

# Parallel Computation

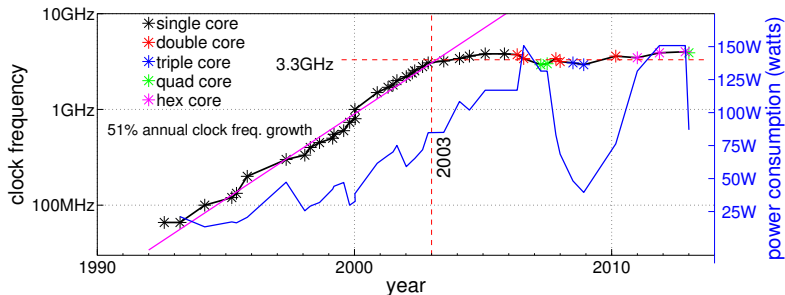
Mark Greenstreet

CpSc 418 – Jan. 4, 2016

## Outline:

- [Why Parallel Computation Matters](#)
- [Course Overview](#)
- [Our First Parallel Program](#)
- [The next few weeks](#)
- [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 12 years.
- Need other ways to increase performance and enable new applications.

# 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

<u>Homework:</u>	35%	roughly one HW every two weeks
<u>Midterm:</u>	25%	February 10, in class
<u>Final:</u>	40%	
<u>Mini-Assignments:</u>	see description on <a href="#">slide 19</a>	
<u>Bug Bounties:</u>	see description on <a href="#">slide 20</a>	
  - ▶ 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 (for the introductory problems we examine).
  - ▶ 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.
- Java threads: use your CPU's many cores
  - ▶ Perhaps the most treacherous of these three approaches.
  - ▶ Lets us explore some widely used algorithms.

# Syllabus

- January: Erlang
  - Jan. 4-8:** Course overview, intro. to Erlang programming.
  - Jan. 11-15:** Parallel programming in Erlang, reduce and scan.
  - Jan. 18-22:** Parallel architectures
  - Jan. 25-29:** Performance analysis
- February: Erlang, Midterm, & CUDA
  - Feb. 1-5:** Matrix Multiplication
  - Feb. 8:** Family Day, no class.
  - Feb. 10:** **Midterm** – in class
  - Feb. 12:** SIMD architectures and GPUs.
  - Feb. 15-19:** Midterm break.
  - Feb. 22-26:** Introduction to CUDA
- March: CUDA and Java Threads
- April: More Java

# Syllabus

- January: Erlang
- February: Erlang, Midterm, & CUDA
- March: CUDA and Java Threads
  - Feb. 29 – Mar. 4:** Matrix multiply with CUDA
  - Mar. 7-11:** Sorting
  - Mar. 14-18:** Dynamic Programming
  - Mar. 21-23:** Introduction to Java Threads
  - Mar. 30 – Apr. 1:** Mutual exclusion
- April: More Java
  - Apr. 4-Apr. 8:** Fun with threads
- Note: I'll make adjustments to this schedule as we go.

# Administrative Stuff – Who

- The instructor

- ▶ **Mark Greenstreet**, [mrg@cs.ubc.ca](mailto:mrg@cs.ubc.ca)
- ▶ ICICS/CS 323, (604) 822-3065
- ▶ Office hours: Mondays, 1pm – 2:30pm, ICICS/CS 323
  - Office hours will change if the proposed time doesn't work for many students in the class,
  - or if I end up with another meeting scheduled at that time.
  - You can always send me e-mail to make an appointment.

- The TAs

**Golnaz Jahesh**, [golnaz.jahesh@gmail.com](mailto:golnaz.jahesh@gmail.com)

Office Hours: Fridays, 11am – 12 noon, Demco 150

**Minchen Li**, [minchenl@cs.ubc.ca](mailto:minchenl@cs.ubc.ca)

Office Hours: Thursdays, 2:15pm – 3:15pm, Demco 150

- 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* (2<sup>nd</sup> ed.), D.B. Kirk and W-M.W. Hwu.
  - ▶ Please get a copy by mid-February – I'll assign readings starting Feb. 22. It's available at [Amazon.ca](http://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.
  - ▶ *The Art of Multiprocessor Programming* (chap. TBD), M. Herlihy & N. Shavit – for something fun with Java threads.
  - ▶ 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

<u>Homework:</u>	35%	roughly one assignment every two weeks
<u>Midterm:</u>	25%	February 10, in class
<u>Final:</u>	40%	
<u>Mini-Assignments:</u>	see description on <a href="#">slide 19</a>	
<u>Bug Bounties:</u>	see description on <a href="#">slide 20</a>	

# 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 books, other book, 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 your assignment by the early-bird date to get a 5% bonus.
- ▶ **No late homework accepted.**

# Exams

- Midterm, in class, on February 10.
- 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.
- ▶ There will be 5-10 such mini-assignments.
- ▶ The first one will be available on Jan. 4 and due Jan. 8.

- I will post it at:

- <http://www.ugrad.cs.ubc.ca/~cs418/2015-2/mini/1/mini1.pdf>

- Note 1: this one is “optional” – I’ll only include it in your average for mini-assignments if you submit a solution.
  - Note 2: If you’re on the class registration wait-list, then do it. I expect only a few openings, and priority will be given to those who do well on the first two mini-assignments.

# 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.

# Grades: the big picture

$$RawGrade = 0.35 * HW + 0.25 * MidTerm + 0.40 * Final$$

$$MiniBonus = 0.20 * (1 - \min(RawGrade, 1)) * Mini$$

$$BB = 0.35 * BB_{HW} + 0.25 * BB_{MT} + 0.40 * BB_{FX}$$

$$CourseGrade = \min(RawGrade + MiniBonus + BB, 1) \times 100\%$$

## Mini-assignments:

- If your raw grade is 90%, you can get at most 2% from the minis. You can afford to skip them if you're doing well and want to spend your on other courses.
- If your raw grade is 70%, you can get at most 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.

# Grades: the big picture

$$RawGrade = 0.35 * HW + 0.25 * MidTerm + 0.40 * Final$$

$$MiniBonus = 0.20 * (1 - \min(RawGrade, 1)) * Mini$$

$$BB = 0.35 * BB_{HW} + 0.25 * BB_{MT} + 0.40 * BB_{FX}$$

$$CourseGrade = \min(RawGrade + MiniBonus + BB, 1) \times 100\%$$

- I'll probably toss in some extra credit marks into the regular HW – these tend to be “unreasonable” problems. They are intended to be fun challenges for those who are otherwise blowing the course away and would enjoy learning more.
- Bug-bounties are for everyone. They reward you for looking at the HW when it first comes out, and not waiting until the day before it is due. 😊

# 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.
- ▶ Can apply these patterns to new 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, CUDA, and Java threads.
- ▶ 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/~cs418/2015-2/lecture/01-04/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` being 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](#) includes a function `rlist(N, M)` 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)).
```

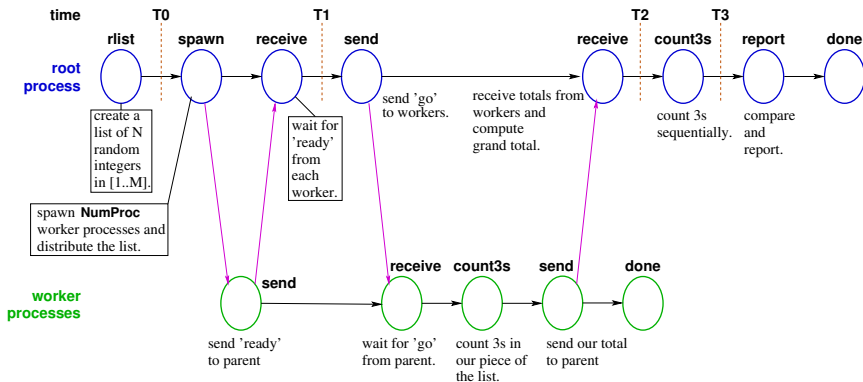
```
99961
```

```
5> q().
```

```
ok
```

```
6> bash-3.2$
```

# A Parallel Version



# The Root Process

```
main(N, M, NumProc) ->
    MyPid = self(),           % Get the root process's id
    R = rlist(N, M),          % Make a list of random numbers
    T0 = erlang:monotonic_time(), % Record the time
    WPids = workers_spawn(R, NumProc, MyPid), % Spawn workers
    T1 = erlang:monotonic_time(), % Record the time
    workers_go(WPids, MyPid), % Tell workers to start counting their 3s
    N3s_par = workers_sum(WPids), % Collect the results and compute total
    T2 = erlang:monotonic_time(), % Record the time
    N3s_seq = count3s(R),      % Now count the 3s sequentially
    T3 = erlang:monotonic_time(), % Record the time
    ...                        % print results and return
```

- See `main(N, M, NumProcs)` in [count3s.erl](#)

# Spawning the Workers

```
workers_spawn(_List, 0, _MyPid) -> [];           % 0 workers -> empty list
workers_spawn(List, NumProc, MyPid) ->           % The recursive case
    Len = length(List),
    {Prefix, Rest} = lists:split(Len div NumProc, List),
    % spawn WPid to work on Prefix:
    WPid = spawn(count3s, worker, [MyPid, Prefix]),
    receive {WPid, ready} -> ok end,              % wait for WPid to say 'ready'
    % spawn remaining workers to work on Rest:
    [WPid | workers_spawn(Rest, NumProc-1, MyPid)].
```

- See `workers_spawn` in [count3s.erl](#)
- [worker](#) is the name for the function to call in the spawned process. It's described on [slide 36](#).

## workers\_go and workers\_sum

```
workers_go([], _MyPid) -> ok; % All workers notified. We're done.  
workers_go([Wpid | Tail], MyPid) ->  
    Wpid ! {MyPid, go}, % Tell our worker to start counting  
    workers_go(Tail, MyPid). % Recurse to handle the remaining workers  
  
workers_sum([]) -> 0; % base case  
workers_sum([Wpid | Tail]) -> % recursive case  
    % Receive tally from our worker and add to total from the remaining workers:  
    (receive {Wpid, N3s} -> N3s end) + workers_sum(Tail).
```

- See `workers_go` and `workers_sum` in [count3s.erl](#)

# The worker processes

% PPid is the pid for the root process.

% MyStuff is our segment of the list in which we're counting 3s.

```
worker(PPid, MyStuff) ->
```

```
    MyPid = self(),
```

```
    PPid ! {MyPid, ready},
```

```
    receive {PPid, go} -> ok end,
```

```
    My3s = count3s(MyStuff),
```

```
    PPid ! {MyPid, My3s},
```

```
    ok.
```

% tell the root we're running

% wait for 'go'

% count our threes

% send our count to our parent

% that's all!

- See worker in [count3s.erl](#)

# Let's try it

On my laptop with a two-core, i5 CPU:

```
1> count3s:main(10000000, 10, 8).
```

```
Random list of 10000000 values from 1..10 has 1001092 3s
```

```
Sequential time: 0.1052 seconds
```

```
Parallel time (without spawn): 0.0418 sec., speed-up=2.52
```

```
Parallel time (including spawn): 0.8791 sec., speed-up=0.12
```

- Not bad for a two-core CPU!

- ▶ The “super-linear” speed-up is due to multi-threading.

- ▶ We'll examine multi-threading in a few weeks.

- The version that includes the spawn time has a slow-down:

- ▶ This is because erlang **copies** the list when spawning the child.

- ▶ It's tempting to create a bunch of workers, send them data, and wait for them to respond.

- ▶ It's much better to keep the workers available, and keep the data on the workers.

- ▶ **Communication cost is one of the biggest overheads for parallel programs.**

- [thetis.ugrad.cs.ubc.ca](http://thetis.ugrad.cs.ubc.ca) gets a speed up of about 12 when running 24 threads (ignoring spawn-time).

## If you're on the class waiting list

- The course is already slightly over-subscribed, so there will only be a few openings available. ☹
- The department will give **strong** preference to those who do the mini-assignments:
  - ▶ Mini-assignment 1 will be released on the morning of Jan. 4 at <http://www.ugrad.cs.ubc.ca/~cs418/2015-2/mini/1/mini1.pdf>
  - ▶ Mini-assignment 1 is due on Friday, Jan. 8 at 10:00am.
  - ▶ Mini-assignment 2 will go out on Jan. 6 at <http://www.ugrad.cs.ubc.ca/~cs418/2015-2/mini/2/mini2.pdf>
  - ▶ Mini-assignment 2 will be due on Jan. 11 at 10:00am.
- I am also teaching CpSc 521, the graduate version of this course.
  - ▶ If you have a GPA  $> 80$  and are interested more research oriented version of this course, talk to me after class or send me e-mail
  - ▶ You can do this even if you aren't on the waiting list – someone who is on the list will thank you. ☺.



# Preview of the next two weeks

## January 6: Introduction to Erlang Programming

Reading: [Learn You Some Erlang](#), the first four sections – [Introduction](#) through [Modules](#). Feel free to skip the stuff on [bit syntax](#) and [binary comprehensions](#).

Mini-Assignment: **Mini-Assignment 2 goes out**

---

## January 8: More FUN with Erlang FUNCTIONS

Reading: [Learn You Some Erlang](#), the next four sections – [Syntax in Functions](#) through [Higher Order Functions](#)

Mini-Assignment: **Mini-Assignment 1 due 10:00pm**

Homework: **Homework 1 goes out** – simple programming with Erlang

---

## January 11: Processes and Messages

Reading: [Learn You Some Erlang](#), [The Hitchhiker's Guide . . .](#) through [More on Multiprocessing](#)

Mini-Assignment: **Mini-Assignment 2 due 10:00am**

---

## January 13: Reduce and Scan (simple)

Reading: Lin & Snyder, chapter 5, pp. 112–125

---

## January 15: Reduce and Scan (generalize)

Homework: **Homework 1 deadline for early-bird bonus (11:59pm)**  
**Homework 2 goes out** – parallel programming with Erlang

---

## January 18: Architecture Review

Reading: Pacheco, Chapter 2, Sections 2.1 and 2.1.

Homework: **Homework 1 due 11:59pm**

---

# 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 interpreter 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](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](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.
- Or, download it for your own computer.
- See <http://www.erlang.org/download.html>
- Or  
<https://www.erlang-solutions.com/resources/download.html>,  
includes a `.dmg` for OSX. ☺

# Starting Erlang

- Start the Erlang interpreter.

```
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](http://en.wikipedia.org/wiki/List_of_CPU_power_dissipation),  
April 2011.

accessed 26 July 2011.



# Table Of Contents (1/2)

- Motivation
- Course Overview
  - ▶ Topics
    - Computer Architecture
    - Performance Analysis
    - Algorithms
    - Languages, Paradigms, and Frameworks
  - ▶ Syllabus
  - ▶ Course Administration – who's who
  - ▶ The Textbook(s)
  - ▶ Grades
    - Homework
    - Midterm and Final Exams
    - Mini-Assignments
    - Bug Bounties
  - ▶ Plagiarism Policy
  - ▶ Learning Objectives

# Table Of Contents (2/2)

- Our First Parallel Program
  - ▶ Introduction to Erlang
  - ▶ The Count 3s Example
    - Figure illustrating the parallel count3s
    - The Root Process
    - Spawning Worker Processes
    - Telling the Workers to 'go'
    - Computing the final sum
    - Code for the worker processes
    - Running the code
- The course waiting list
- Preview of the next two weeks
- Review of this lecture
- Supplementary Material
  - ▶ Erlang Resources
  - ▶ Bibliography
  - ▶ Table of Contents