Parallel Functional Programming
Lecture 1

John Hughes, Mary Sheeran
Moore’s Law (1965)

”The number of transistors per chip increases by a factor of two every year”

...two years (1975)
Number of transistors
What shall we do with them all?

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

A computer consists of three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the von Neumann bottleneck.
The task of a program is to change the contents of the store in some major way; when one considers that this task must be accomplished entirely by pumping single words back and forth through the von Neumann bottleneck, the reason for its name is clear.

Since the state cannot change during the computation... there are no side effects. Thus independent applications can be evaluated in parallel.
programming is HARD!!
Small transistors switch faster. Pipelined architectures permit faster clocks.
Performance per clock

Cache memory

Superscalar processors

Out-of order execution

Speculative execution (branch prediction)

Value speculation
Higher clock frequency ➔ higher power consumption
“By mid-decade, that Pentium PC may need the power of a nuclear reactor. By the end of the decade, you might as well be feeling a rocket nozzle than touching a chip. And soon after 2010, PC chips could feel like the bubbly hot surface of the sun itself.”

—Patrick Gelsinger, Intel’s CTO, 2004
Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

- Dual-Core Itanium 2
- Pentium 4
- Pentium
- 386

More cores
Stable clock frequency
Stable perf. per clock

Graph showing trends from 1970 to 2010 with data on Transistors (M), Clock Speed (MHz), Power (W), and Perf/Clock (ILP).
The Future is Parallel

- Intel Xeon
  - 18 cores
  - 36 threads

- AMD Opteron
  - 16 cores

- EZChip (Tilera)
  - TILE-Mx
  - 100 cores

- Azul Systems Vega 3
  - Cores per chip: 54
  - Cores per system: 864
See also the recent Chalmers Tech Talk from the CEO of Adapteva

http://complab.github.io/abstracts.html#olofsson
See also the recent Chalmers Tech Talk from the CEO of Adapteva

http://complab.github.io/abstracts.html#olofsson

And don’t forget Graphics Processors
$10^{10}$ transistors for $10^3$ dollars!
Why is parallel programming hard?

Race conditions lead to incorrect, non-deterministic behaviour—a nightmare to debug!
• Locking is *error prone*—forgetting to lock leads to errors

• Locking leads to *deadlock* and other concurrency errors

• Locking is *costly*—provokes a *cache miss* (~100 cycles)
It gets worse...

- "Relaxed" memory consistency
Why Functional Programming?

• Data is immutable
  ➔ can be shared without problems!

• No side-effects
  ➔ parallel computations cannot interfere

• Just evaluate everything in parallel!
A Simple Example

\[
\text{nfib :: Integer -> Integer}
\]
\[
\text{nfib n | n<2 = 1}
\]
\[
\text{nfib n = nfib (n-1) + nfib (n-2) + 1}
\]

• A trivial function that returns the number of calls made—and makes a very large number!

<table>
<thead>
<tr>
<th>n</th>
<th>nfib n</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>177</td>
</tr>
<tr>
<td>20</td>
<td>21891</td>
</tr>
<tr>
<td>25</td>
<td>242785</td>
</tr>
<tr>
<td>30</td>
<td>2692537</td>
</tr>
</tbody>
</table>
Compiling Parallel Haskell

• Add a main program
  
  main = print (nfib 40)

• Compile
  
  ghc –O2
  –threaded
  –rtsopts
  –eventlog
  NF.hs

  Enable parallel execution
  Enable run-time system flags
  Enable parallel profiling
Run the code!

- NF.exe
- 331160281
- NF.exe +RTS –N1
- 331160281
- NF.exe +RTS –N2
- 331160281
- NF.exe +RTS –N4
- 331160281
- NF.exe +RTS –N4 –ls
- 331160281

Tell the run-time system to use one core (one OS thread)

Tell the run-time system to collect an event log
Look at the event log!

OBS!
If you have trouble with the latest Haskell platform and threadscope, try Haskell Platform 2012.4
Look at the event log!
What each core was doing:

- Actual useful work
- Cores working: a maximum of one!
- Collecting garbage—in parallel!
Explicit Parallelism

par x y

• "Spark" x in parallel with computing y
  – (and return y)
• The run-time system *may* convert a spark into a parallel task—or it may not
• Starting a task is cheap, but not free
Using par

import Control.Parallel

nfib :: Integer \rightarrow Integer
nfib n | n < 2 = 1
nfib n = \textcolor{red}{\textbf{par}} nf (nf + nfib (n-2) + 1)
  \textbf{where} nf = nfib (n-1)

• Evaluate nf \textit{in parallel with} the body
• Note lazy evaluation: \textbf{where} nf = ... binds nf to an \textit{unevaluated} expression

\textbf{previous nfib called sfib in benchmarks}
Threadscope again...
Benchmarks: nfib 30

- Performance is *worse* for the parallel version
- Performance *worsens* as we use more HECs!
What’s happening?

- There are only four hyperthreads!
- HECs are being scheduled out, waiting for each other...
With 4 HECs

- Looks better (after some GC at startup)
- But let’s zoom in...
• Lots of idle time!
• Very short tasks
Another clue

- Many short-lived tasks
What’s wrong?

\[
\text{nfib n | n < 2 = 1} \\
\text{nfib n = par nf (nf + nfib (n-2) + 1)} \\
\text{where nf = nfib (n-1)}
\]

• Both tasks *start* by evaluating nf!
• One task will *block* almost immediately, and wait for the other
• (In the worst case) *both* may compute nf!
Lazy evaluation in parallel Haskell

```
nfib n = nfib (n-1) + nfib (n-2)
```

$n = 29$

```
nfib (n-1) = 832040
```

$n = 29$

```
nfib (n-2) = Zzz...
```

$n = 29$
Lazy evaluation in parallel Haskell

\[
nfib(n) = 832040 + n = 29
\]
Fixing the bug

\[
\text{rfib } n \mid n < 2 = 1 \\
\text{rfib } n = \text{par nf} \ (\text{rfib } (n-2) + \text{nf} + 1) \\
\text{where nf = rfib } (n-1)
\]

- Make sure we don’t wait for nf until \textit{after} doing the recursive call
• 2 HECs beat sequential performance
• (But hyperthreading is not really paying off)
A bit fragile

rfib n | n < 2 = 1
rfib n = par nf (rfib (n-2) + nf + 1)
where nf = rfib (n-1)

• How do we know + evaluates its arguments left-to-right?
• Lazy evaluation makes evaluation order hard to predict... but we must compute rfib (n-2) first
Explicit sequencing

\texttt{pseq x y}

- Evaluate \texttt{x before y} (and return \texttt{y})

- Used to ensure we get the right evaluation order
rfib with pseq

rfib n | n < 2 = 1
rfib n = par nf1 (pseq nf2 (nf1 + nf2 + 1))
    where nf1 = rfib (n-1)
    nf2 = rfib (n-2)

• Same behaviour as previous rfib... but no longer dependent on evaluation order of +
Actually, GHC 7.8 (which I am using) does not behave as described above for these two versions of rfib. The one with pseq is only slightly faster.

BUT you generally need to be able to use par and pseq!
Spark Sizes

Spark size on a log scale

• Most of the sparks are short
• Spark *overheads* may dominate!
Controlling Granularity

• Let’s go parallel only up to a certain *depth*

```haskell
pfib :: Integer -> Integer -> Integer
pfib 0 n = sfib n
pfib _ n | n < 2 = 1
pfib d n = par nf1 (pseq nf2 (nf1 + nf2) + 1)
  where nf1 = pfib (d-1) (n-1)
        nf2 = pfib (d-1) (n-2)
```
• Two sparks—but uneven lengths leads to waste
Depth 2

- Four sparks, but uneven sizes still leave HECs idle
• 32 sparks
• Much more even distribution of work
Benchmarks (year before last)

Best speedup: 1.9x
On a 4-core i7

![Graph showing speed-up and max values]
Another Example: Sorting

```haskell
qsort [] = []
qsort (x:xs) = qsort [y | y <- xs, y<x] ++ [x] ++ qsort [y | y <- xs, y>=x]
```

- Classic QuickSort
- Divide-and-conquer algorithm
  - Parallelize by performing recursive calls in `//`
  - Exponential `//ism`
Parallel Sorting

```haskell
psort [] = []
psort (x:xs) = par rest $
  psort [y | y <- xs, y<x]
  ++ [x]
  ++ rest
where rest = psort [y | y <- xs, y>=x]
```

- Same idea: name a recursive call and spark it with `par`
- `I know ++ evaluates it arguments left-to-right`
Benchmarking

• Need to run each benchmark many times
  – Run times vary, depending on other activity

• Need to measure carefully and compute statistics

• A *benchmarking library* is very useful
import Criterion.Main

main = defaultMain
    [bench "qsort" (nf qsort randomInts),
     bench "head" (nf (head.qsort) randomInts),
     bench "psort" (nf psort randomInts)]

randomInts =
    take 200000 (randoms (mkStdGen 211570155)) :: [Integer]

• cabal install criterion
• Only a 12% speedup—but easy to get!
• Note how fast head.qsort is!
Results on i7 4-core/8-thread

Best performance with 4 HECs
• Best speedup: 1.39x on four cores
Too lazy evaluation?

- What would happen if we replaced `par rest` by `par (rnf rest)`?

```
psort [] = []
psort (x:xs) = par rest $
  psort [y | y <- xs, y<x]
  ++ [x]
  ++ rest
where rest = psort [y | y <- xs, y>=x]
```
Notice what’s *missing*

- Thread synchronization
- Thread communication
- Detecting termination
- Distinction between shared and private data
- Division of work onto threads
- ...

No •
what’s thread synchronisation
• Thread communication
• Detecting termination
• Distinction between shared and private data
• Division of work onto threads
• ...
Par par everywhere, and not a task to schedule?

• How much speed-up can we get by evaluating *everything* in parallel?

• A ”limit study” simulates a perfect situation:
  – ignores overheads
  – assumes perfect knowledge of which values will be needed
  – infinitely many cores
  – gives an *upper bound* on speed-ups.

• **Refinement**: only tasks > a threshold time are run in parallel.
Limit study results

Figure 4. Implicit parallelism in test programs with different execution thresholds. The y-axis shows the parallelism achieved, so 1 means there was no parallelism but twice that amount means the program was able to parallelize and execute twice as fast.

Some programs have next-to-no parallelism
Some only parallelize with tiny tasks
A few have oodles of parallelism
Amdahl’s Law

- The speed-up of a program on a parallel computer is limited by the time spent in the sequential part
- If 5% of the time is sequential, the maximum speed-up is 20x

- THERE IS NO FREE LUNCH!
References


• *Feedback directed implicit parallelism*, Tim Harris and Satnam Singh. The limit study discussed, and a feedback-directed mechanism to increase its granularity.


• *Real World Haskell*, by Bryan O'Sullivan, Don Stewart, and John Goerzen. The parallel sorting example in more detail.
Book (for the Haskell part)

Simon Marlow’s

http://community.haskell.org/~simonmar/pcph/

(available to read online)
Next

Thursday 10.00 EC

From par and pseq to strategies

Friday  No lecture  (PhD mingle!)
See lecture schedule

TODO  Get started with parallel programming in Haskell. See first exercise and Lab A