - Typically, the operations of imperative languages correspond to common machine instructions.
  - Control-flow (`if`, `for`, `while`, function calls, etc.)
    Each control-flow construct can be implemented using branch, jump, and call instructions.
  - This correspondence program operations and machine instructions simplifies implementing a good compiler.
- Functional programming (Erlang, lisp, scheme, Haskell, ML, ...) is a programming model that corresponds to mathematical definitions.
  - A program is a collection of definitions.
  - These include definitions of expressions.
  - Expressions can be evaluated to produce results.
- See also: the *LYSE* explanation.

Erlang Makes Parallel Programming Easier

- Erlang is functional
  - Each variable gets its value when it's declared – it **never** changes.
  - Erlang eliminates many kinds of races – another process **can't** change the value of a variable while you're using it, because the values of variables never change.
- Erlang uses message passing
  - Interactions between processes are under explicit control of the programmer.
  - Fewer races, synchronization errors, etc.
- Erlang has simple mechanisms for process creation and communication
  - The structure of the program is not buried in a large number of calls to a complicated API.

Big picture: Erlang makes the issues of parallelism in parallel programs more apparent and makes it easier to avoid many common pitfalls in parallel programming.

Referential Transparency

- This notion that a variable gets a value when it is declared and that the value of the variable never changes is called **referential transparency**.
  - You'll here me use the term many times in class – I thought it would be a good idea to let you know what it means. ☺
- We say that the value of the variable is **bound** to the variable.
- Variables in functional programming are much like those in mathematical formulas:
  - If a variable appears multiple places in a mathematical formula, we assume that it has the same value everywhere.
  - This is the same in a functional program.
  - This is not the case in an imperative program. We can declare `x` on line 17; assign it a value on line 20; and assign it another value on line 42.
  - The value of `x` when executing line 21 is different than when executing line 43.

Loops violate referential transparency

```
// vector dot-product
sum = 0.0;
for(i = 0; i < a.length; i++)
   sum = a[i] * b[i];
```

```
// merge, as in merge-sort
while(a != null && b != null) {
   if(a.key <= b.key) {
      last->next = a;
      last = a;
      a = a->next;
      last->next = null;
   } else {
      ...
   }
}
```

- Loops rely on changing the values of variables.
- Functional programs use recursion instead.
- See also the *LYSE* explanation.

Life without loops

Use recursive functions instead of loops.

```
dotProd([], []) -> 0;
dotProd([A | Atl], [B | Btl]) -> A*B + dotProd(Atl, Btl).
```

- Functional programs use recursion instead of iteration:

```
dotProd([], []) -> 0;
dotProd([A | Atl], [B | Btl]) -> A*B + dotProd(Atl, Btl).
```

- Anything you can do with iteration can be done with recursion.
  - But the converse is not true (without dynamically allocating data structures).
  - Example: tree traversal.

Example: Sorting a List

- The simple cases:
  - Sorting an empty list: `sort([]) ->` _________
  - Sorting a singleton list: `sort([A]) ->` _________
- How about a list with more than two elements?
  - Merge sort?
  - Quick sort?
  - Bubble sort (NO WAY! Bubble sort is DISGUSTING!!!).
- Let's figure it out.

Merge sort: Erlang code

- If a list has more than one element:
  - Divide the elements of the list into two lists of roughly equal length.
  - Sort each of the lists.
  - Merge the sorted list.
- In Erlang:

```
sort([]) -> [];
sort([A]) -> [A];
sort([A | Tail]) ->
   {L1, L2} = split([A | Tail]),
   L1_sorted = sort(L1),
   L2_sorted = sort(L2),
   merge(L1_sorted, L2_sorted).
```

- Now, we just need to write `split`, and `merge`.

`split(L)`

Identify the cases and their return values according to the shape of `L`:

```
% If L is empty (recall that split returns a tuple of two lists):
split([]) -> {          ,          }

% If L

split(          ) ->

% If L
```

Finishing merge sort

- An exercise for the reader – see slide 29.
- Sketch:
  - Write `merge(List1, List2) -> List12` – see slide 30
  - Write an Erlang modle with the `sort`, `split`, and `merge` functions – see slide 31
  - Run the code – see slide 33

Fun with functions

- Programming with patterns
  - often, the code just matches the shape of the data
  - like CPSC 110, but pattern matching makes it obvious
  - see slide 16
- Fun expressions
  - in-line function definitions
  - see slide 17
- Higher-order functions
  - encode common control-flow patterns
  - see slide 18
- List comprehensions
  - common operations on lists
  - see slide 19
- Tail call elimination
  - makes recursion as fast as iteration (in simple cases)
  - see slide 20

Programming with Patterns

```
% leafCount: count the number of leaves of a tree represented by a nested list
leafCount([]) -> 0; % base case – an empty list/tree has no leaves

leafCount([Head | Tail]) -> % recursive case
   leafCount(Head) + leafCount(Tail);
leafCount(_Leaf) -> 1; % the other base case – _Leaf is not a list
```

- Let's try it

```
2> examples:leafCount([1, 2, [3, 4, []], [5, [6, banana]]]).
7
```

- Notice how we used **patterns** to show the how the recursive structure of `leafCount` follows the shape of the tree.
- See Pattern Matching in *Learn You Some Erlang* for more explanation and examples.
- Style guideline: if you're writing code with lots of `if`'s `hd`'s, and `tl`'s, you should think about it and see if using patterns will make your code simpler and clearer.

Anonymous Functions

```
3> fun(X, Y) -> X*X + Y*Y end.  % fun ...  end creates an "anonymous function"
#Fun<erl_eval.12.52032458> % ok, I guess, but what can I do with it?!
4> F = fun(X, Y) -> X*X + Y*Y end.
#Fun<erl_eval.12.52032458>
5> F(3, 4).
25
6> Factorial = % We can even write recursive fun expressions!
     fun Fact(0) -> 1;
         Fact(N) when is_integer(N), N > 0 => N*Fact(N-1)
     end.
7> Factorial(3).
6
8> Fact(3).
* 1:  variable 'Fact' is unbound
9> Factorial(-2).
** exception error:  no function clause matching
       erl_eval:'-inside-an-interpreted-fun-'(-2)
10> Factorial(frog).
** exception error:  no function clause matching
       erl_eval:'-inside-an-interpreted-fun-'(frog)
```

See Anonymous Functions in *Learn You Some Erlang* for more explanation and examples.

Higher-Order Functions

- `lists:map(Fun, List)` apply `Fun` to each element of `List` and return the resulting list.

```
11> lists:map(fun(X) -> 2*X+1 end, [1, 2, 3]).
[3, 5, 7]
```

- `lists:fold(Fun, Acc0, List)` use `Fun` to combine all of the elements of codeList in left-to-right order, starting with `Acc0`.

```
12> lists:foldl(fun(X, Y) -> X+Y end, 100, [1, 2, 3]).
106
```

- For more explanation and examples:
  - See Higher Order Functions in *Learn You Some Erlang*.
  - See the lists module in the Erlang standard library. Examples include
    - `all(Pred, List)`: true iff `Pred` evaluates to true for **every** element of `List`.
    - `any(Pred, List)`: true iff `Pred` evaluates to true for **any** element of `List`.
    - `foldr(Fun, Acc0, List)`: like `foldl` but combines elements in right-to-left order.

List Comprehensions

- Map and filter are such common operations, that Erlang has a simple syntax for such operations.
- It's called a **List Comprehension**:
  - `[`*Expr* `||` *Var* `<-` *List*, *Cond*, `...]`.
  - *Expr* is evaluated with *Var* set to each element of *List* that satisfies *Cond*.
  - Example:

```
13>R = count3s:rlist(5, 1000).
[444,724,946,502,312].
14>[X*X || X <- R, X rem 3 == 0].
[197136,97344].
```

- See also List Comprehensions in *LYSE*.

Head vs. Tail Recursion

- I wrote two versions of computing the sum of the first `N` natural numbers:

```
sum_h(0) -> 0; % "head recursive"
sum_h(N) -> N + sum_h(N-1).

sum_t(N) -> sum_t(N, 0).
sum_t(0, Acc) -> Acc; % "tail recursive"
sum_t(N, Acc) -> sum_t(N-1, N+Acc).
```

- Here are some run times that I measured:

N	t_{head}	t_{tail}	N	t_{head}	t_{tail}
1K	$21\mu s$	$13\mu s$	1M	21ms	11ms
10K	$178\mu s$	$114\mu s$	10M	1.7s	115ms
100K	1.7ms	1.1ms	100M	28s	1.16s
			1G	$>$ 8 min	11.6s

Head vs. Tail Recursion – Comparison

- Both grow linearly for $N \leq 10^6$.
  - The tail recursive version has runtimes about 2/3 of the head-recursive version.
- For $N > 10^6$,
  - The tail recursive version continues to have run-time linear in `N`.
  - The head recursive version becomes much slower than the tail recursive version.
- The Erlang compiler optimizes tail calls
  - When the last operation of a function is to call another function, the compiler just revises the current stack frame and jumps to the entry point of the callee.
  - The compiler has turned the recursive function into a while-loop.
  - Conclusion: **When people tell you that recursion is slower than iteration – don't believe them.**
- The head recursive version creates a new stack frame for each recursive call.
  - I was hoping to run my laptop out of memory and crash the Erlang runtime – makes a fun, in-class demo.
  - But, OSX does memory compression. All of those repeated stack frames are very compressible. The code doesn't crash, but it's very slow.

Tail Call Elimination – a few more notes

- I doubt we'll have time for this in lecture. I've included it here for completeness.
- Can you count on your compiler doing tail call elimination:
  - In Erlang, the compiler is **required** to perform tail-call elimination. We'll see why on Monday.
  - In Java, the compiler is forbidden from performing tail-call elimination. This is because the Java security model involves looking back up the call stack.
  - `gcc` performs tail-call elimination when the `-o` flag is used.
- Is it OK to write head recursive functions?
  - Yes! Often, the head-recursive version is much simpler and easier to read. If you are confident that it won't have to recurse for millions of calls, then write the clearer code.
  - Yes! Not all recursive functions can be converted to tail-recursion.
    - Example: tree traversal.
    - Computations that can be written as "loops" in other languages have tail-recursive equivalents.
    - But, recursion is more expressive than iteration.

Summary

- Why Erlang?
  - Functional – avoid complications of side-effects when dealing with concurrency.
  - But, we can't use imperative control flow constructions (e.g. loops).
    - Design by declaration: look at the structure of the data.
    - More techniques coming in upcoming lectures.
- Sequential Erlang
  - Lists, tuple, atoms, expressions
  - Using structural design to write functions: example sorting.
  - Functions: patterns, higher-order functions, head vs. tail recursion.

Preview

January 9: Processes and Messages
Reading: *Learn You Some Erlang*, Higher Order Functions and The Hitchhiker's Guide. . . through More on Multprocessing
Homework: **Homework 1 goes out (due Jan. 18)** – Erlang programming
Mini-Assignment: Mini-Assignment 1 due **10:00am**
Mini-Assignment 2 goes out (due Jan. 13)

January 11: Reduce
Reading: *Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures

January 13: Scan
Reading: Lin & Snyder, chapter 5, pp. 112–125
Mini-Assignment: Mini-Assignment 2 due **10:00am**

January 16: Generalized Reduce and Scan
Homework: Homework 1 deadline for early-bird bonus (11:59pm)
Homework 2 goes out (due Feb. 1) – Reduce and Scan

January 18: Reduce and Scan Examples
Homework: Homework 1 due **11:59pm**

January 20–27: Parallel Architecture

January 29–February 6: Parallel Performance

February 8–17: Parallel Sorting

Review Questions

- What is the difference between `==` and `=:=` ?
- What is an atom?
- Which of the following are valid Erlang variables, atoms, both, or neither?
  `Foo`, `foo`, 25, `'25'`, `'Foo foo'`,
  `"4 score and 7 years ago"`, X2,
  `'4 score and 7 years ago'`.
- Draw the tree corresponding to the nested list
  `[X, [[Y, Z], 2, [A, B+C, [], 23]], 14, [[[8]]]].`
- What is referential transparency?
- Why don't functional languages have loops?
- Use an anonymous function and `lists:filter` to implement the body of `GetEven` below:

```
% GetEven(List) -> Evens, where Evens is a list consisting of all
%     elements of List that are integers and divisible by two.
%     Example: GetEven([1, 2, frog, 1000]) -> [2, 1000]
GetEven(List) ->
   you write this part.
```

A Few More Review Questions

- Use a list comprehension to implement to body of `Double` below:

```
% Double(List) -> List2, where List is a list of numbers, and
%     List2 is the list where each of these are doubled.
%     Example: Double([1, 2, 3.14159, 1000]) ->
%         [2, 4, 6.28318, 2000]
Double(List) ->
   you write this part.
```

- Use a list comprehension to write the body of `Evens` as described on the previous slide.
- What is a tail-recursive function?
- In general, which is more efficient, a head-recursive or a tail-recursive implementation of a function? Why?

Supplementary Material

The remaining slides are some handy material that we won't cover in lecture, but you can refer to if you find it helpful.

- Erlang resources.
- Finishing the merge sort example.
- Common mistakes with lists and how to avoid them.
- A few remarks about atoms.
- Suppressing verbose output when using the Erlang shell.
- Forgetting variable bindings (only in the Erlang shell).
- Table of Contents.

Erlang Resources

- *LYSE* – you should be reading this already!
- Install Erlang on your computer
  - Erlang solutions provides packages for Windows, OSX, and the most common linux distros
    `https://www.erlang-solutions.com/resources/download.html`
  - Note: some linux distros come with Erlang pre-installed, but it might be an old version. You should probably install from the link above.
- `http://www.erlang.org`
  - Searchable documentation
    `http://erlang.org/doc/search/`
  - Language reference
    `http://erlang.org/doc/reference_manual/users_guide.html`
  - Documentation for the standard Erlang library
    `http://erlang.org/doc/man_index.html`
- The CPSC 418 Erlang Library
  - Documentation
    `http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/doc/index.html`
  - `.tgz` (source, and pre-compiled `.beam`)
    `http://www.ugrad.cs.ubc.ca/~cs418/resources/erl/erl.tgz`

Finishing the merge sort example

- Write `merge(List1, List2) -> List12` – see slide 30
- Write an Erlang modle with the `sort`, `split`, and `merge` functions – see slide 31
- Run the code – see slide 33

`merge(L1, L2)`

- Precondition: We assume `L1` and `L2` are each in non-decreasing order.
- Return value: a list that consists of the elements of `L1` and `L2` and the elements of the return-list are in non-decreasing order.
- Identify the cases and their return values.
  - What if `L1` is empty?
  - What if `L2` is empty?
  - What if both are empty?
  - What if neither are empty?
  - Are there other cases?
    Do any of these cases need to be broken down further?
    Are any of these case redundant?
- Now, try writing the code (an exercise for the reader).

Modules

- To compile our code, we need to put it into a module.
- A module is a file (with the extension `.erl`) that contains
  - Attributes: declarations of the module itself and the functions it exports.
    - The module declaration is a line of the form:
      `-module(moduleName).`
      where `moduleName` is the name of the module.
    - Function exports are written as:
      `-export([functionName1/arity1, functionName2/arity2, ...]).`
      The list of functions may span multiple lines and there may be more than one `-export` attribute.
      `arity` is the number of arguments that the function has. For example, if we define
      `foo(A, B) -> A*A + B.`
      Then we could export `foo` with
      `-export([..., foo/2, ...]).`
    - There are many other attributes that a module can have. We'll skip the details. If you really want to know, it's all described here.
  - Function declarations (and other stuff) – see the next slide

A module for `sort`

```
-module(sort).
-export([sort/1]).
% The next -export is for debugging. We'll comment it out later
-export([split/1, merge/2]).
sort([]) -> [];
...
```

Let's try it!

```
1> c(sort).
{ok,sort}
2> R20 = count3s:rlist(20, 100).  % test case: a random list
[45,73,95,51,32,60,92,67,48,60,15,21,70,16,56,22,46,43,1,57]
3> S20 = sort:sort(R20).  % sort it
[1,15,16,21,22,32,43,45,46,48,51,56,57,60,60,67,70,73,92,95]
4> R20 -- S20.  % empty if each element in R20 is in S20
[]
5> S20 -- R20.  % empty if each element in S20 is in R20
[]
```

- Yay – it works!!! (for one test case)
- The code is available at
  `http://www.ugrad.cs.ubc.ca/~cs418/2016-2/notes/01-06/src/sort.erl`

Remarks about Constructing Lists

It's easy to confuse `[A, B]` and `[A | B]`.

- This often shows up as code ends up with crazy, nested lists; or code that crashes; or code that crashes due to crazy, nested lists; . . . .
- Example: let's say I want to write a function `divisible_drop(N, L)` that removes all elements from list `L` that are divisible by `N`:

```
divisible_drop(_N, []) -> []; % the usual base case
divisible_drop(N, [A | Tail]) ->
   if A rem N == 0 -> divisible_filter(N, Tail);
      A rem N /= 0 -> [A | divisible_filter(N, Tail)]
   end.
```

It works. For example, I included the code above in a module called `examples`.

```
6> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
[1,4,17,100]
```

Misconstructing Lists

Working with `divisible_drop` from the previous slide. . .

- Now, change the second alternative in the `if` to
  `A rem N /= 0 -> [A, divisible_filter(N, Tail)]`
  Trying the previous test case:

```
7> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
[1,[4,[17,[100,[]]]]]
```

  Moral: If you see a list that is nesting way too much, check to see if you wrote a comma where you should have used a `|`.
- Restore the code and then change the second alternative for `divisible_drop` to `divisible_drop(N, [A, Tail]) ->` Trying our previous test:

```
8> examples:divisible_drop(3, [0, 1, 4, 17, 42, 100]).
** exception error:  no function clause matching...
```

Punctuation

- Erlang has lots of punctuation: commas, semicolons, periods, and `end`.
- It's easy to get syntax errors or non-working code by using the wrong punctuation somewhere.
- Rules of Erlang punctuation:
  - Erlang declarations end with a period: `.`
  - A declaration can consist of several alternatives.
    - Alternatives are separated by a semicolon: `;`
    - Note that many Erlang constructions such as `case`, `fun`, `if`, and `receive` can have multiple alternatives as well.
  - A declaration or alternative can be a block expression
    - Expressions in a block are separated by a comma: `,`
    - The value of a block expression is the last expression of the block.
  - Expressions that begin with a keyword end with `end`
    - `case` *Alternatives* `end`
    - `fun` *Alternatives* `end`
    - `if` *Alternatives* `end`
    - `receive` *Alternatives* `end`

Remarks about Atoms

- An atom is a special constant.
  - Atoms can be compared for equality.
  - Actually, any two Erlang can be compared for equality, and any two terms are ordered.
  - Each atom is unique.
- Syntax of atoms
  - Anything that looks like an identifier and starts with a lower-case letter, e.g. `x`.
  - Anything that is enclosed between a pair of single quotes, e.g. `'47 BIG apples'`.
  - Some languages (e.g. Matlab or Python) use single quotes to enclose string constants, some (e.g. C or Java) use single quotes to enclose character constants.
    - But not Erlang.
    - The atom `'47 big apples'` is not a string or a list, or a character constant.
    - It's just its own, unique value.
  - **Atom constants can be written with single quotes, but they are not strings.**

Avoiding Verbose Output

- Sometimes, when using Erlang interactively, we want to declare a variable where Erlang would spew enormous amounts of "uninteresting" output were it to print the variable's value.
  - We can use a comma (i.e. a block expression) to suppress such verbose output.
  - Example

```
9> L1_to_5 = lists:seq(1, 5).
[1, 2, 3, 4, 5].
10> L1_to_5M = lists:seq(1, 5000000), ok.
ok
11> length(L1_to_5M).
5000000
12>
```

Forgetting Bindings

- Referential transparency means that bindings are forever.
  - This can be nuisance when using the Erlang shell.
  - Sometimes we assign a value to a variable for debugging purposes.
  - We'd like to overwrite that value later so we don't have to keep coming up with more name.s
- In the Erlang shell, `f(Variable).` makes the shell "forget" the binding for the variable.

```
12> X = 2+3.
5.
13> X = 2*3.
** exception error:  no match of right hand side value 6.
14> f(X).
ok
15> X = 2*3.
6
16>
```

Table of Contents

- Erlang Basics – basic types and their operations.
- Functional Programming – referential transparency, recursion instead of loops.
- Example: Merge Sort
- Fun with functions – patterns, anonymous functions, higher-order functions, list comprehensions, head vs. tail recursion
- Preview of upcoming lectures
- Review of this lecture
- Supplementary Material

Processes and Messages

Mark Greenstreet

CpSc 418 – Jan. 9, 2017

Outline:

- Processes
- Messages
- Timing Measurements
- Preview, Review, etc.
- Table of Contents

Objectives

- Introduce Erlang's features for concurrency and parallelism
  - Spawning processes.
  - Sending and receiving messages.
- Describe timing measurements for these operations and the implications for writing efficient parallel programs.
  - **Communication often dominates the runtime of parallel programs.**
- The source code for the examples in this lecture is available here: procs.erl.

Processes – Overview

- The built-in function spawn creates a new process.
- Each process has a process-id, pid.
  - The built-in function self() returns the pid of the calling process.
  - `spawn` returns the pid of the process that it creates.
  - The simplest form is `spawn(`*Fun*`)`.
    - ★ A new process is created – "the child".
    - ★ The pid of the new process is returned to the caller of `spawn`.
    - ★ The function *Fun* is invoked with no arguments in that process.
    - ★ The parent process and the child process are both running.
    - ★ When *Fun* returns, the child process terminates.

Processes – a friendly example

```
hello(N)->
   [     spawn(fun() -> io:format(
           "hello world from process ~b~n", [I])
         end)
      || I <- lists:seq(1,N)
   ].
```

Running the code:

```
1> c(procs).
{ok,procs}
2> procs:hello(3).
hello world from process 1
hello world from process 2
hello world from process 3
[<0.40.0>,<0.41.0>,<0.42.0>]
```

Messages

- To solve tasks in parallel, the processes need to communicate.
- Sending a message: `Pid ! Expr`.
  - `Expr` is evaluated, and the result is sent to process `Pid`.
  - We can send **any** Erlang term: integers, atoms, lists, tuples, . . .
- Receiving a message:

```
receive
   Pattern1 -> Expr1;
   Pattern2 -> Expr2;
   ...
   PatternN -> ExprN
end
```

If there is a pending message for this process that matches one of the patterns,
  - The message is delivered, and the value of the `receive` expression is the value of the corresponding *Expr*.
  - Otherwise, the process blocks until such a message is received.
- Message passing is asynchronous: the sending process can continue its execution before the receiver gets the message.

Adding two numbers using processes and messages

- The plan:
  - We'll spawn a process in the shell for adding two numbers.
  - This child process receives two numbers, computes the sum, and sends the result back to the parent.

```
add_proc(PPid) ->
   receive
      A -> receive
         B ->
            PPid ! A+B
      end
   end.

adder() ->
   MyPid = self(),
   spawn(fun() ->
      add_proc(MyPid)
   end).
```

```
3> Apid = procs:adder().
<0.44.0>
4> Apid ! 2.
2
5> Apid ! 3.
3
6> receive Sum -> Sum end.
5
```

Reactive Processes and Tail Recursion

- Often, we want processes that do more than add two numbers together.
- We want processes that wait, receive a message, process the message, and then wait for the next message.
- In Erlang, we do this with recursive functions for the child process:

```
acc_proc(Tally) ->
   receive
      N when is_integer(N) ->
         acc_proc(Tally+N);
      {Pid, total} ->
         Pid ! Tally,
         acc_proc(Tally)
   end.

   accumulator() ->
      spawn(fun() ->
         acc_proc(0)
      end).
```

```
7> BPid = procs:accumulator().
<0.53.0>
8> BPid ! 1.
1
9> BPid ! 2.
2
10> BPid ! 3.
3
11> BPid ! {self(), total}.
{<0.33.0>, total}
12> receive T1 -> T1 end.
6
```

Reactive Processes and Tail Recursion

- Often, we want processes that do more than add two numbers together.
- We want processes that wait, receive a message, process the message, and then wait for the next message.
- In Erlang, we do this with recursive functions for the child process:

```
acc_proc(Tally) ->
   receive
      N when is_integer(N) ->
         acc_proc(Tally+N);
      {Pid, total} ->
         Pid ! Tally,
         acc_proc(Tally)
   end.

   accumulator() ->
      spawn(fun() ->
         acc_proc(0)
      end).
```

```
13> BPid ! 4.
4
14> BPid ! {self(), total}.
{<0.33.0>, total}
15> BPid ! 5.
5
16> BPid ! 6.
6
17> BPid ! {self(), total}.
{<0.33.0>, total}
18> receive T2 -> T2 end.
10
19> receive T3 -> T3 end.
21
```

Message Ordering

- Given two processes, *Proc1* and *Proc2*, messages sent from *Proc1* to *Proc2* are received at *Proc2* in the order in which they were sent.
- Message delivery is reliable: if a process doesn't terminate, any message sent to it will eventually be delivered.
- Other than that, Erlang makes no ordering guarantees.
  - In particular, the triangle inequality is not guaranteed.
  - For example, process *Proc1* can send message *M1* to process *Proc2* and after that send message *M2* to *Proc3*.
  - Process *Proc3* can receive the message *M2*, and then send message *M3* to process *Proc2*.
  - Process *Proc2* can receive messages *M1* and *M3* in either order.
  - Draw a picture to see why this is violates the spirit of the triangle inequality.

Tagging Messages

- It's a very good idea to include "tags" with messages.
- This prevents your process from receiving an unintended message:

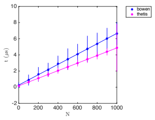
  > *"Oh, I forgot that another process was going to send me that. I thought it would happen later."*

- For example, my accumulator might be better if instead of just receiving an integer, it received `{2, add}`

Timing Measurements

- We write parallel code to solve problems that would take too long on a single CPU.
- To understand performance trade-offs, I'll measure the time for some common operations in Erlang programs:
  - The time to make *N* recursive tail calls.
  - The time to spawn an Erlang process.
  - The time to send and receive messages:
    - ★ Short messages.
    - ★ Messages consisting of lists of varying lengths.

Tail Call Time



`bowen.ugrad.cs.ubc.ca`:

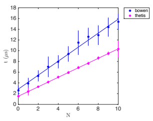
$t = (6.4N + 269)\text{ns}$,	line of best fit
$t = 64.3\mu\text{s}$,	$N =$ 10*K*
$t = 640\mu\text{s}$,	$N =$ 100*K*

`thetis.ugrad.cs.ubc.ca`:

$t = (4.7N + 170)\text{ns}$,	line of best fit
$t = 46.9\mu\text{s}$,	$N =$ 10*K*
$t = 466\mu\text{s}$,	$N =$ 100*K*

- Measurement: start the timing measurement, make *N* tail calls, end the timing measurement.
- The measurements on this slide and throughput the lecture were made using the `time_it:t` function from the course Erlang library.
  - `time_it:t(`*Fun* repeatedly calls *Fun* until about one second has elapsed. It then reports the average time and standard deviation.
  - `time_it:t` has lots of options.

Process Spawning Time



`bowen.ugrad.cs.ubc.ca`:

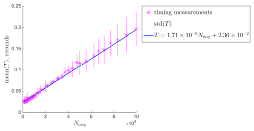
$t = (1.30N + 2.8)\mu\text{s}$,	line of best fit
$t = 127\mu\text{s}$,	$N =$ 100
$t = 1.2\text{ms}$,	$N = 1000$

`thetis.ugrad.cs.ubc.ca`:

$t = (0.88N + 1.5)\mu\text{s}$,	line of best fit
$t = 89.4\mu\text{s}$,	$N =$ 100
$t = 887\mu\text{s}$,	$N = 1000$

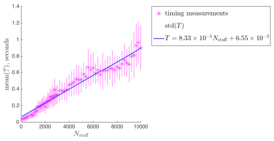
- Measurement: root spawns *Proc1*; *Proc1* spawns *Proc2*, and then *Proc1* exits; *Proc2* spawns *Proc3*, and then *Proc2* exits; . . . ; *ProcN* sends a message to the root process, and then *ProcN* exits. The root process measures the time from just before spawning *Proc1* until receiving the message from *ProcN*.

Send+Receive Time



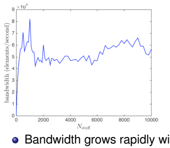
- Set-up: Two processes do a fixed amount of "work" while exchanging short messages with non-blocking receives.
- N_{msg} is the number of messages sent and received by each process.
- The slope of the line is the time per message:
  - $\sim 1.7\mu\text{s/message}$ on `thetis.ugrad.cs.ubc.ca`, `erts` 18.2.
  - My laptop is about three-times faster. I'm running `erts` 19.2.

Message Time vs. Message Size



- Set-up: as on the previous slide. This time each message consists of a list of N_{stuff} small integers.
- Each process sends and receives 5000 messages per run.
- The slope of the line divided by 5000 is the time per element:
  - $\sim 17\text{ns/message}$ on `thetis.ugrad.cs.ubc.ca`, `erts` 18.2.

Bandwidth vs. Message Size



Subtract the "non-message" time from the run-time and calculate:

$$\frac{N_{msg} \times N_{stuff}}{T}$$

To get elements per second.

- Bandwidth grows rapidly with message length for $N_{stuff} < 1000$, then drops.
  - Short messages have low bandwidth due to fixed overheads with each message.
  - I'm guessing that bandwidth drops some for messages with more than 1000 elements because the Erlang runtime is somehow optimized for short messages.

Summarizing the numbers

- Interprocess operations such as `spawn`, `send`, and `receive` are **much** slower than operations within a single process such as + or a function call.
- An Erlang tail call is about 4.7ns, roughly 10 machine instructions.
- An Erlang tail call and add is about 4.7ns, roughly 10 machine instructions.
- Spawning a process is about 200× the cost of a tail call.
- For short messages, send and receive are about 350× the cost of a tail call.
  - The send/receive overhead can be amortized by sending longer message.
  - Each additional list element is about 3× the cost of a tail call.
  - Beware of any model that just counts the overhead and ignores the length, or just considers bandwidth and ignores the overhead.
- We will often refer to the ratio of the relationship between the time for interprocess operations and local operations as **big**.
  - In practice, **big** is 100 to 10000 for shared-memory computers.
  - **Big** can be even bigger for other architectures.

How to Write Efficient Parallel Code

- Think about **communication costs**
  - Message passing is good – it makes communication explicit.
  - Pay attention to both the number of messages and their size.
  - Combining small messages into larger ones often helps.
- Think globally, but **compute locally**
  - Move the computation to the data, not the other way around.
  - Keep the data distributed across the parallel processes.
- Think about **big**–*O*
  - If *N* is the problem size, you want the computation time to grow faster with *N* than the communication costs.
  - Then, your solution becomes more efficient for larger values of *N*.

Summary

- Processes are easy to create in Erlang.
  - The `spawn` mechanism can be used to start other processors on the same CPU or on machines spread around the internet.
- Processes communicate through messages
  - Message passing is asynchronous.
  - The receiver can use patterns to select a desired message.
- Reactive processes are implemented with tail-recursive functions.
- Interprocess operations are much slower than local ones
  - This is a key consideration in designing parallel programs.
  - We'll learn **why** when we look at parallel architectures later this month.

Preview

January 11: Reduce	
Reading:	*Learn You Some Erlang*, Errors and Exceptions through A Short Visit to Common Data Structures
January 13: Scan	
Reading:	Lin & Snyder, chapter 5, pp. 112–125
Mini-Assignment:	Mini-Assignment 2 due **10:00am**
January 16: Generalized Reduce and Scan	
Homework:	Homework 1 deadline for early-bird bonus (11:59pm) **Homework 2 goes out (due Feb. 1)** – Reduce and Scan
January 18: Reduce and Scan Examples	
Homework:	Homework 1 due **11:59pm**
January 20–27: Parallel Architecture	
January 29–February 6: Parallel Performance	
February 8–17: Parallel Sorting	

Review Questions

- How do you spawn a new process in Erlang?
- What guarantees does Erlang provide (or not) for message ordering?
- Give an example of using patterns to select messages.
- Why is it important to use a tail-recursive function for a reactive process?
  - In other words, why is it a bad idea to use a head-recursive function for a reactive process.
  - The answer isn't explicitly on the slides, but you should be able to figure it out from what we've covered.
- Modify one of the examples in this lecture to use a time-out with one or more `receive` operations. Try it and show that it works.
- Implement the message flushing described in *LYSE* to show pending messages on a time-out. Demonstrate how it works.

Supplementary material

- Debugging concurrent Erlang Code.
- Table of contents.

Tracing Processes

When you implement a reactive process, it can be handy to trace the execution. Here's a simple approach:

- Add an `io:format` call when entering the function and after matching each receive pattern.
- Example:

```
acc_proc(Tally) ->
   io:format("~p:  acc_proc(~b)~n", [self(), Tally]),
   receive
      N when is_integer(N) ->
         io:format("~p:  received ~b~n", [self(), N]),
         acc_proc(Tally+N);
      Msg = {Pid, total}
         io:format("~p:  received ~p~n", [self(), Msg]),
         Pid ! Tally,
         acc_proc(Tally)
   end.
```

- Try it (e.g. with the example from slide 7.
- Don't forget to delete (or comment out) such debugging output before releasing your code.

Time Outs

- If your process is waiting for a message that never arrives, e.g. because
  - You misspelled a tag for a message, or
  - The receive pattern is slightly different than the message that was sent, or
  - Something went wrong in the sending process, and it died before sending the message, or
  - You got the message ordering slightly wrong, and there's a cycle of processes waiting for each other to send something, or
  - . . .
- Then your process can wait forever, your Erlang shell can hang, and it's a very unhappy time in life.
- Time-outs can handle these problems more gracefully.
  - See Time Out in *LYSE*.
  - Note: time-outs are great for debugging. They should be used with great caution elsewhere because they are sensitive to changes in hardware, changes in the scale of the system, and so on.

Table of Contents

Scan

Mark Greenstreet

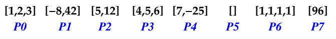
CpSc 418 – Jan. 13, 2017

Outline:

- Reduce Redux
  - The basic algorithm.
  - Performance model.
  - Implementation considerations.
- Scan
  - Understand how reduce generalizes to a method that produces all *N* values for a "cumulative" operation in *O*(log *N*) time.
- A few implementation notes

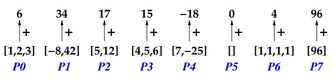
Reduce Redux

Problem statement:
Given P processes that each hold part of an array of numbers, compute the sum of all the numbers in the combined array.



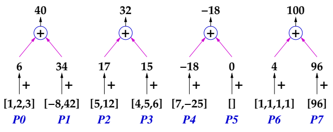
Reduce Redux

Accumulate step:
Each process computes the total of the elments in its local part of the array.



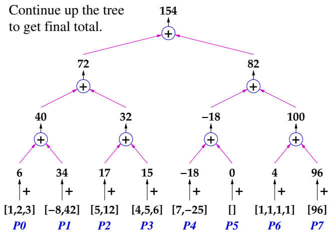
Reduce Redux

Combine step:
Each process sends its result to a coombiner process.
The combiners compute the sums of the values from adjacent pairs of processes.



Reduce Redux

Continue up the tree to get final total.



Reduce Notes

- For simplicity, I drew the tree as if we used separate processes for accumulating the local arrays and doing the combining.
  - In practice, we use the same processes for both accumulating and combining.
  - Note that ½ of the processes are active in the first level of combine; ¼ of the processes are active in the second level; and so on.
- Simple time model:

$$T \in O\left(\frac{N}{P} + \lambda \log P\right)$$

where λ is **big** – i.e. the communication time.

Scan Problem Statement

- Given an array, A, with N elements.
  - Let $B = \text{scan}_+(A)$:

$$B_i = \sum_{k=0}^{i} A_k$$

  - Example:

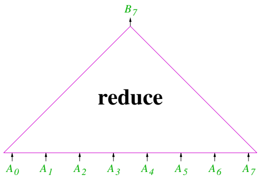
$$\begin{array}{rcl} A & = & [1,2,3,-8,42,5,12,4,5,6,7,-25,1,1,1,1,96] \\ B & = & [1,3,6,-2,40,45,57,61,66,72,79,54,55,56,57,58,154] \end{array}$$

- Is there an efficient parallel algorithm for computing $\text{scan}_+(A)$?
  - I wrote scan_+ because our solution works for any associative operator.

Scan Example: Monthly Bank Statement

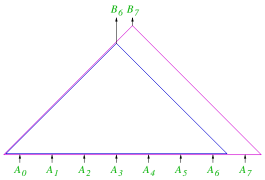
- Assumptions:
  - You make **lots** of transactions; so, the bank needs to use a parallel algorithm just for your account.
  - Months have 32 days – the power-of-two version of the algorithm is simpler. It generalizes to any number of processors.
  - Each processes has the transaction data for one day.
- Using parallel scan:
  - Each process computes the total of the transactions for its day.
  - Using parallel scan, we determine the balance at the beginning of each day for each process.
  - The process can use its start-of-day balance, and compute the balance after each transaction for that day.

Brute force Scan



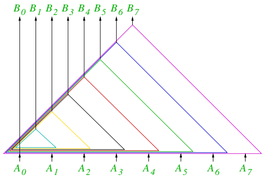
- Use a reduce tree to compute B_7.

Brute force Scan



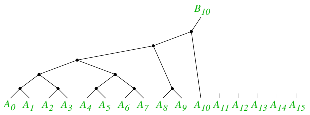
- Use a reduce tree to compute B_7.
- Use another reduce tree to compute B_6.

Brute force Scan



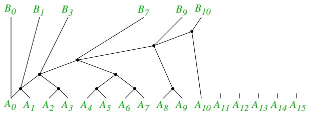
- Use a reduce tree to compute B_7.
- Use another reduce tree to compute B_6.
- Use 6 more reduce trees to compute $B_{5\ldots0}$
- It works. It's $O(\log P)$ time! But it's not very efficient.

Reuse trees


B_{10}
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}

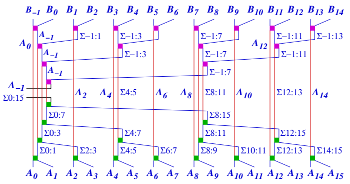
- Key idea: we don't need the trees to be balanced.
- We just want them to be $O(\log P)$ in height.
- If we need a tree for 2^k nodes, we'll make a balanced tree.
- Otherwise:
  - Make the largest balanced tree we can on the left.
  - Repeat this process for what's left on the right.

Reuse trees


B_0 B_1 B_3 B_7 B_9 B_{10}
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}

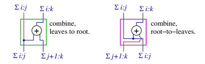
- Key idea: we don't need the trees to be balanced.
- We just want them to be $O(\log P)$ in height.
- If we need a tree for 2^k nodes, we'll make a balanced tree.
- Otherwise:
  - Make the largest balanced tree we can on the left.
  - Repeat this process for what's left on the right.
- Notice that while computing B_{10}, we produced many other of the Bs as intermediate results.

Scan


B_{-1} B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14}
A_0
A_{-1}
$\Sigma 0{:}15$
Σ–1:1
Σ–1:3
Σ–1:7
Σ–1:11
Σ–1:13
A_2 A_4 Σ4:5 A_6 A_8 Σ8:11 A_{10} Σ12:13 A_{12} A_{14}
Σ0:7
Σ8:15
Σ0:3
Σ4:7
Σ8:11
Σ12:15
Σ0:1 Σ2:3 Σ4:5 Σ6:7 Σ8:9 Σ10:11 Σ12:13 Σ14:15
A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}

- See the next slide for an explanation of the notation, etc.

Scan: explained



The green and magenta boxes are both "combine" units.The only difference is the terminal placement, to make the big diagram less cluttered.

- Notation
  - A_{-1} is initializer for the sum.
  - $A_0, A_1, \ldots A_{15}$ is the initial array.
  - $B_{-1}, B_1, \ldots B_{15}$ is the result of the scan.

$$B_i = \sum_{k=-1}^{i} A_k, \text{Include the initializer } A_{-1}$$

  - $\Sigma i : j$ is shorthand for $\sum_{k=i}^{j} A_k$
- Each process needs to compute its local part of the scan at the end, starting from the value it receives from the tree.

A few implementation notes

- On slide 3 I pointed out that for efficiency, it is better to use the same processes for the leaves and the combine.

```
% reduce:
  treeLevels = ceil(log2(NProcs0);
  tally = localAccumulate(...);
  for(k = 0; k < treeLevels; k++) {
     if((myPid & (1 << k)) != 0) {
        send(myPid - (1 << k), myPid, tally);
        break;
     } else
        tally += receive(myPid + (1<< k));
  } // Process 0 now has the grand total.
  // We can use another loop to broadcast the result.
```

- I'll provide an Erlang version on Monday.

Reduce & Scan

Scan is very similar to reduce. We just change the downward tree.

- For reduce, each process just forwards the grand total to its descendants.
- For scan:
  - Each process records the tallies from its left subtree(s) during the upward sweep.
  - During the downward sweep, each process receives the tally for **everything to the left of the subtree for this process**.
    - ★ The process adds the tally from its own left subtree to the value from its parent, and sends this to its own right subtree.
    - ★ The process continues the downward sweep for its own left subtree.
    - ★ When we reach a leaf, the process does the final accumulate.

Preview

January 16: Generalized Reduce and Scan
Homework: Homework 1 deadline for early-bird bonus (11:59pm)
Homework 2 goes out (due Feb. 1) – Reduce and Scan

January 18: Reduce and Scan Examples
Homework: Homework 1 due **11:59pm**

January 20: Architecture Review
Reading: Pacheco, Chapter 2, through section 2.2

January 23: Shared Memory Architectures
Reading: Pacheco, Chapter 2, through section 2.3
Homework: Homework 2 deadline for early-bird bonus (11:59pm)
Homework 3 goes out (due Feb. 17)

January 25: Message Passing Architectures
Homework: Homework 2 due **11:59pm**

January 27–February 6: Parallel Performance

February 8–17: Parallel Sorting

Review Questions

- What is the cumulative sum of `[1,7,-5,12,73,19,0,12]`?
  - For the same list as above, what is the cumulative product?
  - For the same list as above, what is the cumulative maximum?
- Draw a tree showing how the sum (simple, not cumulative) of the values in the list above can be computed using reduce. Assume that there are eight processes, and each starts with one element of the list.
- Draw a graph like the one on slide 8 for a scan of eight values.
- Label each edge of your graph with the value that will be sent along that edge when computing the cumulative sum of the values in the list above. Assume that there are eight processes, and each starts with one element of the list.
- Add a second label to each edge indicating whether the value is local to that process or if the edge requires inter-process communication. Write 'L' for local, and 'G' for global (i.e. inter-process communication).

Generalize Reduce and Scan

Mark Greenstreet

CpSc 418 – Jan. 16, 2017

Outline:

- Reduce in Erlang
- Scan in Erlang

Objectives

- Understand relationship between reduce and scan
  - Both are tree walks.
  - The initial combination of values from leaves is identical.
  - Reduce propagates the grand total down the tree.
  - Scan propagates the total "everything to the left" down the tree.
- Generalized Reduce and Scan
  - Understand the role of the *Leaf*, `Combine`, and `Root` functions.
  - Understand the use use of higher-order functions to implement reduce and scan.
- The CS418 class library
  - Able to create a tree of processes.
  - Able to distribute data and tasks to those processes.
  - Able to use the `reduce` and `scan` functions from the library.
  - Know where to find more information.

Reduce in Erlang

- Build a tree.
- Each process creates a lists of random digits.
- The processes meet at a barrier so we can measure the time to count the 3s.
- Each process counts its threes.
- The processes use reduce to compute the grand total.
- Each process reports the grand total and its own tally.
- The root process reports the time for the local tallies and the reduce.
- Get the code at

http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-16/code/reduce.erl

The Reduce Pattern

- It's a parallel version of *fold*, e.g. `lists:foldl`.
- Reduce is described by three functions:

  Leaf(): What to do at the leaves, e.g.
  `fun() -> count3s(Data) end`.
  Combine(): What to do at the root, e.g.
  `fun(Left, Right) -> Left+Right end`.
  Root(): What to do with the final result. For count 3s, this is just the identity function.

The `wtree` module

- Part of the course Erlang library.
- Operations on worker trees"

  `wtree:create(NProcs) -> [pid()].`
  Create a list of `NProcs` processes, organized as a tree.
  `wtree:broadcast(W, Task, Arg) -> ok.`
  Execute the function `Task` on each process in `W`. Note: `W` means "worker pool".
  `wtree:reduce(P, Leaf, Combine, Root) -> term().`
  A generalized reduce.
  `wtree:reduce(P, Leaf, Combine) -> term().`
  A generalized reduce where `Root` defaults to the identity function.

Store Locally

- Communication is expensive – each process should store its own data whenever possible.
- How do we store data in a functional language?
  - Our processes are implemented as Erlang functions that receive messages, process the message, and make a tail-call to be ready to receive the next message.
  - We add a parameter to these functions, `State`, that is a mapping from *Keys* to *Values*.
- What this means when we write code:

  Functions such as *Leaf* for `wtree:reduce` or *Task* for `wtree:broadcast` have a parameter for `State`.
  `worker:put(State, Key, Value) -> NewState.`
  Create a new version of `State` that associates `Value` with `Key`.
  `worker:get(State, Key, Default) -> Value.`
  Return the value associated with `Key` in `State`. If no such value is found, `Default` is returned. Note: `Default` can be a function in which case it is called to determine a default value – see the documentation.

Count3s using `wtree`

```
count3s_par(N, P) ->
   W = wtree:create(P),
   wtree:rlist(W, N, 10, 'Data'),
   wtree:reduce(W,
      fun(ProcState) ->
         count3s(workers:get(ProcState, 'Data'))
      end,
      fun(Left, Right) -> Left+Right end
   ).
```

Reduce and Scan

- The root node:
  - Reduce: `count3s_reduce(none, [], Total3s) -> Total3s;`
  - Scan: `count3s_scan(none, [], Total3s) -> 0;`
- Internal nodes:

```
% Reduce:
 count3s_reduce(Parent, [Child | MoreKids], ThreesInLeftSubtree) ->
    ThreesInRightSubtree = count3s_wait(Child),
    ThreesInMyTree = ThreesInLeftSubtree + ThreesInRightSubtree,
    Total3s = count3s_reduce(Parent, MoreKids, ThreesInMyTree),
    count3s_notify(Child, Total3s).

% Scan:
 count3s_scan(Parent, [Child | MoreKids], ThreesInLeftSubtree) ->
    ThreesInRightSubtree = count3s_wait(Child),
    ThreesInMyTree = ThreesInLeftSubtree + ThreesInRightSubtree,
    ThreesToMyLeft = count3s_scan(Parent, MoreKids, ThreesInMyTree),
    count3s_notify(Child, ThreesToMyLeft + ThreesInLeftSubtree),
    ThreesToMyLeft.
```

Scan in Erlang

- Remarkably like reduce.
- Reduce has
  - an upward pass to compute the grand total
  - a downward pass to broadcast the grand total.
- Scan has
  - an upward pass where the grand total – **just like reduce**
  - On the downward pass, we compute the total of all elements to the left of each subtree.
- Get the code at

http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/01-16/code/scan.erl

The Scan Pattern

- It's a parallel version of *mapfold*, e.g. `lists:mapfoldl` and `lists:mapfoldr`.
- `wtree:scan(`*Leaf1, Leaf2, Combine, Acc0*`)`
  - *Leaf1*`(`*ProcState*`) ->` *Value*
    Each worker process computes its *Value* based on its *ProcState*.
  - *Combine*`(`*Left*`,` *Right*`) ->` *Value*
    Combine values from sub-trees.
  - *Leaf2*`(`*ProcState*`,` *AccIn*`) ->` *ProcState*
    Each worker updates its state using the *AccIn* value – i.e. the accumulated value of everything to the worker's "left".
  - *Acc0*: The value to use for *AccIn* for the leftmost nodes in the tree.

Scan example: prefix sum

```
prefix_sum_par(W, Key1, Key2) ->
   wtree:scan(W,
      fun(ProcState) -> % Leaf1
         lists:sum(wtree:get(ProcState, Key1)) end,
      fun(ProcState, AccIn) -> % Leaf2
         wtree:put(ProcState, Key2,
            prefix_sum(wtree:get(ProcState, Key1), AccIn)
         ) end,
      fun(Left, Right) -> % Combine
         Left + Right end,
      0 % Acc0
   ).

prefix_sum(L, Acc0) ->
   element(1,
      lists:mapfoldl(fun(X, Y) -> Sum = X+Y, {Sum,Sum} end,
                     Acc0, L).
```

More Examples of scan

- Account balance with interest:
  - Input: a list of transactions, where each transaction can be a deposit (add an amount to the balance), a withdrawal (subtract an amount from the balance), or interest (multiply the balance by an amount). For example:
    `[{deposit, 100.00}, {withdraw, 5.43}, {withdraw, 27.75}`
  - Output: the account balance after each transaction. For example, if we assume a starting balance of $1000.00 in the previous example, we get
    `[1100.00, 1094.57, 1066.82, 1067.40, ...]`
- Delete 3s
  - Given a list that is distributed across *NProc* processes, delete all 3s, and rebalance the list so each process has roughly the same length sublisth.
  - Solution (sketch):
    - □ Using scan, each process determines how many 3s preceed its segment, the total list length preceeding it, and the total list length after deleting 3s.
    - □ Each process deletes its 3s and send portions of its lists

Preview

January 18: Reduce and Scan Examples
Homework: Homework 1 due **11:59pm**

January 20: Finish Reduce and Scan
Mini-assignments: Mini assignment 3 goes out.

January 23: Architecture Review
Reading: Pacheco, Chapter 2, through section 2.2

January 27: Shared Memory Architectures
Reading: Pacheco, Chapter 2, through section 2.3
Mini-assignments: Mini assignment 3 due, 10am.

January 27: Message Passing Architectures

January 27–February 6: Parallel Performance
January 30: HW 2 Earlybird due (11:59pm), HW 3 goes out.
February 1: HW 2 due (11:59pm).

February 8–17: Parallel Sorting
February 15: HW 3 Earlybird (11:59pm).
February 17: HW 3 due (11:59pm).
February 27: TBD

March 1: Midterm

Scan

Mark Greenstreet

CpSc 418 – Jan. 20, 2016

Objectives

- Prefix sum
  - Spawning processes.
  - Sending and receiving messages.
- The source code for the examples in this lecture is available here: procs.erl.

Prefix Sum

- Scan is similar to reduce, but every process calculates its cumulative total.
- Example:

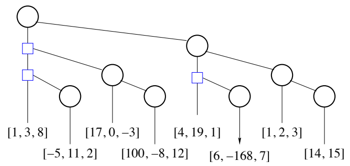
```
% prefix_sum: compute prefix sum.
prefix_sum(L) when is_list(L) -> prefix_sum_tr(L, 0).
prefix_sum_tr([], Acc) -> [];
prefix_sum_tr([H | T], Acc) ->
   MySum = H+Acc,
   [MySum | prefix_sum_tr(T, MySum)].
```

- Let's try it:

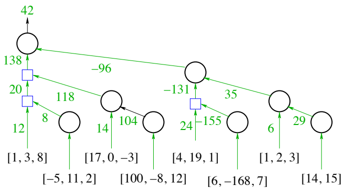
```
1> examples:prefix_sum([1, 13, 2, -5, 17, 0, 33]).
[1,14,16,11,28,28,61]
```

- How can we do this in parallel?

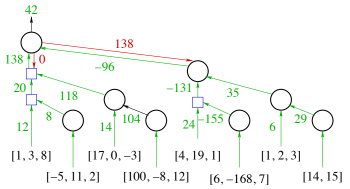
Parallel Prefix Sum



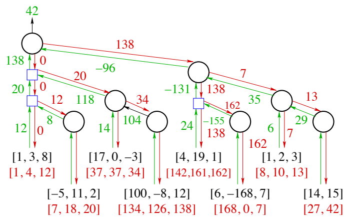
Parallel Prefix Sum



Parallel Prefix Sum



Parallel Prefix Sum



The Scan Pattern

- It's a parallel version of *mapfold*, e.g. `lists:mapfoldl` and `lists:mapfoldr`.
- `wtree:scan(`*Leaf1, Leaf2, Combine, Acc0*`)`
  - *Leaf1*`(`*ProcState*`) ->` *Value*
Each worker process computes its *Value* based on its *ProcState*.
  - *Combine*`(`*Left*`,` *Right*`) ->` *Value*
Combine values from sub-trees.
  - *Leaf2*`(`*ProcState*`,` *AccIn*`) ->` *ProcState*
Each worker updates its state using the *AccIn* value – i.e. the accumulated value of everything to the worker's "left".
  - *Acc0*: The value to use for *AccIn* for the leftmost nodes in the tree.

Scan example: prefix sum

```
prefix_sum_par(W, Key1, Key2) ->
   wtree:scan(W,
      fun(ProcState) -> % Leaf1
         lists:sum(wtree:get(ProcState, Key1)) end,
      fun(ProcState, AccIn) -> % Leaf2
         wtree:put(ProcState, Key2,
            prefix_sum(wtree:get(ProcState, Key1), AccIn)
         ) end,
      fun(Left, Right) -> % Combine
         Left + Right end,
      0 % Acc0
   ).

prefix_sum(L, Acc0) ->
   element(1,
      lists:mapfoldl(fun(X, Y) -> Sum = X+Y, {Sum,Sum} end,
                     Acc0, L)).
```

Prefix Sum Using Scan, example (part 1 of 4)

- Consider the example from slide 4.
  - We'll assume that the original lists for each processes are associated with the key `raw_data`.
  - We'll store the cummulative sum using the key `cooked_data`.
- `Leaf1`: each worker computes the sum of the elements in its list:
  - Worker 0:

```
Leaf1(ProcState) ->
   lists:sum(wtree:get(ProcState, raw_data)) ->
   lists:sum([1,3,8]) ->
12.
```

  - Worker 1:

```
Leaf1(ProcState) -> lists:sum([-5,11,2]) -> 8.
```

  - Worker 2:

```
Leaf1(ProcState) -> lists:sum([17,0,-3]) -> 14.
```

  - Workers 3–6: . . .
  - Worker 7:

```
Leaf1(ProcState) -> lists:sum([14,15]) -> 29.
```

Prefix Sum Using Scan, example (part 2 of 4)

- `Combine` (upward, first round):
  - Worker 0: `Combine(12, 8) -> 20`.
  - Worker 2: `Combine(14, 104) -> 118`.
  - Worker 4: `Combine(24, -155) -> -131`.
  - Worker 6: `Combine(6, 29) -> 35`.
- `Combine` (upward, second round):
  - Worker 0: `Combine(20, 118) -> 138`.
  - Worker 4: `Combine(-131, 35) -> -96`.
- `Combine` (upward, final round):
  - Worker 0: `Combine(138, -96) -> 42`.
  - This value is returned to the caller of `wtree:scan`.

Prefix Sum Using Scan, example (part 3 of 4)

- `Combine` (downward)
- The root sends `AccIn`, `0` to the left subtree.
- Each worker that did a combine remembers the arguments from the upward combines, and uses them in the downward sweep. In the code, each upward step is a recursive function call, and each downward step is a return.
- `Combine` (downward, first round)
  - Worker 0: `Combine(0, 138) -> 138`.
  - The `0` is `AccIn` from the root.
  - The `138` is the stored value from the left subtree.
  - Worker 0 sends this result to its right subtree, worker 4.
- `Combine` (downward, second round)
  - Worker 0: `Combine(0, 20) -> 20`. Send to worker 2.
  - Worker 4: `Combine(138, -131) -> 7`. Send to worker 6.
- `Combine` (downward, third round)
  - Worker 0: `Combine(0, 12) -> 12`. Send to worker 1.
  - Worker 2: `Combine(20, 14) -> 34`. Send to worker 3.
  - Worker 4: `Combine(138, 24) -> 162`. Send to worker 5.
  - Worker 6: `Combine(7, 6) -> 13`. Send to worker 7.

Prefix Sum Using Scan, example (part 4 of 4)

- `Leaf2` (update worker state)
  - Worker 0:

```
Leaf2(ProcState, 0) ->
   wtree:put(ProcState, Key2,
      prefix_sum(wtree:get(ProcState, Key1), 0)) ->
   wtree:put(ProcState, Key2,
      prefix_sum([1, 3, 8], 0)) ->
   wtree:put(ProcState, Key2, [1, 4, 12]).
```

  - Worker 1:

```
Leaf2(ProcState, 0) ->
   wtree:put(ProcState, Key2,
      prefix_sum(wtree:get(ProcState, Key1), 0)) ->
   wtree:put(ProcState, Key2,
      prefix_sum([-5, 11, 2], 12)) ->
   wtree:put(ProcState, Key2, [7, 18, 20]).
```

  - Workers 2–7: . . .

Let's Try It

```
2> W = wtree:create(8).
[<0.65.0>,<0.66.0>,<0.67.0>,<0.68.0>
 <0.69.0>,<0.70.0>,<0.71.0>,<0.72.0>]
3> workers:update(W, raw_data,
  [ [1,3,8],  [-5,11,2],  [17,0,-3], [100,-8,12],
    [4,19,1], [6,-168,7], [1,2,3],   [14,15]]).
ok
4> examples:prefix_sum_par(W, raw_data, cooked_data).  42
5> workers:retrieve(W, cooked_data).
[ [1,4,12], [7,18,20], "%%\"", [134,126,138],
  [142,161,162], [168,0,7], "\b\n\r", "\e*"] 6> $37
```

- Likewise, `$" == 34`, `$== 8`, `$\n == 10`, `$\r == 13`, `$\e == 27`, and `$* == 42`.
- All is well.

More Examples of scan

- Account balance with interest:
  - Input: a list of transactions, where each transaction can be a deposit (add an amount to the balance), a withdrawal (subtract an amount from the balance), or interest (multiply the balance by an amount). For example:
`[{deposit, 100.00}, {withdraw, 5.43}, {withdraw, 27.75}`
  - Output: the account balance after each transaction. For example, if we assume a starting balance of $1000.00 in the previous example, we get
`[1100.00, 1094.57, 1066.82, 1067.40, ...]`
- Delete 3s
  - Given a list that is distributed across *NProc* processes, delete all 3s, and rebalance the list so each process has roughly the same length sublisth.
  - Solution (sketch):
    - ★ Using scan, each process determines how many 3s preceed its segment, the total list length preceeding it, and the total list length after deleting 3s.
    - ★ Each process deletes its 3s and send portions of its lists and/or receives list portions to rebalance.

More^2 Examples of scan

- Carry-Lookahead Addition:
  - Given two large integers as a list of bits (or machine words), compute their sum.
    - ★ Note that the "pencil-and-paper" approach works from the least significant bit (or digit, or machine word) and works sequentially to the most-significant bit. This takes $O(N)$ time where N is the number of bits in the work.
  - Carries can be computed using scan.
    - ★ This allows a parallel implementation that adds two integers in $O(\log N)$ time.
    - ★ This is how the hardware in your CPU does addition – the adder takes $O(\log N)$ gate delays to add two, machine words, where N is the number of bits in a word.
- See *Principles of Parallel Programming*, pp. 119f.
- See homework 2 (later today, I hope).

Preview

January 23: Architecture Review	
Reading:	Pacheco, Chapter 2, Sections 2.1 and 2.2.
January 25: Shared-Memory Machines	
Reading:	Pacheco, Chapter 2, Section 2.3
January 27: Distributed-Memory Machines	
Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
Mini Assignments	Mini 4 goes out.
January 30: Parallel Performance: Speed-up	
Reading:	Pacheco, Chapter 2, Section 2.6.
Homework:	HW 2 earlybird (11:59pm). HW 3 goes out.
February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 3 due (10am)
February 6: Parallel Performance: Wrap Up	
January 8–February 15: Parallel Sorting	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

Review Questions

- What is scan? Give an example.
- Compare scan with `lists:mapfoldl`?
- What property must an operator have to be amenable use with scan?
- What are the components of a generalized scan?
As an example, what functions do you need to define to use `wtree:scan`?
- Consider the following variations on the bank account problem:
  - Add a transaction `{reset, Balance}`, where `Balance` is a number. The account balance is set to this amount. For example, this can be used to open an account with an initial balance. We'll also assume that a `reset` can be done at any point in a sequence of transactions.
  - Change interest computations so that the bank charges a daily interest of X% for negative balances, neither charges nor pays interest for positive balances less than $1000, and pays a daily interest of Y% for positive balances greater than $1000.
  - For each of these:
    - ★ Can the account balance still be computed using scan?
    - ★ If yes, explain how to do. If no, explain why it's not possible.

Computer Architecture Review

Mark Greenstreet

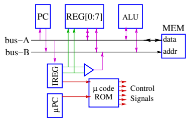
CpSc 418 – Jan. 23, 2017

- A microcoded machine
- A pipelined machine: RISC
- Let's write some code
- Superscalars and the memory bottleneck

Objectives

- Review classical, sequential architectures
  - a simple microcoded, machine
  - a pipelined, one-instruction per clock cycle machine
- Pipelining **is** parallel execution
  - the machine is supposed to appear (nearly) sequential
  - introduce the ideas of hazards and dependencies.

Microcoded machines



A simple, microcoded machine

- The microcode (μcode) ROM specifies the sequence of operations necessary to carry out an instruction.
- For simplicity, I'm assuming that the op-code bits of the instruction form the most significant bits of the μcode ROM address, and that the value of the micro-PC (μPC) form the lower half of the address.

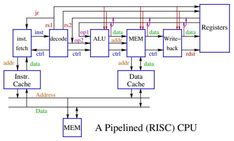
Microcode: summary

- Separates hardware from instruction set.
  - Different hardware can run the same software.
  - Enabled IBM to sell machines with a wide range of performance that were all compatible
    - □ I.e. IBM built an empire and made a fortune on the IBM 360 and its successors.
    - □ Intel has done the same with the x86.
- But, as implemented on slide 3, it's very sequential.

```
while(true) {
   fetch an instruction;
   perform the instruction
}
```

- Instruction fetch is "overhead"
  - Motivates coming up with complicated instructions that perform lots of operations per instruction fetch.
  - But these are hard for compilers to use.
  - Can we do better?

Pipelined instruction execution



A Pipelined (RISC) CPU

- Successive instructions in each stage
- When instruction `i` in `ifetch`, instruction `i-1` in decode, . . .
- Allows throughput of one instruction per cycle.
- Favors simple instructions that execute on a single pass through the pipeline.
  - This is known as RISC: "Reduced Instruction Set Computer"
  - A modern x86 is CISC on the outside, but RISC on the inside.

What about Dependencies?

- Multiple-instructions are in the pipeline at the same time.
- An instruction starts before all of its predecessors have completed.
- Data hazards occur if
  - an instruction can read a different value than would have been read with a sequential execution of instructions,
  - or if a register or memory location is left holding a different value than it would have had in a sequential execution.
- Control hazards occurs if
  - an instruction is executed that would not have been executed in a sequential execution.
  - This is because the instruction "depends" on a jump or branch that hasn't finished in time.

Handling Hazards

- Bypass: If an instruction has a result that a later instruction needs, the earlier instruction can provide that result directly without waiting to go through the register file.
- Move common operations early:
  - Decide branches in decode stage
  - ALU operations in the stage after decode
  - Memory reads take longer, but they happen less often.
- Let the compiler deal with it
- If nothing else helps, stall.

Break for Live Coding

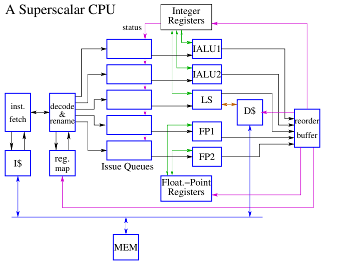
Back to Architecture

- the microcoded machine takes 5+ clock-cycles per instruction.
- the RISC machine takes 1 clock-cycle per instruction – in the best case:
  - There can be stalls due to cache misses,
  - unfilled delay slots, or
  - multi-cycle operations.
- Can we break the one-cycle-per instruction barrier?

The Memory Bottleneck

- A CPU core can execute roughly one instruction per clock-cycle.
  - With a 3GHz clock, that's roughly 0.3ns per instruction.
- Main memory accesses take 60-200ns (or longer)
  - That's 200-600 instructions per main memory access.
- Why?
  - CPUs designed for speed.
  - Memory designed for capacity:
    - □ fast memories are small
    - □ large memories are slow

Superscalar Processors



Superscalar Execution

- Fetch several, W, instructions each cycle.
- Decode them in parallel, and send them to issue queues for the appropriate functional unit.
- But what about dependencies?
  - We need to make sure that data and control dependencies are properly observed.
  - Code should execute on a superscalar as if it were executing on sequential, one-instruction-at-a-time machine.
  - Data dependencies can be handled by "**register renaming**" – this uses register indices to dynamically create the dependency graph as the program runs.
  - Control dependencies can be handled by "**branch speculation**" – guess the branch outcome, and rollback if wrong.
- The opportunity to execute instructions in parallel is called **Instruction Level Parallelism**, ILP.

What superscalars are good at

- Scientific computing:
  - often successive loop iterations are independent
  - the superscalar **pipelines** the loop
  - Perform memory reads for loop i, while doing multiplications for loop i-2, while doing additions for loop i-4, while storing the results for loop i-5.
- Commercial computing (databases, webservers, ...)
  - often have large data sets and high cache miss rates.
  - the superscalar can find executable instructions after a cache miss.
  - if it encounters more misses, the CPU benefits from **pipelined** memory accesses.
- Burning lots of power
  - many operations in a superscalar require hardware that grows quadratically with W.
  - basically, all instructions in a batch of W have to compare there register indices with all of the other ones.

Superscalar Reality

- Most general purpose CPUs (x86, Arm, Power, SPARC) are superscalar.
- Register renaming works **very** well:
- Branch prediction is also very good, often $>$ 90% accuracy.
  - But, data dependent branches can cause very poor performance.
- Superscalar designs make multi-threading possible
  - The features for executing multiple instruction in parallel work well for mixing instructions from several threads or processes – this is called "multithreading" (or "hyperthreading", if you're from Intel).
  - In practice, superscalars are often *better at multithreading* than they are at extracting ILP from a sequential program.

Preview

January 25: Shared-Memory Machines	
Reading:	Pacheco, Chapter 2, Section 2.3
January 27: Distributed-Memory Machines	
Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
Mini Assignments	Mini 4 goes out.
January 30: Parallel Performance: Speed-up	
Reading:	Pacheco, Chapter 2, Section 2.6.
Homework:	HW 2 earlybird (11:59pm). HW 3 goes out.
February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 3 due (10am)
February 6: Parallel Performance: Wrap Up	
January 8–February 15: Parallel Sorting	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

Review

- How does a pipelined architecture execute instruction in parallel?
- What are hazards?
- What are dependencies?
- What is multithreading.
- For further reading on RISC:
  "Instruction Sets and Beyond: Computers, Complexity, and Controversy"
  R.P. Colwell, *et al.*, *IEEE Computer*, vol. 18, no. 3,
  - You can download the paper for free if your machine is on the UBC network.
  - If you are off-campus, you can use the library's proxy.

Shared Memory Multiprocessors

Mark Greenstreet

CpSc 418 – Jan. 25, 2017

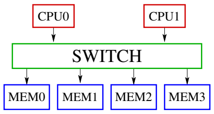
Outline:

- Shared-Memory Architectures
- Memory Consistency
- Coding Break
- Weak Consistency

Objectives

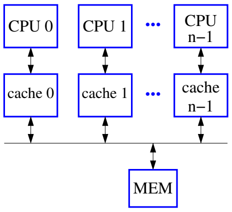
- Understand how processors can communicate by sharing memory.
- Able to explain the term "sequential consistency"
  - Describe a simple cache-coherence protocol, MESI
  - Describe how the protocol can be implemented by snooping.
  - Describe "sequential consistency".
  - Be aware that real machines make guarantees that are weaker than sequential consistency.

An Ancient Shared-Memory Machine



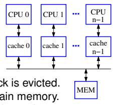
- Multiple CPU's (typically two) shared a memory
- If both attempted a memory read or write at the same time
  - One is chosen to go first.
  - Then the other does its operation.
  - That's the role of the switch in the figure.
- By using multiple memory units (partitioned by address), and a switching network, the memory could keep up with the processors.
- But, now that processors are 100's of times faster than memory, this isn't practical.

A Shared-Memory Machine with Caches



- Caches reduce the number of main memory reads and writes.
- But, what happens when a processor does a write?

Cache Inconsistency

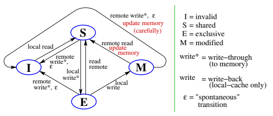


- Assume caches are write-back:
  - **write-back**: writes only update the cache. Main memory updated when the cache block is evicted.
  - **write-through**: writes update cache and main memory.
  - Modern processors have to use write-back for performance: Main memory is way too slow for write-through.
- Step 0: CPU 0 and CPU 1 have both read memory location addr0 and addr1 and have copies in their cache.
- Step 1: CPU 0 writes to addr0 and CPU 1 writes to addr1.
- Step 2: CPU 0 reads from addr1 and CPU 1 reads to from addr0.
  - Both CPUs see the **old** value.
  - The writes only updated the writer's cache.
  - The readers got the old values.

Cache Coherence Protocols

- **Big idea:** caches communicate with each other so that:
  - Multiple CPUs can have read-only copies for the same memory location.
  - If a cache has a dirty block, then no other cache has a copy of that block.

The MESI protocol

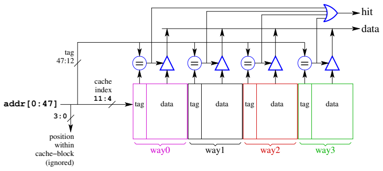


- Caches can share read-only copies of a cache block.
- When a processor writes a cache block, the first write goes to main memory.
  - The other caches are notified and invalidate their copies.
  - This ensures that writeable blocks are exclusive.

How caches work

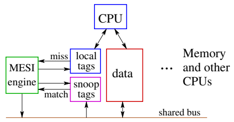
- Caching rhymes with hashing and the two ideas are similar.
  - Caches store data in "blocks" – the block size is a small power-of-two times the machine word size.
  - A cache has one or more "ways" – each way holds a power-of-two number number of blocks.
  - A hash-value is computed from the address.
    - ★ `blockAddr = addr / blockSize; %` right shift
    - ★ `blockIndex = blockAddr % (BlocksPerWay-1); %` bit masking
- Read:
  - The `blockIndex` is used to look up one entry in each "way".
  - Each block has a tag that includes the full-address for the data stored in that block.
  - The tags from each way are compared with the tag of the address:
    - ★ If any tag matches, that way provides the data.
    - ★ If no tags match, then a cache miss occurs.
    - ★ Some current block is evicted from the cache to make room for the incoming block.
- Writes are similar to reads.

A typical cache



- Only the read-path is shown. Writing is similar.
- This is a 16K-byte, 4-way set-associative cache, with 16 byte cache blocks.

Implementing MESI: Snooping



- Caches read and write main memory over a shared memory bus.
- Each cache has two copies of the tags: one for the CPU, the other for the bus.
- If the cache sees another CPU reading or writing a block that is in this cache, it takes the action specified by the MESI protocol.

Implementing MESI: Directories

- Main memory keeps a copy of the data and
  - a bit-vector that records which processors have copies, and
  - a bit to indicate that one processor has a copy and it may be modified.
- A processor accesses main memory as required by the MESI protocol.
  - The memory unit sends messages to the other CPUs to direct them to take actions as needed by the protocol.
  - The ordering of these messages ensures that memory stays consistent.
- Comparison:
  - Snooping is simple for machines with a small number of processors.
  - Directory methods scale better to large numbers of processors.

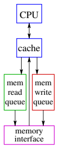
Sequential Consistency

Memory is said to be sequentially consistent if

- All memory reads and writes from all processors can be arranged into a single, sequential order, such that:
  - The operations for each processor occur in the global ordering in the same order as they did on the processor.
  - Every read gets the value of the preceding write to the same address.
- Sequential consistency corresponds to what programmers think "ought" to happen.
  - Very similar to "serializability" for database transactions.
- MESI guarantees sequential consistency

Coding Break

Weak Consistency



- CPUs typically have "write-buffers" because memory writes often come in bursts.
- Typically, reads can move ahead of writes to maximize program performance.
- Why?
  - Because there may be instructions waiting for the data from a load.
  - A transition from "shared" to "modified" requires notifying all processors – this can take a long time.
  - Memory writes don't happen until the instruction commits.
- This means that real computers don't guarantee sequential consistency.
  - Warning: classical algorithms for locks and shared buffers fail when run on a real machines!

Programming Shared Memory Machines

- Shared memory make parallel programming "easier" because:
  - One thread can pass an entire data structure to another thread just by giving a pointer.
  - No need to pack-up trees, graphs, or other data structures as messages and unpack them at the receiving end.
- Shared memory make parallel programming harder because:
  - It's easy to overlook synchronization (control to shared data structures). Then, we get data races, corrupted data structures, and other hard-to-track-down bugs.
  - A defensive reaction is to wrap *every* shared reference with a lock. But locks are slow (that λ factor for communication), and this often results is slow code, or even deadlock.
- In practice, shared memory code that works often has a message-passing structure.
- Finally, beware of weak consistency
  - Use a thread library.
  - There are elegant algorithms that avoid locking overhead, even with weak consistency, but they are beyond the scope of this class.

Shared Memory and Performance

- Shared memory can offer better performance than message passing because
  - High bandwidth: the buses that connect the caches can be very wide, especially if the caches are on a single chip.
  - Low latency: the hardware handles moving the data – no operating system calls and context-switch overheads.
- But, shared memory doesn't scale as well as message passing
  - For large machines, the latency of directory accesses can severely degrade performance.
    - ★ In a message passing machine, each CPU has its own memory, nearby and fast.
    - ★ For shared memory, each CPU has part of the shared main memory – accessing a directory may require accessing the memory of a distant CPU.
  - Shared memory moves the data after the cache miss
    - ★ this stalls a thread
    - ★ message passing can send data in advance and avoid these stalls

Summary

- Shared-Memory Architectures
  - Use cache-coherence protocols to allow each processor to have its own cache while maintaining (almost) the appearance of having one shared memory for all processors.
    - ★ A typical protocol: MESI
    - ★ The protocol can be implemented by snooping or directories.
  - Using cache-memory interconnect for interprocessor communication provides:
    - ★ High-bandwidth
    - ★ Low-latency, but watch out for fences, etc.
    - ★ High cost for large scale machines.
- Shared-Memory Programming
  - Need to avoid interference between threads.
    - ★ Assertional reasoning (e.g. invariants) are crucial, much more so than in sequential programming.
    - ★ There are too many possible interleavings to handle intuitively.
    - ★ In practice, we don't formally prove complete programs, but we use the ideas of formal reasoning.
  - Real computers don't provide sequential consistency.
    - ★ Use a thread library.

Preview

January 27: Distributed-Memory Machines	
Reading:	Pacheco, Chapter 2, Sections 2.4 and 2.5.
Mini Assignments	Mini 4 goes out.
January 30: Parallel Performance: Speed-up	
Reading:	Pacheco, Chapter 2, Section 2.6.
Homework:	HW 2 earlybird (11:59pm). HW 3 goes out.
February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 4 due (10am)
February 6: Parallel Performance: Wrap Up	
January 8–February 15: Parallel Sorting	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

Review

- What is sequential consistency?
- Using the MESI protocol, can multiple processors simultaneously have entries in their caches for the same memory address?
- Using the MESI protocol, can multiple processors simultaneously modify entries in their caches for the same memory address?
- How can a cache-coherence protocol be implemented by snooping?
- How can a cache-coherence protocol be implemented using directories?
- What is false sharing (in the reading, but not covered in these slides)?
- Do real machines provide sequential consistency?
- How do these issues influence good software design practice?

Classifying Cache Misses

- **Compulsory:** The first reference to a cache block will cause a miss.
  - Note that the first access should be a write – otherwise the location is uninitialized.
  - A cache can avoid stalling the processor by using "allocate on write".
  - If a miss is a write, assign a block for the line, start the main memory read, track which bytes have been written, and merge with the data from memory when it arrives.
- **Capacity:** The cache is not big enough to hold all of the data used by the program.
- **Conflict:** Many active memory locations map to the same cache index.
  - If there are more such references than the associativity of the cache, these will cause conflict misses.
- **Coherence:** A cache block was evicted because another CPU was writing to it.
  - A subsequent read incurs a cache miss.

Cache Design Trade-Offs (1 of 2)

- **Capacity:** Larger caches have lower miss rates, but longer access times. This motivates using multiple levels of caches.
  - L1: closest to the CPU, smallest capacity (16-64Kbytes), fastest access (1-3 clock cycles).
  - L2: typically 128Kbytes to 1Mbyte, 5-10 cycle access time.
  - L3: becoming common, several Mbytes of capacity.
- **Block Size:**
  - Larger blocks can lower miss rate by exploiting spatial locality.
  - Larger blocks can raise miss rate due to conflict and coherence misses.
  - Larger blocks increase miss penalty by requiring more time to transfer all that data.
  - Typical block sizes are 16 to 256 bytes – sometimes block size changes with cache level.

Cache Design Trade-Offs (2 of 2)

- **Associativity:**
  - Increasing associativity generally reduces the number of conflict misses.
  - Increasing associativity makes the cache hardware more complicated.
  - Typical caches are direct mapped to four- or eight-way associative.
  - Associativity doesn't need to be a power of two!
- **Other stuff**
  - cache inclusion: is everything in the L1 also in the L2?
  - interaction with virtual memory: are cache addresses virtual or physical?
  - coherence protocol details:
    Example, Intel uses MESIF, the "F" stands for "forwarding". If a processor has a read miss, and another cache has a copy, one of the caches with a copy will be the "forwarding cache". The forwarding cache provides the data because it's much faster than main memory.
  - error detection and creation – caches $+$ cosmic rays $=$ flipped bits.
  - and all kinds of other optimizations that are beyond the scope of this class.

False Sharing

- False sharing occurs when two CPUs are actively writing different words in the same cache block.
  - Each write forces the other CPU to invalidate its cache block.
  - Each read forces the other CPU to change its cache block from `modified` or `exclusive` to `shared`.
- Example: count 3s
  - Here's an implementation with awful performance.
  - We create a global array of `int`s to hold the accumulators for each process.
  - Each time a process finds a `3`, it writes to its element in the array.
  - This forces the other CPUs whose accumulators are in the same block to invalidate their cache entry.
  - This turns accumulator accesses into main memory accesses.
  - And these accesses are serialized: one CPU at a time.

Message Passing Computers

Mark Greenstreet

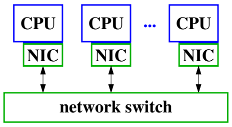
CpSc 418 – Jan. 27, 2017

Outline:
- Network Topologies
- Performance Considerations
- Examples

Objectives

- Familiar with typical network topologies:
  rings, meshes, crossbars, tori, hypercubes, trees, fat-trees.
- Understand implications for programming
  - bandwidth bottlenecks
  - latency considerations
  - location matters
  - heterogeneous computers.

Message Passing Computers



- Multiple CPU's
- Communication through a network:
  - Commodity networks for small clusters.
  - Special high-performance networks for super-computers
- Programming model:
  - Explicit message passing between processes (like Erlang)
  - No shared memory or variables.

Some simple message-passing clusters

- 25 linux workstations (e.g. lin01 . . . lin25.ugrad.cs.ubc.ca) and standard network routers.
  - A good platform for learning to use a message-passing cluster.
  - But, we'll figure out that network bandwidth and latency are key bottlenecks.
- A "blade" based cluster, for example:
  - 16 "blades" each with 4 6-core CPU chips, and 32G of DRAM.
  - An "infiniband" or similar router for about 10-100 times the bandwidth of typical ethernet.
  - The price tag is ∼\$300K.
    - ★ Great if you need the compute power.
    - ★ But, we won't be using one in this class.

The Sunway TaihuLight

- The world's fastest (Linpack) super-computer (as of June 2016)
- 40,960 multicore CPUs
  - 256 cores per CPU chip.
  - 1.45GHz clock frequency, 8 flops/core/cycle.
- Total of 10,485,760 cores
- LINPACK performance: 93 PFlops
- Power consumption 15MW (computer) + cooling (unspecified)
- Tree-like
  - Five levels of hierarchy.
  - Each level has a high-bandwidth switch.
  - Some levels (all?) are fully-connected for that level.
- Programming model: A version linux with MPI tuned for this machine.
- For more information, see *Report on the Sunway TaihuLight System*, J. Dongarra, June 2016.

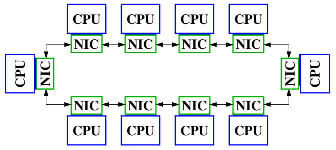
The Westgrid Clusters

- Clusters at various Western Canadian Universities (including UBC).
- Up to 9600 cores.
- Available for research use.

Network Topologies

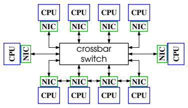
- Network topologies are to the message-passing community what cache-coherence protocols are to the shared-memory people:
  - **Lots** of papers have been published.
  - Machine designers are always looking for better networks.
  - Network topology has a strong impact on performance, the programming model, and the cost of building the machine.
- A message-passing machine may have multiple networks:
  - A general purpose network for sending messages between machines.
  - Dedicated networks for reduce, scan, and synchronization:
    - ★ The reduce and scan networks can include ALUs (integer and/or floating point) to perform common operations such as sums, max, product, all, any, etc. in the networking hardware.
    - ★ A synchronization network only needs to carry a few bits and can be designed to minimize latency.

Ring-Networks



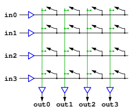
- Advantages: simple.
- Disadvantages:
  - Worst-case latency grows as $O(P)$ where P is the number of processors.
  - Easily congested – limited bandwidth.

Star Networks

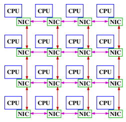


- Advantages:
  - Low-latency – single hop between any two nodes
  - High-bandwidth – no contention for connections with different sources and destinations.
- Disadvantages:
  - Amount of routing hardware grows as $O(P^2)$.
  - Requires lots of wires, to and from switch –
    Imagine trying to build a switch that connects to 1000 nodes!
- Summary
  - Surprisingly practical for 10-50 ports.
  - Hierarchies of cross-bars are often used for larger networks.

A crossbar switch

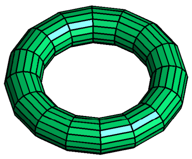


Meshes



- Advantages:
  - Easy to implement: chips and circuit boards are effectively two-dimensional.
  - Cross-section bandwidth grow with number of processors – more specifically, bandwidth grows as $\sqrt{P}$.
- Disadvantages:
  - Worst-case latency grows as $\sqrt{P}$.
  - Edges of mesh are "special cases."

Tori



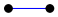
- Advantages:
  - Has the good features of a mesh, and
  - No special cases at the edges.
- Disadvantages:
  - Worst-case latency grows as $\sqrt{P}$.

Hypercubes

A 0−dimensional (1 node), radix−2 hypercube

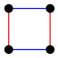
Hypercubes

A 1−dimensional (2 node), radix−2 hypercube



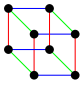
Hypercubes

A 2−dimensional (4 node), radix−2 hypercube



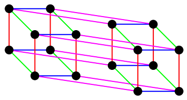
Hypercubes

A 3−dimensional (8 node), radix−2 hypercube



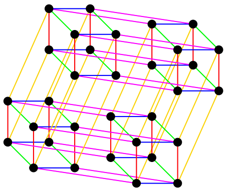
Hypercubes

A 4−dimensional (16 node), radix−2 hypercube



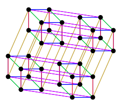
Hypercubes

A 5−dimensional (32 node), radix−2 hypercube



Hypercubes

A 5−dimensional (32 node), radix−2 hypercube

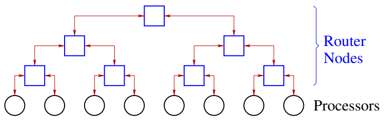


- Advantages
  - Small diameter (log N)
  - Lots of bandwidth
  - Easy to partition.
  - Simple model for algorithm design.
- Disadvantages
  - Needs to be squeezed into a three-dimensional universe.
  - Lots of long wires to connect nodes.
  - Design of a node depends on the size of the machine.

Dimension Routing

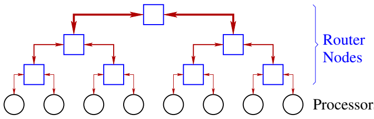
```
% Send a message, msg, from node src to node dst
for i = 1:d                               % d is dimension of the hypercube
  if(bit(i, src) != bit(i, dst))          % if different for dimension i
    send(msg, link[i]);                   % then send msg to our i-neighbour
```

Trees



- Simple network: number of routing nodes = number of processors − 1.
- Wiring: $O(\log N)$ extra height ($O(N \log N)$) extra area.
  - Wiring: $O(\sqrt{N} \log N)$ extra area for H-tree.
- Low-latency: $O(\log N) + \text{wire delay}$.
- Low-bandwidth: bottleneck at root.

Fat-Trees



- Use M^α parallel links to connect subtrees with M leaves.
- $0 \le \alpha \le 1$
  - $\alpha = 0$: simple tree
  - $\alpha = 1$: strange crossbar
- Fat-trees are "universal"
  - For $\frac{2}{3} < \alpha < 1$ a fat-tree interconnect with volume V can simulate any interconnect that occupies the same volume with a time overhead that is poly-log factor of N.

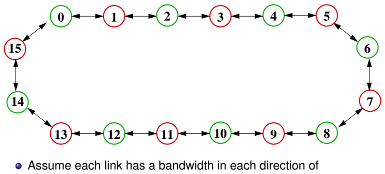
Performance Considerations

- Bandwidth
  - How many bytes per-second can we send between two processors?
    - ★ May depend on which two processors: neighbours may have faster links than spanning the whole machine.
  - Bisection bandwidth: find the worst way to divide the processors into to sets of $P/2$ processors each.
    - ★ How many bytes per-second can we send between the two partitions?
    - ★ If we divide this by the number of processors, we typically get a much smaller value that the peak between two processors.
- Latency
  - How long does it take to send a message from one processor to another?
    - ★ Typically matters the most for short messages.
    - ★ Round-trip time is often a good way to measure latency.
- Cost
  - How expensive is the interconnect – it may dominate the total machine cost.
    - ★ Cost of the network interface hardware.
    - ★ Cost of the cables.

Real-life networks

- InfiniBand is becoming increasingly prevalent
- Peak bandwidths $\geq$ 6GBytes/sec.
  - achieved bandwidths of 2–3GB/s.
- Support for RDMA and "one-sided" communication
  - CPU A can read or write a block of memory residing with CPU B.
- Often, networks include trees for synchronization (e.g. barriers), and common reduce and scan operations.
- The MPI (message-passing interface) evolves to track the capabilities of the hardware.

Bandwidth Matters



- Assume each link has a bandwidth in each direction of 1Gbyte/sec.
- Each node, i, sends an 8Kbyte message to node $(i+1) \bmod P$, where P is the number of processors?
- How long does this take?
- What if each node, i, sends an 8Kbyte message to node $(i+P/2) \bmod P$?

What this means for programmers

- Location matters.
  - The meaning of location depends on the machine.
  - Getting a good programming model is hard.
  - Challenges of heterogeneous machines.
- What it means for different kinds of computers
  - Supercomputers
  - Clouds
  - PCs of the future(?)

Summary

- Message passing machines have an architecture that corresponds to the message-passing programming paradigm.
- Message passing machines can range from
  - Clusters of PC's with a commodity switch.
  - Clouds: lots of computers with a general purpose network.
  - Super-computers: lots of compute nodes tightly connected with high-performance interconnect.
- Many network topologies have been proposed:
  - Performance and cost are often dominated by network bandwidth and latency.
  - The network can be more expensive than the CPUs.
  - Peta-flops or other instruction counting measures are an indirect measure of performance.
- Implications for programmers
  - Location matters
  - Communication costs of algorithms is very important
  - Heterogeneous computing is likely in your future.

Preview

January 30: Parallel Performance: Speed-up	
Reading:	Pacheco, Chapter 2, Section 2.6.
Homework:	HW 2 earlybird (11:59pm). HW 3 goes out.
February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 4 due (10am)
February 6: Parallel Performance: Wrap Up	
January 8–February 15: Parallel Sorting	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

- There will be readings assigned from *Programming Massively Parallel Processors* starting after the midterm.
- Make sure you have a copy. Note: this year, we'll make sure the course works with either the 2nd or 3rd edition.

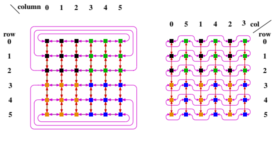
Review

- Consider a machine with 4096 processors.
- What is the maximum latency for sending a message between two processors (measured in network hops) if the network is
  - A ring?
  - A crossbar?
  - A 2-D mesh?
  - A 3-D mesh?
  - A hypercube?
  - A binary tree?
  - A radix-4 tree?

Supplementary Material

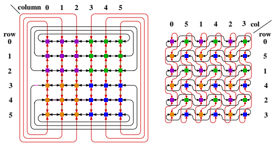
- Message-passing origami: how to fold a mesh into a torus.
- How big is a hypercube: it's all about the wires.

From a mesh to a torus (1/2)



- Fold left-to-right, and make connections where the left and right edges meet.
- Now, we've got a cylinder.
- Note that there are no "long" horizontal wires: the longest wires jump across one processor.

From a mesh to a torus (2/2)



- Fold top-to-bottom, and make connections where the top and bottom edges meet.
- Now, we've got a torus.
- Again there are no "long" wires.

How big is a hypercube?

- Consider a hypercube with $N = 2^d$ nodes.
- Assume each link can transfer one message in each direction in one time unit. The analysis here easily generalizes for links of higher or lower bandwidths.
- Let each node send a message to each of the other nodes.
- Using dimension routing,
  - Each node will send $N/2$ messages for each of the d dimensions.
  - This takes time $N/2$.
  - As soon as one batch of messages finishes the dimension-0 route, that batch can continue with the dimension-1 route, and the next batch can start the dimension 0 route.
  - So, we can route with a throughput of $\binom{N}{2}$ messages per $N/2$ time.

How big is a hypercube?

- Consider a hypercube with $N = 2^d$ nodes.
- Assume each link can transfer one message in each direction in one time unit. The analysis here easily generalizes for links of higher or lower bandwidths.
- Let each node send a message to each of the other nodes.
- Using dimension routing,

  we can route with a throughput of $\binom{N}{2}$ messages per $N/2$ time.
- Consider any plane such that $N/2$ nodes are on each side of the plane.
  - $\frac{1}{2}\binom{N}{2}$ messages must cross this plane in $N/2$ time.
  - This means that at least $N-1$ links must cross the plane.
  - The plane has area $O(N)$.

How big is a hypercube?

- Consider a hypercube with $N = 2^d$ nodes.
- Assume each link can transfer one message in each direction in one time unit. The analysis here easily generalizes for links of higher or lower bandwidths.
- Let each node send a message to each of the other nodes.
- Using dimension routing,

  we can route with a throughput of $\binom{N}{2}$ messages per $N/2$ time.
- Consider any plane such that $N/2$ nodes are on each side of the plane.
  - The plane has area $O(N)$.
- Because the argument applies for *any* plane, we conclude that the hypercube has diameter $O(\sqrt{N})$ and thus volume $O(N^{\frac{3}{2}})$.
- Asymptotically, the hypercube is all wire.

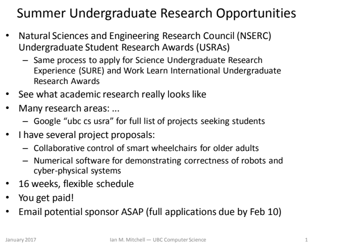
Speed-Up

Mark Greenstreet

CpSc 418 – Jan. 30, 2017

Outline:
- Measuring Performance
- Speed-Up
- Amdahl's Law
- The law of modest returns
- Superlinear speed-up
- Embarrassingly parallel problems

But first, USRA



Summer Undergraduate Research Opportunities

- Natural Sciences and Engineering Research Council (NSERC) Undergraduate Student Research Awards (USRAs)
  - Same process to apply for Science Undergraduate Research Experience (SURE) and Work Learn International Undergraduate Research Awards
- See what academic research really looks like
- Many research areas: ...
  - Google "ubc cs usra" for full list of projects seeking students
- I have several project proposals:
  - Collaborative control of smart wheelchairs for older adults
  - Numerical software for demonstrating correctness of robots and cyber-physical systems
- 16 weeks, flexible schedule
- You get paid!
- Email potential sponsor ASAP (full applications due by Feb 10)

January 2017 Ian M. Mitchell — UBC Computer Science 1

Objectives

- Understand key measures of performance
  - Time: latency vs. throughput
  - Time: wall-clock vs. operation count
  - Speed-up: slide 5
- Understand common observations about parallel performance
  - Amdahl's law: limitations on parallel performance (and how to evade them)
  - The law of modest returns: high complexity problems are bad, and worse on a parallel machine.
  - Superlinear speed-up: more CPUs ⇒ more, fast memory – and sometimes you win.
  - Embarrassingly parallel problems: sometimes you win, without even trying.

Measuring Performance

- The main motivation for parallel programming is performance
  - Time: make a program run faster.
  - Space: allow a program to run with more memory.
- To make a program run faster, we need to know how fast it is running.
- There are many possible measures:
  - Latency: time from starting a task until it completes.
  - Throughput: the rate at which tasks are completed.
  - Key observation:

$$\begin{array}{rcll} \textit{throughput} & = & \frac{1}{\textit{latency}}, & \text{sequential programming} \\ \textit{throughput} & \geq & \frac{1}{\textit{latency}}, & \text{parallel programming} \end{array}$$

Speed-Up

- Simple definition:

$$\textit{speed\_up} \quad = \quad \frac{\text{time(sequential\_execution)}}{\text{time(parallel\_execution)}}$$

- We can also describe speed-up as how many percent faster:

$$\%\textit{faster} \quad = \quad (\textit{speed\_up} - 1) * 100\%$$

- But beware of the spin:
  - Is "time" latency or throughput?
  - How big is the problem?
  - What is the sequential version:
    - The parallel code run on one processor?
    - The fastest possible sequential implementation?
    - Something else?
- More practically, how do we measure time?

Speed-Up – Example

- Let's say that count 3s of a million items takes 10ms on a single processor.
- If I run count 3s with four processes on a four CPU machine, and it takes 3.2ms, what is the speed-up?
- If I run count 3s with 16 processes on a four CPU machine, and it takes 1.8ms, what is the speed-up?
- If I run count 3s with 128 processes on a 32 CPU machine, and it takes 0.28ms, what is the speed-up?

Time complexity

- What is the time complexity of sorting?
  - What are you counting?
  - Why do you care?
- What is the time complexity of matrix multiplication?
  - What are you counting?
  - Why do you care?

Big-O and Wall-Clock Time

- In our algorithms classes, we count "operations" because we have some belief that they have something to do with how long the actual program will take to execute.
  - Or maybe not. Some would argue that we count "operations" because it allows us to use nifty techniques from discrete math.
  - I'll take the position that the discrete math is nifty because it tells us something useful about what our software will do.
- In our architecture classes, we got the formula:

$$\text{time} \quad = \quad \frac{(\#\text{inst. executed}) * (\text{cycles}/\text{instruction})}{\text{clock frequency}}$$

- The approach in algorithms class of counting comparisons or multiplications, etc., is based on the idea that everything else is done in proportion to these operations.
- BUT, in parallel programming, we can find that a communication between processes can take 1000 times longer than a comparison or multiplication.
  - This may not matter if you're willing to ignore "constant factors."
  - In practice, factors of 1000 are too big to ignore.

Amdahl's Law

- Given a sequential program where
  - fraction s of the execution time is inherently sequential.
  - fraction $1 - s$ of the execution time benefits perfectly from speed-up.
- The run-time on P processors is:

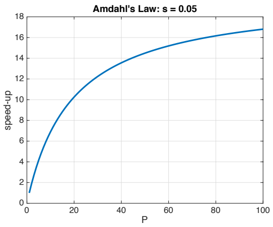
$$T_{\textit{parallel}} \quad = \quad T_{\textit{sequential}} * (s + \frac{1-s}{P})$$

- Consequences:
  - Define

$$\textit{speed\_up} \quad = \quad \frac{T_{\textit{sequential}}}{T_{\textit{parallel}}}$$

  - Speed-up on P processors is at most $\frac{1}{s}$.
  - Gene Amdahl argued in 1967 that this limit means that parallel computers are only useful for a few special applications where s is very small.

Amdahl's Law



Amdahl's Law, 49 years later

Amdahl's law is not a physical law.
- Amdahl's law is mathematical theorem:
  - If $T_{\textit{parallel}}$ is $\left(s + \frac{1-s}{P}\right) T_{\textit{sequential}}$
  - and $\textit{speed\_up} = T_{\textit{sequential}} / T_{\textit{parallel}}$,
  - then for $0 < s \leq 1$, $\textit{speed\_up} \leq \frac{1}{s}$.
- Amdahl's law is also an economic law:
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap
    - The cost of fabricating eight cores on a die is very little more that the cost of fabricating one.
    - Computer cost is dominated by the rest of the system: memory, disk, network, monitor, . . .
- Amdahl's law assumes a fixed problem size.

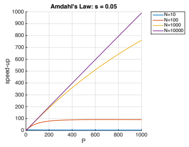
Amdahl's Law, 49 years later

- Amdahl's law is an economic law, not a physical law.
  - Amdahl's law was formulated when CPUs were expensive.
  - Today, CPUs are cheap (see previous slide)
- Amdahl's law assumes a fixed problem size
  - Many computations have s (sequential fraction) that decreases as N (problem size) increases.
  - Having lots of cheap CPUs available will
    - Change our ideas of what computations are easy and which are hard.
    - Determine what the "killer-apps" will be in the next ten years.
      - Ten years from now, people will just take it for granted that most new computer applications will be parallel.
  - Examples: see next slide

Amdahl's Law, 49 years later

- Amdahl's law is an economic law, not a physical law.
- Amdahl's law assumes a fixed problem size
  - Ten years from now, people will just take it for granted that most new computer applications will be parallel.
  - Examples:
    - Managing/searching/mining massive data sets.
    - Scientific computation.
      - Note that most of the computation for animation and rendering resembles scientific computation. Computer games benefit tremendously from parallelism.
      - Likewise for multimedia computing.

Amdahl's Law, one more try



- We can have problems where the parallel work grows faster than the sequential part.
- Example: parallel work grows as $N^{3/2}$ and the sequential part grows as $\log P$.

The Law of Modest Returns

More bad news. ☹
- Let's say we have an algorithm with a sequential run-time $T = (12\text{ns})N^4$.
  - If we're willing to wait for one hour for it to run, what's the largest value of N we can use?
  - If we have 10000 machines, and perfect speed-up (i.e. $\textit{speed\_up} = 10000$), now what is the largest value of N we can use?
  - What if the run-time is $(5ns)1.2^N$?
- The law of modest returns
  - Parallelism offers modest returns, unless the problem is of fairly low complexity.
  - Sometimes, modest returns are good enough: weather forecasting, climate models.
  - Sometimes, problems have huge N and low complexity: data mining, graphics, machine learning.

Super-Linear Speed-up

Sometimes, $\textit{speed\_up} > P$. ☺
- How does this happen?
  - Impossibility "proof": just simulate the P parallel processors with one processor, time-sharing P ways.
- Memory: a common explanation
  - P machines have more main memory (DRAM)
  - and more cache memory and registers (total)
  - and more I/O bandwidth, . . .
- Multi-threading: another common explanation
  - The sequential architecture underutilizes the parallel capabilities of the CPU.
  - A parallel algorithm can make better use.
- Algorithmic advantages: once in a while, you win!
  - Simulation as described above has overhead.
  - If the problem is naturally parallel, the parallel version can be more efficient.
- BUT: be very skeptical of super-linear claims, especially if $\textit{speed\_up} \gg P$.

Embarrassingly Parallel Problems

Problems that can be solved by a large number of processors with very little communication or coordination.
- Rendering images for computer-animation: each frame is independent of all the others.
- Brute-force searches for cryptography.
- Analyzing large collections of images: astronomy surveys, facial recognition.
- Monte-Carlo simulations: same model, run with different random values.
- **Don't be ashamed if your code is embarrassingly parallel:**
  - Embarrassingly parallel problems are great: you can get excellent performance without heroic efforts.
  - The only thing to be embarrassed about is if you don't take advantage of easy parallelism when it's available.

Lecture Summary

Parallel Performance
- Speed-up: slide 5
- Limits
  - Amdahl's Law, slide 9.
  - Modest gains, slide 15.
- Sometimes, we win
  - Super-linear speedup, slide 16.
  - Embarrassingly Parallel Problems, slide 17.

Preview

February 1: Parallel Performance: Overheads	
Homework:	HW 2 due (11:59pm).
February 3: Parallel Performance: Models	
Mini Assignments	Mini 4 due (10am)
February 6: Parallel Performance: Wrap Up	
February 8: Parallel Sorting – The Zero-One Principle	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 10: Bitonic Sorting (part 1)	
February 15: Family Day – no class	
February 13: Bitonic Sorting (part 2)	
Homework (Feb. 15):	HW 3 earlybird (11:59pm), HW 4 goes out.
February 17: Map-Reduce	
Homework:	HW 3 due (11:59pm).
February 27: TBD	
March 1: Midterm	

- Reading from "Programming Massively Parallel Computers" (D.B. Kirk & W.-M. Hwu) start right after the midterm. Make sure you have a copy.
- You can use either the 2nd or 3rd edition.

Review Questions

- What is speed-up? Give an intuitive, English answer and a mathematical formula.
- Why can it be difficult to determine the sequential time for a program when measuring speed-up?
- What is Amdahl's law? Give a mathematical formula. Why is Amdahl's law a concern when developing parallel applications? Why in many cases is it not a show-stopper?
- Is parallelism an effective solution to problems with high big-O complexity? Why or why not?
- What is super-linear speed-up? Describe two causes.
- What is an embarrassingly parallel problem. Give an example.

Performance-Loss

Mark Greenstreet

CpSc 418 – Feb. 1, 2017

Outline:
- Overhead: work the parallel code has to do that the sequential version avoids.
  - Communication and Synchronization
  - Extra computation, extra memory
- Limited parallelism
  - Code that is inherently sequential or has limited parallelism
  - Idle processors
  - Resource contention

Objectives

- Learn about main causes of performance loss:
  - Overhead
  - Non-parallelizable code
  - Idle processors
  - Resource contention
- See how these arise in message-passing, and shared-memory code.

Causes of Performance Loss

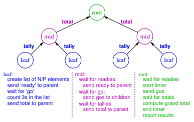
- Ideally, we would like a parallel program to run *P* times faster than the sequential version when run on *P* processors.
- In practice, this rarely happens because of:
  - Overhead: work that the parallel program has to do that isn't needed in the sequential program.
  - Non-parallelizable code: something that has to be done sequentially.
  - Idle processors: There's work to do, but some processor are waiting for something before they can work on it.
  - Resource contention: Too many processors overloading a limited resource.

Overhead

Overhead: work that the parallel program has to do that isn't needed in the sequential program.

- Communication:
  - The processes (or threads) of a parallel program need to communicate.
  - A sequential program has no interprocess communication.
- Synchronization.
  - The processes (or threads) of a parallel program need to coordinate.
  - This can be to avoid interference, or to ensure that a result is ready before it's used, etc.
  - Sequential programs have a completely specified order of execution: no synchronization needed.
- Computation.
  - Recomputing a result is often cheaper than sending it.
- Memory Overhead.
  - Each process may have its own copy of a data structure.

Communication Overhead



leaf:
create list of N/P elements
send 'ready' to parent
wait for 'go'
count 3s in the list
send total to parent

mid:
wait for readies:
send ready to parent
wait for go:
send gos to children
wait for tallies
send total to parent

root:
wait for readies
start timer
send gos
wait for totals
compute grand total
end timer
report results

- In a parallel program, data must be sent between processors.
- This isn't a part of the sequential program.
- The time to send and receive data is overhead.
- Communication overhead occurs with both shared-memory and message passing machines and programs.
- Example: Reduce (e.g. Count 3s):
  - Communication between processes adds time to execution.
  - The sequential program doesn't have this overhead.

Communication with shared-memory

- In a shared memory architecture:
  - Each core has it's own cache.
  - The caches communicate to make sure that all references from different cores to the same address look like there is one, common memory.
  - It takes longer to access data from a remote cache than from the local cache. This creates overhead.
- False sharing can create communication overhead even when there is no logical sharing of data.
  - This occurs if two processors repeatedly modify different locations on the same cache line.

Communication overhead: example

- The *Principles of Parallel Programming* book considered an example of Count 3s (in C, with threads), where there was a global array, `int count[P]` where `P` is the number of threads.
  - Each thread (e.g. thread *i*) initially sets its count, `count[i]` to 0.
  - Each time a thread encounters a `3`, it increments its element in the array.
- The parallel version ran much slower than the sequential one.
  - Cache lines are much bigger than a single `int`. Thus, many entries for the `count` array are on the same cache line.
  - A processor has to get exclusive access to update the count for its thread.
  - This invalidates the copies held by the other processors.
  - This produces lots of cache misses and a slow execution.
- A better solution:
  - Each thread has a local variable for its count.
  - Each thread counts its threes using this local variable and copies its final total to the entry in the global array.

Communication overhead with message passing

- The time to transmit the message through the network.
- There is also a CPU overhead: the time set up the transmission and the time to receive the message.
- The context switches between the parallel application and the operating system adds even more time.
- Note that many of these overheads can be reduced if the sender and receiver are different threads of the same process running on the same CPU.
  - This has led to SMP implementations of Erlang, MPI, and other message passing parallel programming frameworks.
  - The overheads for message passing on an SMP can be very close to those of a program that explicitly uses shared memory.
  - This allows the programmer to have one parallel programming model for both threads on a multi-core processor and for multiple processes on different machines in a cluster.

Synchronization Overhead

- Parallel processes must coordinate their operations.
  - Example: access to shared data structures.
  - Example: writing to a file.
- For shared-memory programs (e.g. `pthreads` or `Java threads`, there are explicit locks or other synchronization mechanisms.
- For message passing (e.g. `Erlang` or `MPI`), synchronization is accomplished by communication.

Computation Overhead

A parallel program may perform computation that is not done by the sequential program.

- Redundant computation: it's faster to recompute the same thing on each processor than to broadcast.
- Algorithm: sometimes the fastest parallel algorithm is fundamentally different than the fastest sequential one, and the parallel one performs more operations.

Sieve of Eratosthenes

To find all primes $\leq$ N:

```
1.   Let MightBePrime = [2, 3, ..., N].
2.   Let KnownPrimes = [].
3.   while(MightBePrime ≠ []) do
        % Loop invariant: KnownPrimes contains all primes less than the
        % smallest element of MightBePrime, and MightBePrime
        % is in ascending order. This ensure that the first element of
        % MightBePrime is prime.
3.1.    Let P = first element of MightBePrime.
3.2.    Append P to KnownPrimes.
3.3.    Delete all multiples of P from MightBePrime.
4.   end
```

See http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

Prime-Sieve in Erlang

```
% primes(N): return a list of all primes ≤ N.
primes(N) when is_integer(N) and (N < 2) -> [];
primes(N) when is_integer(N) ->
   do_primes([], lists:seq(2, N)).

% invariants of do_primes(Known, Maybe):
%   All elements of Known are prime.
%   No element of Maybe is divisible by any element of Known.
%   lists:reverse(Known) ++ Maybe is an ascending list.
%   Known ++ Maybe contains all primes ≤ N, where N is from p(N).
do_primes(KnownPrimes, []) -> lists:reverse(KnownPrimes);
do_primes(KnownPrimes, [P | Etc]) ->
do_primes([P | KnownPrimes],
          lists:filter(fun(E) -> (E rem P) /= 0 end, Etc)).
```

A More Efficient Sieve

- If *N* is composite, then it has at least one prime factor that is at most $\sqrt{N}$.
- This means that once we've found a prime that is $\geq \sqrt{N}$, all remaining elements of `Maybe` must be prime.
- Revised code:

```
% primes(N): return a list of all primes ≤ N.
primes(N) when is_integer(N) and (N < 2) -> [];
primes(N) when is_integer(N) ->
   do_primes([], lists:seq(2, N), trunc(math:sqrt(N))).

do_primes(KnownPrimes, [P | Etc], RootN)
      when (P =< RootN) ->
   do_primes([P | KnownPrimes],
      lists:filter(fun(E) -> (E rem P) /= 0 end, Etc), RootN);
   do_primes(KnownPrimes, Maybe, _RootN) ->
      lists:reverse(KnownPrimes, Maybe).
```

Prime-Sieve: Parallel Version

- Main idea
  - Find primes from $1 \ldots \sqrt{N}$.
  - Divide $\sqrt{N}+1 \ldots N$ evenly between processors.
  - Have each processor find primes in its interval.
- We can speed up this program by having each processor compute the primes from $1 \ldots \sqrt{N}$.
  - Why does doing extra computation make the code faster?

Memory Overhead

The total memory needed for *P* processes may be greater than that needed by one process due to replicated data structures and code.

- Example: the parallel sieve: each process had its own copy of the first $\sqrt{N}$ primes.

Overhead: Summary

Overhead is loss of performance due to extra work that the parallel program does that is not performed by the sequential version. This includes:

- Communication: parallel processes need to exchange data. A sequential program only has one process; so it doesn't have this overhead.
- Synchronization: Parallel processes may need to synchronize to guarantee that some operations (e.g. file writes) are performed in a particular order. For a sequential program, this ordering is provided by the program itself.
- Extra Computation:
  - Sometimes it is more efficient to repeat a computation in several different processes to avoid communication overhead.
  - Sometimes the best parallel algorithm is a different algorithm than the sequential version and the parallel one performs more operations.
- Extra Memory: Data structures may be replicated in several different processes.

Limited Parallelism

Sometimes, we can't keep all of the processors busy doing useful work.

- Non-parallelizable code
  The dependency graph for operations is narrow and deep.
- Idle processors
  There is work to do, but it hasn't been assigned to an idle processor.
- Resource contention
  Several processes need exclusive access to the same resource.

Non-parallelizable Code

- Finding the length of a linked list:

```
int length=0;
for(List p = listHead; p != null; p = p->next)
   length++;
```

  - Must dereference each `p->next` before it can dereference the next one.
  - Could make more parallel by using a different data structure to represent lists (some kind of skiplist, or tree, etc.)
- Searching a binary tree
  - Requires 2^k processes to get factor of *k* speed-up.
  - Not practical in most cases.
  - Again, could consider using another data structure.
- Interpreting a sequential program.
- Finite state machines.

Idle Processors

- There is work to do, but processors are idle.
- Start-up and completion costs.
- Work imbalance.
- Communication delays.

Resource Contention

.

- Processors waiting for a limited resource.
- It's easy to change a compute-bound task into an I/O bound one by using parallel programming.
- Or, we run-into memory bandwidth limitations:
  - Processing cache-misses.
  - Communication between CPUs and co-processors.
- Network bandwidth.

Lecture Summary

Causes of Performance Loss in Parallel Programs

- Overhead
  - Communication, slide 5.
  - Synchronization, slide 9.
  - Computation, slide 10.
  - Extra Memory, slide 15.
- Other sources of performance loss
  - Non-parallelizable code, slide 18
  - Idle Processors, slide 19.
  - Resource Contention, slide 20.

Review Questions

- What is overhead? Give several examples of how a parallel program may need to do more work or use more memory than a sequential program.
- Do programs running on a shared-memory computer have communication overhead? Why or why not?
- Do message passing program have synchronization overhead? Why or why not?
- Why might a parallel program have idle processes even when there is work to be done?

Models of Parallel Computation

Mark Greenstreet

CpSc 418 – Feb. 6, 2017

- The RAM Model of Sequential Computation
- Models of Parallel Computation
- An entertaining proof

Objectives

- Learn about models of computation
  - Sequential: Random Access Machine (RAM)
  - Parallel
    - ★ Parallel Random Access Machine (PRAM)
    - ★ Candidate Type Architecture (CTA)
    - ★ Latency-Overhead-Bandwidth-Processors (LogP)
- An entertaining algorithm and its analysis
  - If a model has invalid assumptions,
  - then we can show that algorithm 1 is faster than algorithm 2,
  - but in real life algorithm 2 is faster.
  - Valiant's algorithm also provides some mathematical entertainment.

The RAM Model

RAM = Random Access Machine

- Axioms of the model
  - Machines work on words of a "reasonable" size.
  - A machine can perform a "reasonable" operation on a word as a single step.
    - ★ such operations include addition, subtraction, multiplication, division, comparisons, bitwise logical operations, bitwise shifts and rotates.
  - The machine has an unbounded amount of memory.
    - ★ A memory address is a "word" as described above.
    - ★ Reading or writing a word of memory can be done in a single step.

The Relevance of the RAM Model

- If a single step of a RAM corresponds (to within a factor close to 1) to a single step of a real machine.
- Then algorithms that are efficient on a RAM will also be efficient on a real machine.
- Historically, this assumption has held up pretty well.
  - For example, `mergesort` and `quicksort` are better than `bubblesort` on a RAM and on real machines, and the RAM model predicts the advantage quite accurately.
  - Likewise, for many other algorithms
    - ★ graph algorithms, matrix computations, dynamic programming, ...
    - ★ hard on a RAM generally means hard on a real machine as well: NP complete problems, undecidable problems, ...

The Irrelevance of the RAM Model

The RAM model is based on assumptions that don't correspond to physical reality:

- Memory access time is highly non-uniform.
  - Architects make heroic efforts to preserve the illusion of uniform access time fast memory –
    - ★ caches, out-of-order execution, prefetching, . . .
  - – but the illusion is getting harder and harder to maintain.
    - ★ Algorithms that randomly access large data sets run much slower than more localized algorithms.
    - ★ Growing memory size and processor speeds means that more and more algorithms have performance that is sensitive to the memory hierarchy.
- The RAM model does not account for energy:
  - Energy is the critical factor in determining the performance of a computation.
  - The energy to perform an operation drops rapidly with the amount of time allowed to perform the operation.

The PRAM Model

PRAM = Parallel Random Access Machine

- Axioms of the model
  - A computer is composed of multiple processors and a shared memory.
  - The processors are like those from the RAM model.
    - ★ The processors operate in lockstep.
    - ★ I.e. for each $k > 0$, all processors perform their k^{th} step at the same time.
  - The memory allows each processor to perform a read or write in a single step.
    - ★ Multiple reads and writes can be performed in the same cycle.
    - ★ If each processor accesses a different word, the model is simple.
    - ★ If two or more processors try to access the same word on the same step, then we get a bunch of possible models:

      EREW: Exclusive-Read, Exclusive-Write
      CREW: Concurrent-Read, Exclusive-Write
      CRCW: Concurrent-Read, Concurrent-Write
      ★ See slide 25 for more details.

The Irrelevance of the PRAM Model

The PRAM model is based on assumptions that don't correspond to physical reality:

- Connecting *N* processors with memory requires a switching network.
  - Logic gates have bounded fan-in and fan-out.
  - $\Rightarrow$ any switch fabric with *N* inputs (and/or *N* outputs) must have depth of at least $\log N$.
  - This gives a lower bound on memory access time of $\Omega(\log N)$.
- Processors exist in physical space
  - *N* processors take up $\Omega(N)$ volume.
  - The processor has a diameter of $\Omega(N^{1/3})$.
  - Signals travel at a speed of at most *c* (the speed of light).
  - This gives a lower bound on memory access time of $\Omega(N^{1/3})$.

The CTA Model

CTA = Candidate Type Architecture

- Axioms of the model
  - A computer is composed of multiple processors.
  - Each processor has
    - Local memory that can be accessed in a single processor step (like the RAM model).
    - A small number of connections to a communications network.
  - There is a communication network connecting the processors.
    - The general model:
      - The communication network is a graph where all vertices (processors and switches) have bounded degree.
      - Each edge has an associated bandwidth and latency.
    - The simplified model:
      - Global actions have a cost of λ times the cost of local actions.
      - λ is assumed to be "large".
    - The exact communication mechanism is not specified.

The (Ir)Relevance of the CTA Model

- Recognizing that communication is expensive is the one, most important point to grasp to understand parallel performance.
  - CTA highlights the central role of communication.
  - PRAM ignores it.
- The general model is parameterized by the communication network
  - Can we apply results from analysing a machine with a 3-D toroidal mesh to a machine with fat trees?
  - PRAM ignores it.
- The simple model neglects bandwidth issues
  - Messages are assumed to be "small".
  - But, bigger messages often lead to better performance.
  - If we talk about bandwidth, do we mean the bandwidth of each link?
  - Or, do we mean the bisection bandwidth?

The LogP Model

- **Motivation (1993): convergence of parallel architectures**
  - Individual nodes have microprocessors and memory of a workstation or PC.
  - A large parallel machine had at most 2000 such nodes.
  - Point-to-point interconnect –
    - Network bandwidth much lower than memory bandwidth.
    - Network latency much higher than memory latency.
    - Relatively small network diameter: 5 to 20 "hops" for a 1000 node machine.
- **The model parameters:**

L	the latency of the communication network fabric
o	the overhead of a communication action
g	the bandwidth of the communication network
P	the number of processors

Why does **g** stand for "bandwidth"?

Marketing!

- What if we used **b** for "bandwidth"?
- Need a catchy acronym with 'ℓ', 'o', 'b', and 'p' . . .
  - got it: **BLOP**
  - but the marketing department vetoed it.

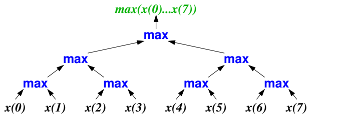
logP in practice

- The authors got some surprisingly good performance prediction for a few machines and a few algorithms by finding the "right" values for ℓ, o, g, and P for each architecture.
- It's rare to get a model that comes to within 10-20% on several examples. So, this looked very promising.
- Since then, logP seems to be a model with more parameters than simplified CTA, but not particularly better accuracy.
- Good to know about, because if you meet an algorithms expert, they'll probably know that PRAM is unrealistic.
  - Then, you'll often hear "What about logP"? – the paper has lots of citations.
  - In practice, it's a slightly fancier was of saying "communication costs matter".

Fun with the PRAM Model

Finding the maximum element of an array of $N$ elements.

- The obvious approach
  - Do a reduce.
  - Use $N/2$ processors to compute the result in $\Theta(\log_2 N)$ time.



A Valiant Solution

L. Valiant, 1975

- Use $N$ processors.
- The big picture:
  - Initially, we can use clumps of three processors to find the largest of three elements in $O(1)$ time – just do all three comparisons.
  - Now, we have $N/3$ elements but we still have $N$ processors. We can perform all of the comparisons for larger clusters of elements in $O(1)$ time in a single step because we have more processors per element.
  - Valiant showed that the size of a cluster for which we can do all of the pair-wise comparisons in a single step grows as 2^{k^2} where k is the number of steps.
  - This leads to a $\log\log N$ time bound for finding the max.
- I'll sketch the proof.
- Then we'll look at why this shows that you can't actually build a PRAM.

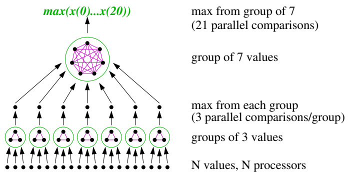
Valiant's algorithm, step 1

- Step 1:
  - Divide the $N$ elements into $N/3$ sets of size 3.
  - Assign 3 processors to each set, and perform all three pairwise comparisons in parallel.
  - Mark all the "losers" (requires a CRCW PRAM) and move the max of each set of three to a fixed location.
- The PRAM operations in a bit more detail.
  - Initially, every element has a flag set to `1` that says "might be the max".
  - When $\binom{k}{2}$ processors perform all of the pairwise comparisons of k values,
    - Each processor sets the flag for the smaller value to `0`.
    - Note that several processors may write `0` to the same location, but the CRCW allows this because they are all writing the same value.
  - One processor for each value checks if its flag is still set to `1`.
    - The winner for the cluster is moved to a specific location;
    - The flag for that location is set to `1`
    - And now we're ready for subsequent rounds.

Valiant's algorithm, step 2

- We now have $N/3$ elements left and still have $N$ processors.
- We can make groups of 7 elements, and have 21 processors per group, which is enough to perform all $\binom{7}{2} = 21$ pairwise comparisons in a single step.
- Thus, in $O(1)$ time we move the max of each set to a fixed location. We now have $N/21$ elements left to consider.

Visualizing Valiant



Valiant's Algorithm, the remaining steps

- On step k, we have N/m_k elements left.
- On step m_k is the "sparsity" of the problem – i.e. the number of processors per remaining element.
- We can make groups of $2m_k + 1$ elements, and have

$$\begin{aligned} m_k(2m_k+1) &= \tfrac{(2m_k+1)((2m_k+1)-1)}{2} \\ &= \binom{2m_k+1}{2} \end{aligned}$$

  processors per group, which is enough to perform all pairwise comparisons in a single step.
- We now have $N/(m_k(2m_k+1))$ elements to consider.
- Therefore, $m_{k+1} = 2m_k^2 + m_k$.
  - The sparsity is squared at each step.
  - It follows that the algorithm requires $O(\log\log N)$.
  - Valiant showed a matching lower bound and extended the results to show merging is $\theta(\log\log N)$ and sorting is $\theta(\log N)$ on a CRCW PRAM.
  - See slide 26 to see the details of the first few rounds.

Valiant's Algorithm, run-time

- The sparsity is roughly squared at each step.
- It follows that the algorithm requires $O(\log\log N)$.
- Valiant showed a matching lower bound and extended the results to show merging is $\theta(\log\log N)$ and sorting is $\theta(\log N)$ on a CRCW PRAM.
- See slide 27 for the details.

Take-home message from Valiant's algorithm

- The PRAM model is simple, and elegant, and many clever algorithms have been designed based on the PRAM model.
- It is also physically unrealistic:
  - As shown on slide 7, logic gates have bounded fan-in and fan-out.
  - Implementing the processor to memory interconnect requires a logic network of depth $\Omega(\log P)$.
  - Therefore, access time must be $\Omega(\log P)$.
  - Each step of the PRAM must take $\Omega(\log P)$ physical time.
- Valiant's $O(\log\log N)$ algorithms takes $O(\log N \log\log N)$ physical time
  - It's slower than doing a simple reduce.
  - And it uses lots of communication – think of all those λ penalties!
  - But it's very clever. ☺
- Valiant understood this and pointed these issues in his paper.
  - But there has still be extensive research on PRAM algorithms.
  - It's an elegant model, what can I say?

Summary

- Simplified CTA reminds us that communication is expensive, but it doesn't explicitly charge for bandwidth.
- LogP accounts for bandwidth, but doesn't recognize that all bandwidth is not the same:
  - Communicating with an immediate neighbour is generally much cheaper than communicating with a distant machine.
  - Otherwise stated, the bisection bandwidth for real machines is generally much less than the per-machine bandwidth times the number of machines.
    - We can't have everyone talk at once at full bandwidth.
    - logP uses the bisection bandwidth – this is conservative, but it doesn't recognize the advantages of local communcation.
- Both are based on a 10-20 year old machine model
  - That's ok, the papers are 18-25 years old.
  - Doesn't account for the heterogeniety of today's parallel computers:
    - multi-core on chip, faster communication between processors on the same board than across boards, etc.
- We'll use CTA because it's simple.
  - But recognize the limitations of any of these models.
- Getting a model of parallel computation that's as all-purpose as the RAM is still a work-in-progress.

Preview

February 8: Parallel Sorting – The Zero-One Principle
Reading: https://en.wikipedia.org/wiki/Sorting_network

February 10: Bitonic Sorting (part 1)
Reading: https://en.wikipedia.org/wiki/Bitonic_sorter
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

February 13: Family Day – no class

February 15: Bitonic Sorting (part 2)
Homework: HW 3 earlybird (11:59pm), HW 4 goes out.

February 17: Map-Reduce
Homework: HW 3 due (11:59pm).
HW 4 goes out

February 27: TBD

March 1: Midterm

March 3: GPU Overview
Reading The GPU Computing Era

March 6: Intro. to CUDA
Reading Kirk & Hwu Ch. 2

March 8: CUDA Threads, Part 1
Reading Kirk & Hwu Ch. 3
Homework: HW 4 earlybird (11:59pm)

March 8: CUDA Threads, Part 2
Homework: HW4 due (11:59pm).

Review

- Compare and Contrast the main features of the PRAM, CTA, and LogP models?
- How does each model represent computation?
- How does each model represent communication?
- How might one determine parameter values for the CTA and LogP models? Describe at a high-level the kinds of experiments you could run to estimate the parameters. Hint: review the Jan. 9 lecture.
- What does the '**g**' stand for in "logP"?

For further reading

- [Valiant1975] Leslie G. Valiant, "Parallelism in Comparison Problems," *SIAM Journal of Computing*, vol. 4, no. 3, pp. 348–355, (Sept. 1975).
- [Fortune1979] Steven Fortune and James Wyllie, "Parallelism in Random Access Machines," *Proceeding of the 11th ACM Symposium on Theory of Computing* (STOC'79), pp. 114–118, May 1978.
- [Snyder1986] Lawrence Snyder, "Type architectures, shared memory, and the corollary of modest potential", *Annual review of computer science*, vol. 1, no. 1, pp. 289–317, 1986.
- [Culler1993] David Culler, Richard Karp, *et al.*, "LogP: towards a realistic model of parallel computation," *ACM SIGPLAN Notices*, vol. 28, no. 7, pp. 1–12, (July 1993).

EREW, CREW, and CRCW

- EREW: Exclusive-Read, Exclusive-Write
  - If two processors access the same location on the same step,
    - then the machine fails.
- CREW: Concurrent-Read, Exclusive-Write
  - Multiple machines can read the same location at the same time, and they all get the same value.
  - At most one machine can try to write a particular location on any given step.
  - If one processor writes to a memory location and another tries to read or write that location on the same step,
    - then the machine fails.
- CRCW: Concurrent-Read, Concurrent-Write
  If two or more machines try to write the same memory word at the same time, then if they are all writing the same value, that value will be written. Otherwise (depending on the model),
  - the machine fails, or
  - one of the writes "wins", or
  - an arbitrary value is written to that address.

Valiant Details

round	values remaining	group size	processors per group
1	N	$2*1+1=3$	$3 = 3$ choose 2
2	$\frac{N}{3}$	$2*3+1=7$	$3*7=21=7$ choose 2
3	$\frac{1}{7}\frac{N}{3} = \frac{N}{21}$	$2*21+1=43$	$21*43=903=43$ choose 2
4	$\frac{1}{43}\frac{N}{21} = \frac{N}{903}$	$2*903+1=1,807$	$903*1,807=1,631,721=1807$ choose 2
…	…	…	…
k	$\frac{N}{m_k}$	$2m_k+1$	$m_k(2m_k+1) = (2m_k+1)$ choose 2
$k+1$	$\frac{1}{2m_k+1}$ $= \frac{N}{m_k}\frac{N}{m_k(2m_k+1)}$ $= \frac{N}{m_{k+1}}$	$2m_{k+1}+1$	$m_{k+1}(2m_{k+1}+1) = (2m_{k+1}+1)$ choose 2

- m_k is the "sparsity" at round k:

$$\begin{aligned} m_1 &= 1 \\ m_{k+1} &= m_k(2m_k+1) \end{aligned}$$

- Now note that $m_{k+1} = m_k(2m_k+1) > 2m_k^2 > m_k^2$.
- Thus, $\log(m_{k+1}) > 2\log(m_k)$.
- For $k \geq 3$, $m_k > 2^{2^{k-1}}$.
- Therefore, if $N \geq 2$, $k > \log\log(N) + 1 \;\Rightarrow\; m_k > N$.

Let's solve the run-time recurrence

- For Valiant's algorithm. Let $m_0 = 3$ denote the sparsity at the first step.
- $m_{k+1} = 2m_k^2 + m_k$
  - $\log_2 m_{k+1} = \log_2(2m_k^2 + m_k)$
  - $2\log_2 m_k + 1 < \log_2 m_{k+1} < 2\log_2 m_k + 1 + \alpha/m_k$; where $\alpha = \log_2(e)/2$.
  - $2^k \log_2 m_0 + 2^k - 1 < \log_2 m_k < 2^k m_0 + (5/4)(2^k - 1)$; because $m_k \geq 3$, $\log_2(e)/6 = 0.240449\ldots < 1/4$.
  - $(1 + \log_2 3)2^k - 1 < \log_2 m_k < ((5/4) + \log_2 3)2^k - (5/4)$; because $m_0 = 3$.
- We want to find k such that $m_k \geq N$. It is sufficient if
  - $(1 + \log_2 3)2^k - 1 > \log_2 N$
  - $2^k > (\log_2 N + 1)/(1 + \log_2 3)$
  - $k > \log_2[(\log_2 N + 1)/(1 + \log_2 3)]$
  - For $N > 2$, $(\log_2 N + 1)/(1 + \log_2 3) < \log_2 N$.
- For $N > 2$, let $k = \log_2 \log_2 N$. We have shown that $m_k > N$.
  - Valiant's algorithm takes $O(\log\log N)$ rounds.
  - Each round takes constant time on a CRCW PRAM.
  - $\therefore$ Valiant's algorithm takes $O(\log\log N)$ time on a CRCW PRAM.

Sorting Networks

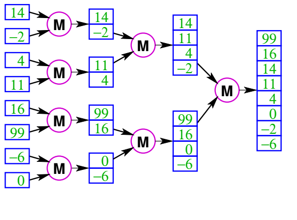
Mark Greenstreet

CpSc 418 – Feb. 8, 2017

- Parallelizing mergesort and/or quicksort
- Sorting Networks
- The 0-1 Principle
- Summary

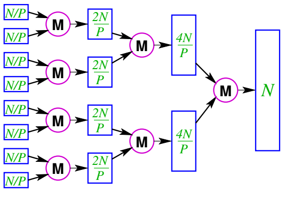
Parallelizing Mergesort

We could use reduce?



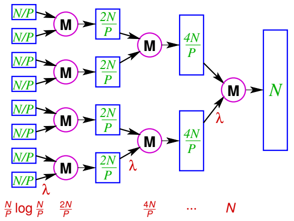
Parallelizing Mergesort

We could use reduce?



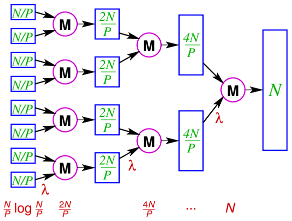
Parallelizing Mergesort

We could use reduce?



Parallelizing Mergesort

We could use reduce?



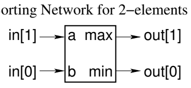
Total time: $\frac{N}{P}(\log N + 2(P-1) - \log P) + (\log P)\lambda$

Parallelizing Quicksort

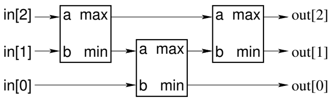
How would you write a parallel version of quicksort?

Sorting Networks

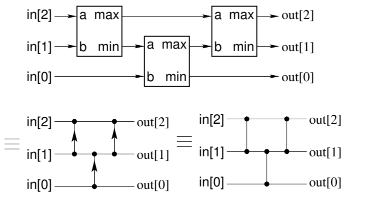
Sorting Network for 2–elements



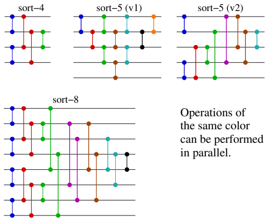
A Sorting Network for 3–elements



Sorting Networks – Drawing



Sorting Networks – Examples



Operations of
the same color
can be performed
in parallel.

See: http://pages.ripco.net/~jgamble/nw.html

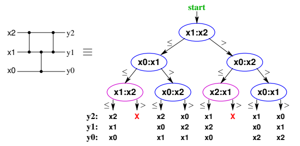
Sorting Networks: Definition

Structural version:

- A sorting network is an acyclic network consisting of compare-and-swap modules.
  - Each primary input is connected either to the input of exactly one compare-and-swap module or to exactly one primary output.
  - Each compare-and-swap input is connected either to a primary input or to the output of exactly one compare-and-swap module.
  - Each compare-and-swap output is connected either to a primary output or to the input of exactly one compare-and-swap module.
  - Each primary output is connected either to the output of exactly one compare-and-swap module or to exactly one primary input.
- More formally, a sorting network is either
  - the identity network (no compare and swap modules).
  - a sorting network, S composed with a compare-and-swap module such that two outputs of S are the inputs to the compare-and-swap, and the outputs of the compare-and-swap are outputs of the new sorting network (along with the other outputs of the original network).

Sorting Networks: Definition

Decision-tree version:



- Let v be an arbitrary vertex of a decision tree, and let x_i and x_j be the variables compared at vertex v.
- A decision tree is a sorting network iff for every such vertex, the left subtree is the same as the right subtree with x_i and x_j exchanged.

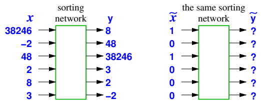
The 0-1 Principle

If a sorting network correctly sorts all inputs consisting only of 0s and 1s, then it correctly sorts inputs consisting of arbitrary (comparable) values.

- The 0-1 principle doesn't hold for arbitrary algorithms:
  - Consider the following linear-time "sort"
  - In linear time, count the number of zeros, nz, in the array.
  - Set the first nz elements of the array to zero.
  - Set the remaining elements to one.
  - This correctly sorts any array consisting only of 0s and 1s, but does not correctly sort other arrays.
- By restricting our attention to sorting networks, we can use the 0-1 principle.

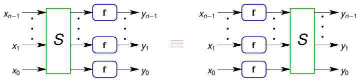
The 0-1 Principle: Proof Sketch

- We will show the contrapositive: if y is not sorted properly, then there exists an $\tilde{x}$ consisting of only 0s and 1s that is not sorted properly.



- Choose $i < j$ such that $y_i > y_j$.
- Let $\tilde{x}_k = 0$ if $x_k < x_i$ and $\tilde{x}_k = 1$ otherwise.
  - Clearly $\tilde{x}$ consists only of 0s and 1s.
  - We will show that the sorting network does not sort correctly with input $\tilde{x}$.

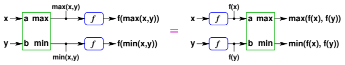
Monotonicity Lemma



Lemma: sorting networks commute with monotonic functions.

- Let S be a sorting network with n inputs an $N$ outputs.
  - I'll write $x_0, \ldots, x_{n-1}$ to denote the inputs of S.
  - I'll write $y_0, \ldots, y_{n-1}$ to denote the outputs of S.
- Let f be a monotonic function.
  - If $x \leq y$, then $f(x) \leq f(y)$.
- The monotonicity lemma says
  - applying S and then f produces the same result as
  - applying f and then S.
- Observation: `f(X) when X <` X_i `-> 0; f(_) -> 1.` is monotonic.

Compare-and-Swap Commutes with Monotonic Functions



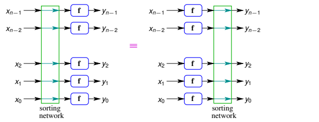
Compare-and-Swap commutes with monotonic functions.

- Case $x \leq y$:

$$\begin{array}{rcll} f(x) & \leq & f(y), & \text{because } f \text{ is monotonic.} \\ \max(f(x), f(y)) & = & f(y), & \text{because } f(x) \leq f(y) \\ \max(f(x), f(y)) & = & f(max(x,y)), & \text{because } x \leq y \end{array}$$

- Case $x \geq y$: equivalent to the $x \leq y$ case.
- □

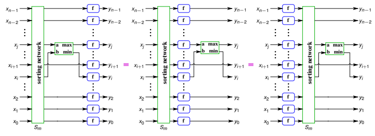
The monotonicity lemma – proof sketch



Induction on the structure of the sorting network, S.
Base case:

- The simplest sorting network, S_0 is the identity function.
- It has 0 compare-and-swap modules.
- Because S_0 is the identity function, $S_0(f(x)) = f(x) = f(S_0(x))$.

The monotonicity lemma – induction step



- Let S_m be a sorting network with n inputs and let $0 \leq i < j < n$.
- Let S_{m+1} be the sorting network obtained by composing a compare-and-swap module with outputs i and j of S_m.
- We can "move" the f operations from the outputs of the new compare-and-swap to the inputs (see slide 12).
- We can "move" the f operations from the outputs S_m to the inputs (induction hypothesis).
- Therefore, S_{m+1} commutes with f.

The 0-1 Principle

If a sorting network correctly sorts all inputs consisting only of 0s and 1s, then it correctly sorts inputs of any values.

I'll prove the contrapositive.

- If a sorting network does not correctly sort inputs of any values, then it does not correctly sort all inputs consisting only of 0s and 1s.
- Let S be a sorting network, let x be an input vector, and let $y = S(x)$, such that there exist i and j with $i < j$ such that $y_i > y_j$.
- Let

$$\begin{array}{rcll} f(x) & = & 0, & \text{if } x < y_i \\ & = & 1, & \text{if } x \geq y_i \\ \tilde{y} & = & S(f(x)) & \end{array}$$

- By the definition of f, $f(x)$ is an input consisting only of 0s and 1s.
- By the monotonicity lemma, $\tilde{y} = f(y)$. Thus,

$$\tilde{y}_i \;=\; f(y_i) \;=\; 1 \;>\; 0 \;=\; f(y_j) \;=\; \tilde{y}_j$$

- Therefore, S does not correctly sort an input consisting only of 0s and 1s.
- □

Summary

- Sequential sorting algorithms don't parallelize in an "obvious" way because they tend to have sequential bottlenecks.
  - Later, we'll see that we can combine ideas from sorting networks and sequential sorting algorithms to get practical, parallel sorting algorithms.
- Sorting networks are a restricted class of sorting algorithms
  - Based on compare-and-swap operations.
  - The parallelize well.
  - They don't have control-flow branches – this makes them attractive for architectures with large branch-penalties.
- The zero-one principle:
  - If a sorting-network sorts all inputs of 0s and 1s correctly, then it sorts all inputs correctly.
  - This allows many sorting networks to be proven correct by counting arguments.

Preview

February 10: Bitonic Sorting (part 1)
Reading: https://en.wikipedia.org/wiki/Bitonic_sorter
http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm

February 13: Family Day – no class

February 15: Bitonic Sorting (part 2)
Homework: HW 3 earlybird (11:59pm), HW 4 goes out.

February 17: Map-Reduce
Homework: HW 3 due (11:59pm).
HW 4 goes out

February 27: TBD

March 1: Midterm

March 3: GPU Overview
Reading The GPU Computing Era

March 6: Intro. to CUDA
Reading Kirk & Hwu Ch. 2

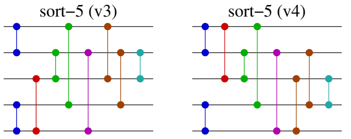
March 8: CUDA Threads, Part 1
Reading Kirk & Hwu Ch. 3
Homework: HW 4 earlybird (11:59pm)

March 8: CUDA Threads, Part 2
Homework: HW4 due (11:59pm).

Review 1

- Why don't traditional, sequential sorting algorithms parallelize well?
- Try to parallelize another sequential sorting algorithm such as heap sort? What issues do you encounter?
- Consider network sort-5(v2) from slide 6. Use the 0-1 principle to show that it sorts correctly?
  - What if the input is all 0s?
  - What if the input has exactly one 1?
  - What if the input has exactly two 1s?
  - What if the input has exactly three 1s? Note, it may be simpler to think of this the input having exactly two 0s.
  - What if the input has exactly four 1s? Five ones?

Review 2



Consider the two sorting networks shown above. One sorts correctly; the other does not.

- Identify the network that sorts correctly, and prove it using the 0-1 principle.
- Show that the other network does not sort correctly by giving an input consisting of 0s and 1s that is not sorted correctly.

Review 3

I claimed that `max` and `min` can be computed without branches. We could work out the hardware design for a compare-and-swap module. Instead, consider an algorithm that takes two "words" as arguments – each word is represented as a list of characters. The algorithm is supposed to output the two words, but in alphabetical order. For example:

```
% See: http://www.ugrad.cs.ubc.ca/~cs418/2016-2/lecture/02-08/cas.erl
compareAndSwap(L1, L2) when is_list(L1), is_list(L2) ->
   compareAndSwap(L1, L2, []).
compareAndSwap([], L2, X) ->
   {lists:reverse(X), lists:reverse(X, L2)};
compareAndSwap(L1, [], X) ->
   {lists:reverse(X), lists:reverse(X, L1)};
compareAndSwap([H1 | T1], [H2 | T2], X) when H1 == H2 ->
   compareAndSwap(T1, T2, [H1 | X]);
compareAndSwap(L1=[H1 | _], L2=[H2 | _], X) when H1 < H2 ->
   {lists:reverse(X, L1), lists:reverse(X, L2)};
compareAndSwap(L1, L2, X) ->
   {lists:reverse(X, L2), lists:reverse(X, L1)}.
```

Show that `compareAndSwap` can be implemented as a scan operation.

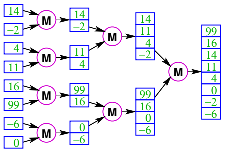
Bitonic Sort

Mark Greenstreet

CpSc 418 – Feb. 10, 2017

- Merging
- Shuffle and Unshuffle
- The Bitonic Sort Algorithm
- Summary
- I know that some of the links in the electronic version are broken. I know that it would be nice if I complete the final slides. I will post to piazza when this is done.

Parallelizing Mergesort



- We looked at this in the Feb. 8 lecture.
- The challenge is the merge step:
  - Can we make a parallel merge?

Merging and the 0-1 Principle

Easy cases

A	B	A	B	A	B
1	1	1	1	1	1
1	1	1	1	1	1
1	0	0	1	1	1
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

The main idea:

- Use divide-and-conquer.
  - Given two arrays, A and B, divide them into smaller arrays that we can merge, and then easily combine the results.
  - What criterion should we use for dividing the arrays?
- Observation:
  - It's easy to merge two arrays of the same size, if they both have the same number of **1s**.
  - If they have **nearly** the same number of **1s**, that's easy as well.

Dividing the problem (part 1)

- For simplicity, assume each array has an even number of elements.
  - As we go on, we'll assume that each array has an power-of-two number of elements.
  - That's the easiest way to explain bitonic sort.
  - Note: the algorithm works for arbitrary array sizes.
    - See the lecture slides from 2013.
- Divide each array in the middle?
  - If A has $N$ elements and N_1 are ones,
  - How many ones are in $A[0, \ldots, (N/2) - 1]$?
  - How many ones are in $A[N/2, \ldots, N - 1]$?
- Taking every other element?
  - How many ones are in the $A[0, 2, \ldots, N - 2]$?
  - How many ones are in the $A[1, 3, \ldots, N - 1]$?
- Other schemes?

Dividing the problem (part 2)

- Let A and B be arrays that are sorted into ascending order.
  - Let A_0 be the odd-indexed element of A and A_1 be the odd-indexed.
  - Likewise for B_0 and B_1.
- Key observations:

$$\begin{array}{lclcl} \text{HowManyOnes}(A_0) & \leq & \text{HowManyOnes}(A_1) & \leq & \text{HowManyOnes}(A_0) + 1 \\ \text{HowManyOnes}(B_0) & \leq & \text{HowManyOnes}(B_1) & \leq & \text{HowManyOnes}(B_0) + 1 \end{array}$$

- With a bit of algebra, we get

$$\left| \text{HowManyOnes}(A_0 \mathbin{++} B_1) - \text{HowManyOnes}(A_1 \mathbin{++} B_0) \right| \;\leq\; 1$$

- In English that says that
  - If we merge A_0 with B_1 to get C_0,
  - and we merge A_1 with B_0 to get C_1,
  - then C_0 and C_1 differ by at most one in the number of ones that they have.
    - This is an "easy" case from slide 3.

Merging

- Given $N$ that is a power of 2, and arrays A and B that each have $N$ elements and are sorted into ascending order, we can merge them with a sorting network.
- If $N = 1$, then just do `CompareAndSwap(`A`, `B`)`.
- Otherwise, let A_0 be the odd-indexed element of A and A_1 be the odd-indexed, and likewise for B_0 and B_1.
- Merge A_0 and B_1 into a single ascending sequence, C_0.
- Merge A_1 and B_0 into a single ascending sequence, C_1.
  - Note that the number of ones in C_0 and C_1 differ by at most one.
- Merge C_0 and C_1 into a single ascending sequence.
  - This is an "easy" case from slide 3.
  - We can perform this merge using $^{N/2}\!/$compare-and-swap modules.
- Complexity:
  - Depth: $O(\log N)$ – logarithmic parallel time.
  - Number of compare-and-swap modules $O(N \log N)$.
- **Pause:** If you understand this, you've got all of the key ideas of bitonic sorting.
  - The bitonic approach just improves on this simple algorithm.

Bitonic Sequences

- A sequence is **bitonic** if it consists of a monotonically increasing sequence followed by a monotonically decreasing sequence.
  - Either of those sub-sequences can be empty.
  - We'll also consider a monotonically decreasing followed by monotonically increasing sequence to be bitonic.
- Properties of bitonic sequence
  - Any subsequence of a bitonic sequence is bitonic.
  - Let A be a bitonic sequence consisting of **0s** and **1s**. Let A_0 and A_1 be the even- and odd-indexed subsequences of A.
  - The number of **1s** in A_0 and A_1 differ by at most 1.
    - We'll examine the number of **0s** on slide 10.

Bitonic Merge – big picture

- Bitonic merge produces a monotonic sequence from an bitonic input.
- Given two sorted sequences, A and B, note that

$$X \;=\; A \mathbin{++} \text{reverse}(B)$$

is bitonic.

  - We don't require the lengths of A or B to be powers of two.
  - If fact, we don't even require that A and B have the same length.
- Divide X into X_0 and X_1, the even-indexed and odd-indexed subsequences.
  - X_0 and X_1 are both bitonic.
  - The number of **1s** in X_0 and X_1 differ by at most 1.
- Use bitonic merge (recursion) to sort X_0 and X_1 into ascending order to get Y_0 and Y_1.
  - $\text{HowManyOnes}(Y_0) = \text{HowManyOnes}(X_0)$, and $\text{HowManyOnes}(Y_1) = \text{HowManyOnes}(X_1)$.
  - Therefore, the number of **1s** in Y_0 and Y_1 differ by at most 1.
  - This is an "easy" case from slide 3.

Counting the 0s and 1s (even total length)

X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1
0	0	0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	0	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1

- First, we'll look at the case when $\text{length}(A \mathbin{++} B)$ is even.
- Given two sorted sequences, A and B, let

$$\begin{array}{rcl} X_0 & = & \text{EvenIndexed}(A \mathbin{++} \text{reverse}(B)) \\ X_1 & = & \text{OddIndexed}(A \mathbin{++} \text{reverse}(B)) \end{array}$$

  - This means that $X[i] = X_{i \bmod 2}[i \text{ div } 2]$.
  - In English, the elements of X go left-to-right and then bottom-to-top in X_0 and X_1.
- The number of **1s** in X_0 and the number of ones in X_1 differ by at most 1.
- Likewise for the number of **0s**.

Counting the 0s and 1s (odd total length)

X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1	X_0	X_1
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1
0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	1	1	0	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0		0		0		0		0		0		0		1	

- Let $N = \text{length}(A \mathbin{++} B)$, where $N$ is odd.
- The number of **1s** in X_0 and the number of ones in X_1 differ by at most 1.
- The number of **0s** in $X_0[1, \ldots, \lfloor N/2 \rfloor]$ and the number of zeros in X_1 differ by at most 1.
- Either $X_0[0]$ or $X_0[\lfloor N/2 \rfloor]$ is the **least** element of $A \mathbin{++} B$.

After applying bitonic merge to X_0 and Y_0

Y_0	Y_1	Y_0	Y_1	Y_0	Y_1	Y_0	Y_1	Y_0	Y_1	Y_0	Y_1	Y_0	Y_1	Y_0	Y_1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1	1	1	0	1	1	1	1
0	0	0	1	0	0	0	0	0	1	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
				0		0		0		0		0		1	

- Let $N = \text{length}(A \mathbin{++} B)$.
-
- If $N$ is even,
  - Any out of order elements are in the same row, i.e. $X_0[i] > X_1[i]$ for some $0 \leq i < N/2$.
- If $N$ is odd
  - Any out of order elements are of the form $X_0[i+1] > X_1[i]$ for some $0 \leq i < N/2$.
  - $X_0[0]$ is the least element of X_0 and X_1.

The complexity of bitonic merge

- We'll count the compare-and-swap operations
  - Is it OK to ignore reversing one array, concatenating the arrays, separating the even- and odd-indexed elements, and recombining them later?
  - Yes. The number of these operations is proportional to the number of compare-and-swaps
  - Yes. Even better, in the next lecture, we'll show how to eliminate most of these data-shuffling operations.
- A bitonic merge of $N$ elements requires:
  - two bitonic merges of $N/2$ items (if $N > 2$)
  - $\lfloor N/2 \rfloor$ compare-and-swap operations.
- The total number of compare and swap operations is $O(N \log N)$.

Bitonic-Sort, and it's complexity

Shuffle and unshuffle

- Shuffle is like what you can do with a deck of cards:
  - Divide the deck in half
  - Select cards alternately from the two halves.
  - Shuffle is a circular-right-shift of the index bits.
    - Assuming the number of cards in the deck is a power of two.
- Unshuffle is the inverse of shuffle.
  - Unshuffling a deck of cards is dealing to two players.
  - Unshuffle is a circular-left-shift of the index bits.

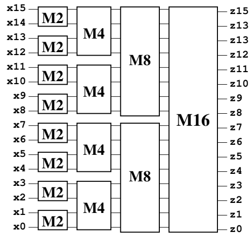
Bitonic Sort

Mark Greenstreet

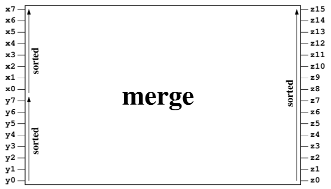
CpSc 418 – Feb 15, 2017

- The Bitonic Sort Algorithm
- Shuffle, Unshuffle, and Bit-operations
- Bitonic Sort In Practice
- Related Algorithms

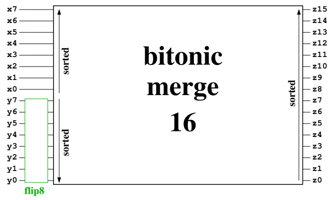
Parallelizing Mergesort



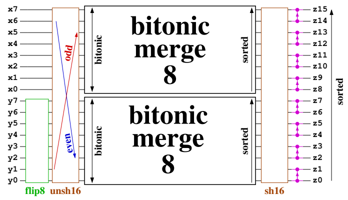
Bitonic Merge



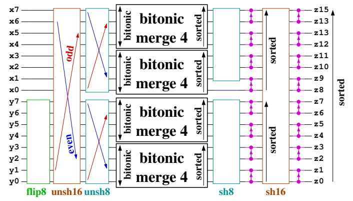
Bitonic Merge



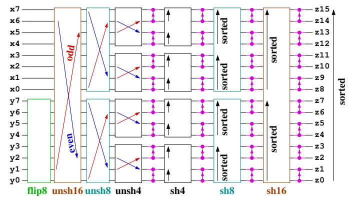
Bitonic Merge



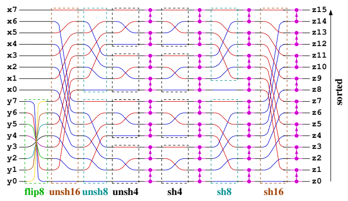
Bitonic Merge



Bitonic Merge



Bitonic Merge



Shuffle

- Given two sequences, X of length N where N is even, the **shuffle** of X is $Y = \text{shuffle}(X)$ where

$$\begin{aligned} Y_i &= X_{i/2}, & \text{if } i \text{ is even} \\ &= X_{(i+N-1)/2}, & \text{if } i \text{ is odd} \end{aligned}$$

  - $\text{shuffle}([0,1,2,3,4,5,6,7]) \rightarrow [0,4,1,5,2,6,3,7]$.
- shuffle is like shuffling a deck of cards.
  - Split the deck in half.
  - Interleave the cards from the two halves.
- If N is a power of 2, then shuffle rotates the least-significant bit of the index to the most significant bit:

```
shuffle([000, 001, 010, 011, 100, 101, 110, 111]) ->
  [000, 100, 001, 101, 010, 110, 011, 111])
```

- If N is odd,

$$\begin{aligned} Z_i &= X_{i/2}, & \text{if } i \text{ is even} \\ &= X_{(i+N)/2}, & \text{if } i \text{ is odd} \end{aligned}$$

  - $\text{shuffle}([0,1,2,3,4]) \rightarrow [0,3,1,4,2]$

Unshuffle

- The inverse of shuffle.
- Let $N = \text{length}(Y)$ and $X = \text{unshuffle}(Y)$, then

$$\begin{aligned} X_i &= Y_{2i}, & \text{if } i < N/2 \\ &= X_{2i-N+1}, & \text{if } N/2 \leq i \end{aligned}$$

- It's like dealing a deck of cards into two piles, and then stacking one pile on top of the other.
- If N is a power of 2, then unshuffle rotates the most significant bit of the index to the least significant bit:
- If N is odd,

$$\begin{aligned} X_i &= Y_{2i}, & \text{if } i < (N+1)/2 \\ &= X_{2i-N}, & \text{if } (N+1)/2 \leq i \end{aligned}$$

Bit operations: rotr and rotl

- rotr(I, W) % Rotate the lower W bits of I one place to the right:

```
rotr(I, 0) -> I;
rotr(I, W) when is_integer(W), W > 0 ->
   Mask = (1 bsl W) - 1,        % ones in the W least-significant bits
   Ilo = I band Mask,           % the W least-significant bits of I
   Ihi = I band (bnot Mask),    % the rest of I
   Ilsb = I band 1,             % the least-significant-bit of I
   % Ilor is Ilo rotated one place to the right
   Ilor = (Ilsb bsl (W-1)) bor (Ilo bsr 1),
   Ihi bor Ilor.
```

  - rotr(6,3) -> 5;
  - rotr(6,4) -> 12;
- rotr(I, W) rotates the lower W bits of I 1 place to the left.
- Note: rotr(I,1) -> I, and rotl(I,1) -> I.

Shuffle, Unshuffle, and Bit-Operations

- If K is a power of 2, x[0..(K-1)] is the input of a shuffle_K module, and y[0..(K-1)] is the output, then
  - the shuffle_K operation moves x[i] to y[rotl(i, log2(k))].
  - equivalently: y[j] = x[rotr(j, log2(k))].
- If K is a power of 2, x[0..(K-1)] is the input of a unshuffle_K module, and y[0..(K-1)] is the output, then
  - the unshuffle_K operation moves x[i] to y[rotr(i, log2(k))].
  - equivalently: y[j] = x[rotl(j, log2(k))].

The Initial Unshuffles

- Bitonic merge for K elements starts with an unshuffle_K, followed by a unshuffle_$\frac{K}{2}$, followed by a unshuffle_$\frac{K}{4}$, …, followed by a unshuffle_1.
- If we let x[0..(K-1)] be the input to this network (I'm assuming we've already done the flip for inputs x[0..((K/2)-1)]), and y[0..(K-1)] be the output then:
  - y[j] = x[rotl(rotl(...rotl(rotl(j, 1), 2), ..., log2(K)-1), log2(K))]
  - and we note that:
    rotl(rotl(...rotl(rotl(j, 1), 2), ..., log2(K)-1), log2(K)) = bitrev(j, log2(K))
    where bitrev(j, W) is the bit-reverse of the lower W bits of j.
- More specifically, for the 16-way bitonic merge, $K = 16$ and $\log_2(K) = 4$.
  - If we write array indices as four bits, b_3, b_2, b_1, b_0,
  - Then y[b_3, b_2, b_1, b_0] = x[b_0, b_1, b_2, b_3].

The first compare-and-swap

The first compare-and-swap operates on y[$b_3, b_2, b_1, 0$] and y[$b_3, b_2, b_1, 1$], for all 8 choices of b_3, b_2, and b_1.

- This corresponds to a compare-and-swap of x[$b_0, b_1, b_2, 0$] with x[$b_0, b_1, b_2, 1$].
- I'll call the result of the compare-and-swap z where
  - z[$b_3, b_2, b_1, 0$] = min(y[$b_3, b_2, b_1, 0$], y[$b_3, b_2, b_1, 1$]);
  - z[$b_3, b_2, b_1, 1$] = max(y[$b_3, b_2, b_1, 0$], y[$b_3, b_2, b_1, 1$]);
- And I'll write $\widetilde{z}$ for z with "x indexing":
  - $\widetilde{z}$[b_3, b_2, b_1, b_0] = z[b_0, b_1, b_2, b_3];
  - $\widetilde{z}$[$0, b_2, b_1, b_0$] = min(x[$0, b_2, b_1, b_0$], x[$1, b_2, b_1, b_0$]);
  - $\widetilde{z}$[$1, b_2, b_1, b_0$] = max(x[$0, b_2, b_1, b_0$], x[$1, b_2, b_1, b_0$])
  - These are comparisons with a "stride" of 8 (for x).

The first shuffle

- The first shuffle takes z as an input and I'll call the output w.
- The first shuffle is a shuffle_4; so
  - w[i] = z[rotl(i,2)].
  - Equivalently, w[b_3, b_2, b_1, b_0] = z[b_3, b_2, b_0, b_1].
- Let

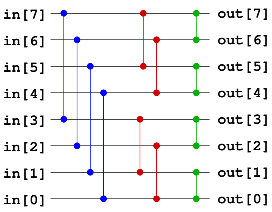
$$\begin{aligned} \widetilde{w}[b_3, b_2, b_1, b_0] &= w[b_0, b_1, b_3, b_2] \\ &= z[b_0, b_1, b_2, b_3] \\ &= \widetilde{z}[b_3, b_2, b_1, b_0] \end{aligned}$$

- The second stage of compare-and-swap modules operates on
  - w[$b_3, b_2, b_1, 0$] and w[$b_3, b_2, b_1, 1$]
  - Equivalently, $\widetilde{w}$[$b_1, 0, b_2, b_3$] and $\widetilde{w}$[$b_1, 1, b_2, b_3$].
  - These are comparisons with a stride of 4 for $\widetilde{z}$ and $\widetilde{w}$.

The rest of the merge

- In the same way, the third stage of compare-and-swap modules operates has a stride of 2 for x indices,
- And the final stage has a stride of 1.
- More generally, to merge two sequences of length 2^L:
  - Flip the lower sequence
  -
  - Or, just sort it in reverse in the first place.
  - Perform compare-and-swap operations with stride L.
  - Perform compare-and-swap operations with stride $L/2$ – note that these operate on pairs of elements whose indices differ in the $L/2$ bit, and all of their other index bits are the same.
  - Perform compare-and-swap operations with stride $L/4$, …
  - Perform compare-and-swap operations with stride 1 – this compares the element at 2*i with the element at 2*i+1 for 0 ≤ i < 2^L.
- Done!

The "Textbook" Diagram

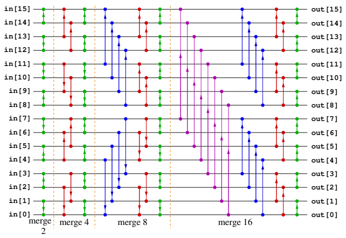


Flipping Out

What should we do about the flips?

- Push them back (right-to-left) through the network
  - Keep track of how many flips we've accumulated.
  - Sort up for an even number of flips.
  - Sort down for an odd number of flips.
- Flip the wiring in the bottom half of each unshuffle.
- In practice:
  - Do the one that's easier for your implementation.

Bitonic Sort



Bitonic Sort in practice

- Sorting networks can be used to design practical sorting algorithms.
- To sort N values with P processors:
  - Divide input into $2P$ segments of length $\frac{N}{2P}$.
  - Each processor sorts its pair of segments into one long segment.
    - The sorted segments are the inputs to the sorting network.
  - Now, follow the actions of the sorting network:
    - Processor I handles rows $2I$ and $2I + 1$ of the sorting network.
    - Each compare-and-swap is replaced with "merge two sorted sequences and split into top half and bottom half."
    - When the sorting network has a compare-and-swap between rows $2I$ and $2I + 1$, each processor handles it locally.
    - When the sorting network has a compare-and-swap between rows $2I$ and $2I + K$ for $K > 1$, then processor I sends the upper half of its data to processor $I + (K/2)$, and processor $I + (K/2)$ sends the lower half of its data to processor I. Both perform merges.
    - Note, if the compare-and-swap was flipped, then flip "upper-half" and "lower half".

Practical performance

- Complexity
  - Total number of comparisons: $O(N(\log N \log^2 P))$.
  - Time: $O\left(\frac{N}{P}(\log N + \log^2 P)\right)$, assuming each processor sorts N/P elements in $O((N/P)\log(N/P))$ time and merges two sequences of N/P elements in $O(N/P)$ time.
- Remarks:
  - The idea of replacing compare-and-swap modules with processors that can perform merge using an algorithm optimized for the processor, is an extremely powerful and general one. It is used in the design of many practical parallel sorting algorithms.
  - Sorting networks are cool because they avoid branches:
    - Ideal for SIMD machines that can't really branch.
    - Need to experiment some to see the trade-offs of branch-divergence vs. higher asymptotic complexity on a GPU.

Related Algorithms

- Counting Networks
  - How to match servers to requests.
- FFT
  - The Platonic Ideal of a Divide-and-Conquer Algorithm
  - Used for speech processing, signal processing, and lots of scientific computing tasks.