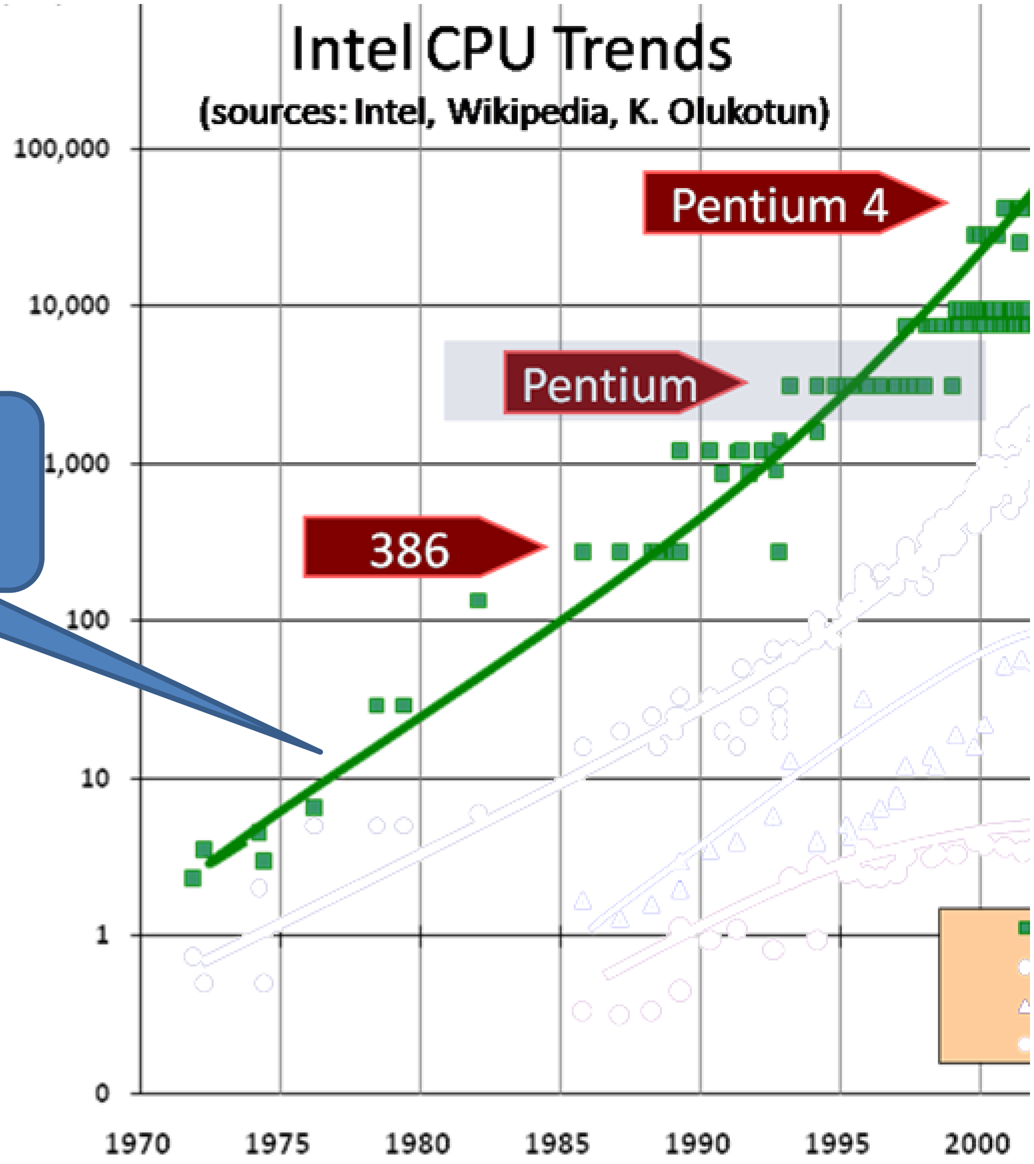
Parallel Functional Programming
Lecture 1

John Hughes

Moore’s Law (1965)

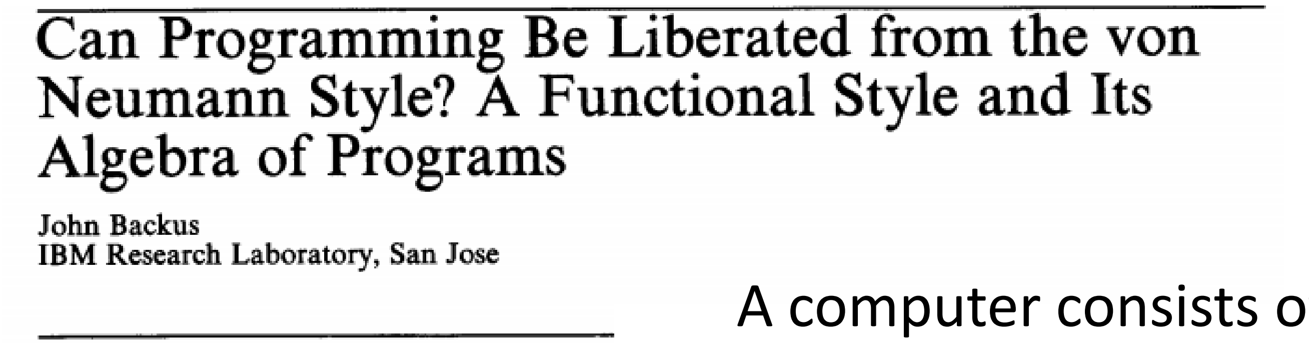
”The number of transistors per chip increases by a factor of two every year”

…two years (1975)



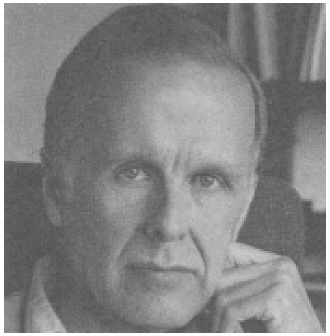
Number of transistors

What shall we do with them all?



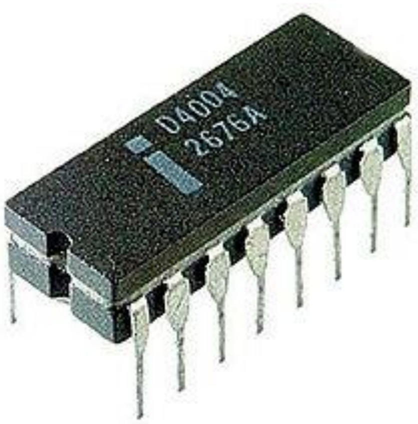
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Turing Award address, 1978

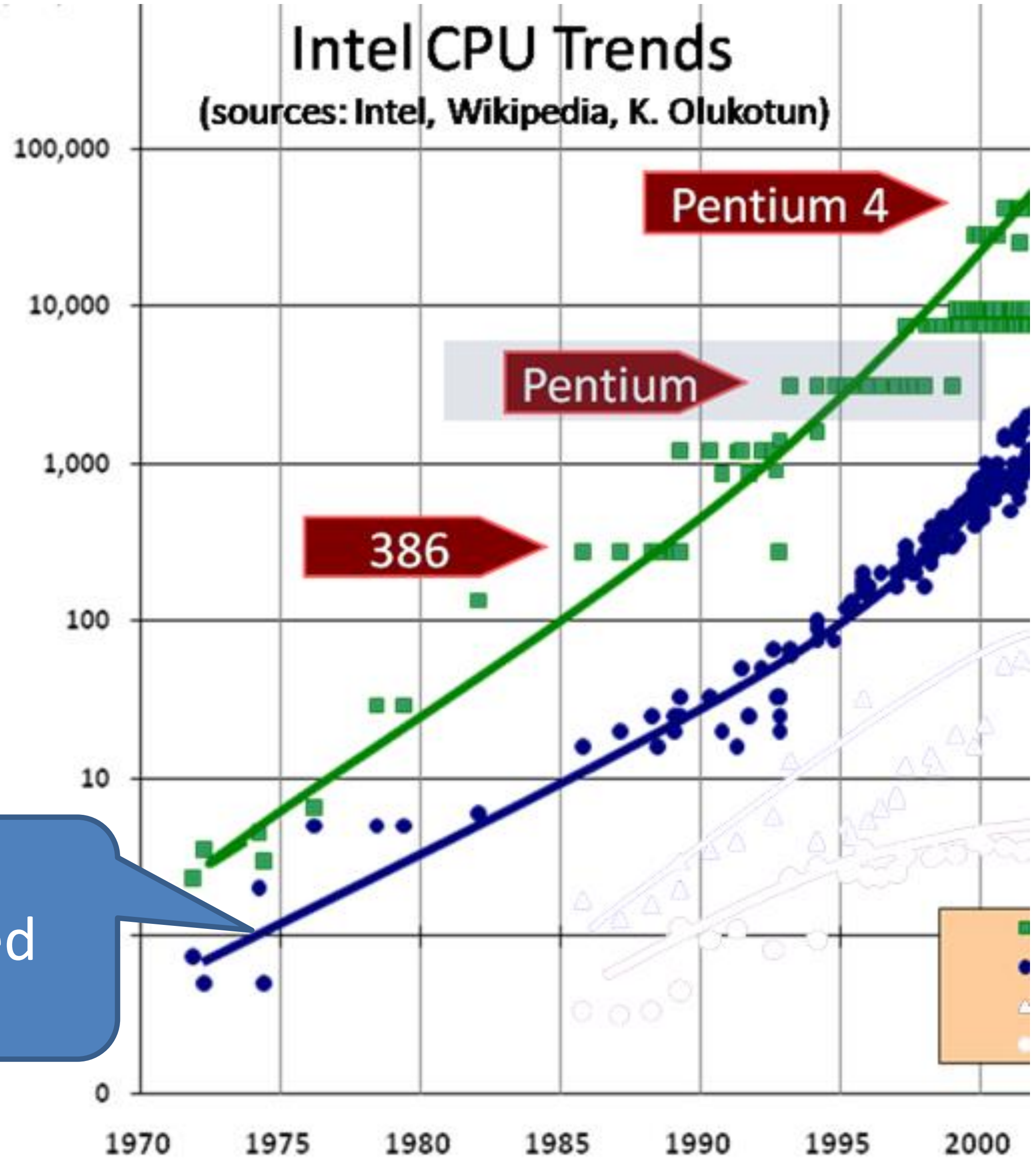
A computer consists of three parts: a central processing unit (or CPU), a store, and a connecting tube that can transmit a single word between the CPU and the store (and send an address to the store). I propose to call this tube the von Neumann bottleneck.



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!!



Clock speed

Smaller transistors switch faster

Pipelined architectures permit faster clocks

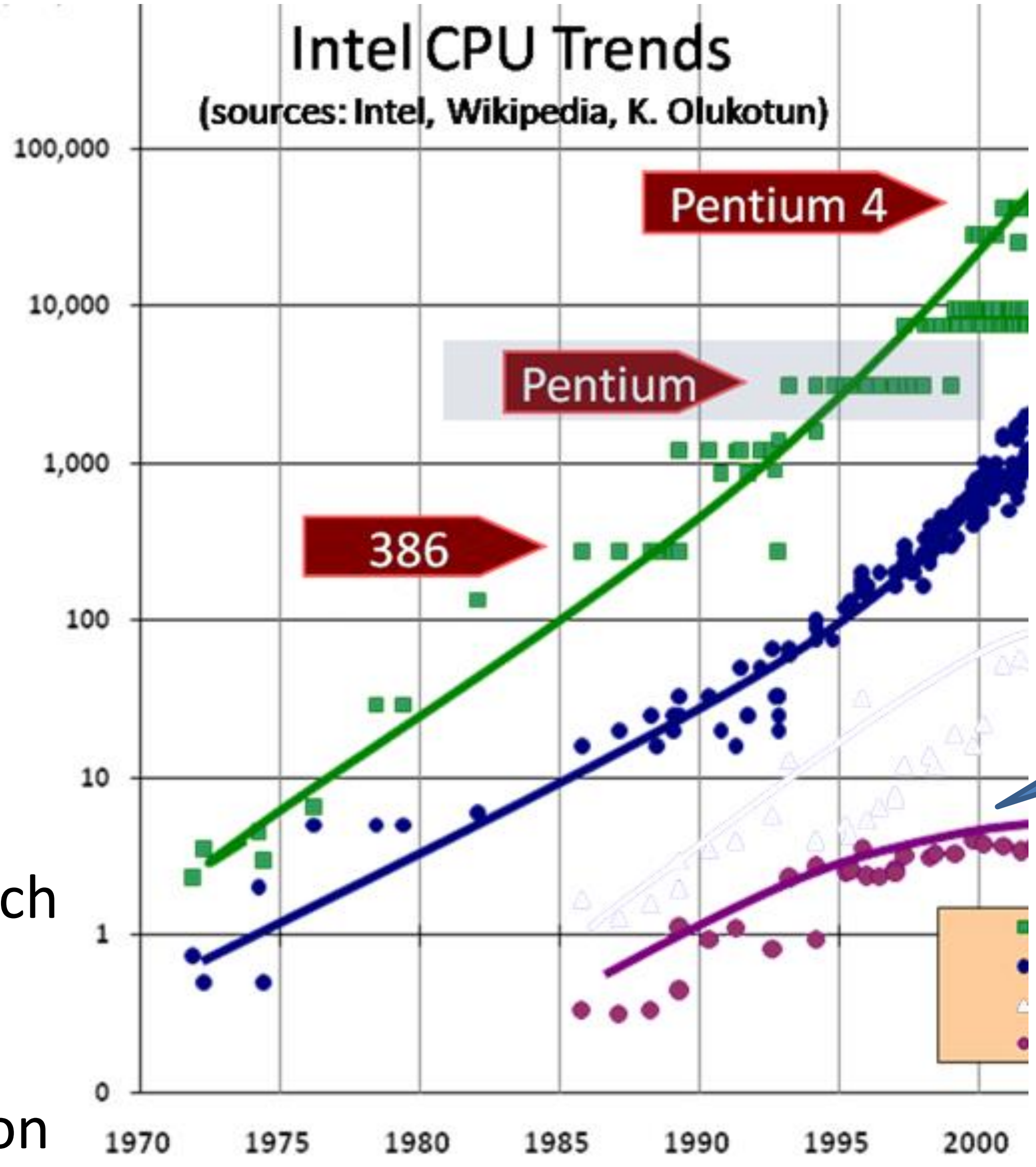
Cache memory

Superscalar processors

Out-of order execution

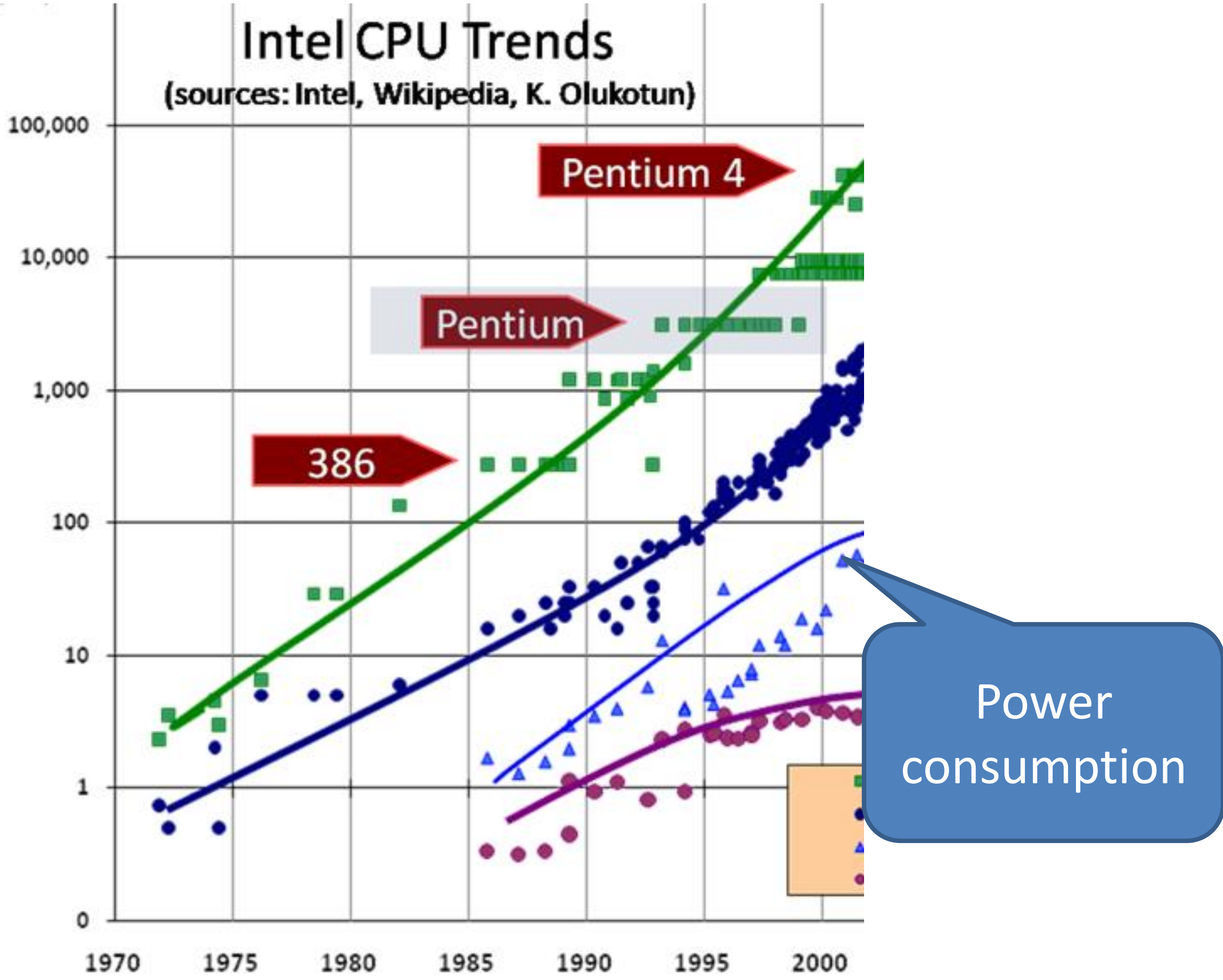
Speculative execution (branch prediction)

Value speculation



Performance per clock

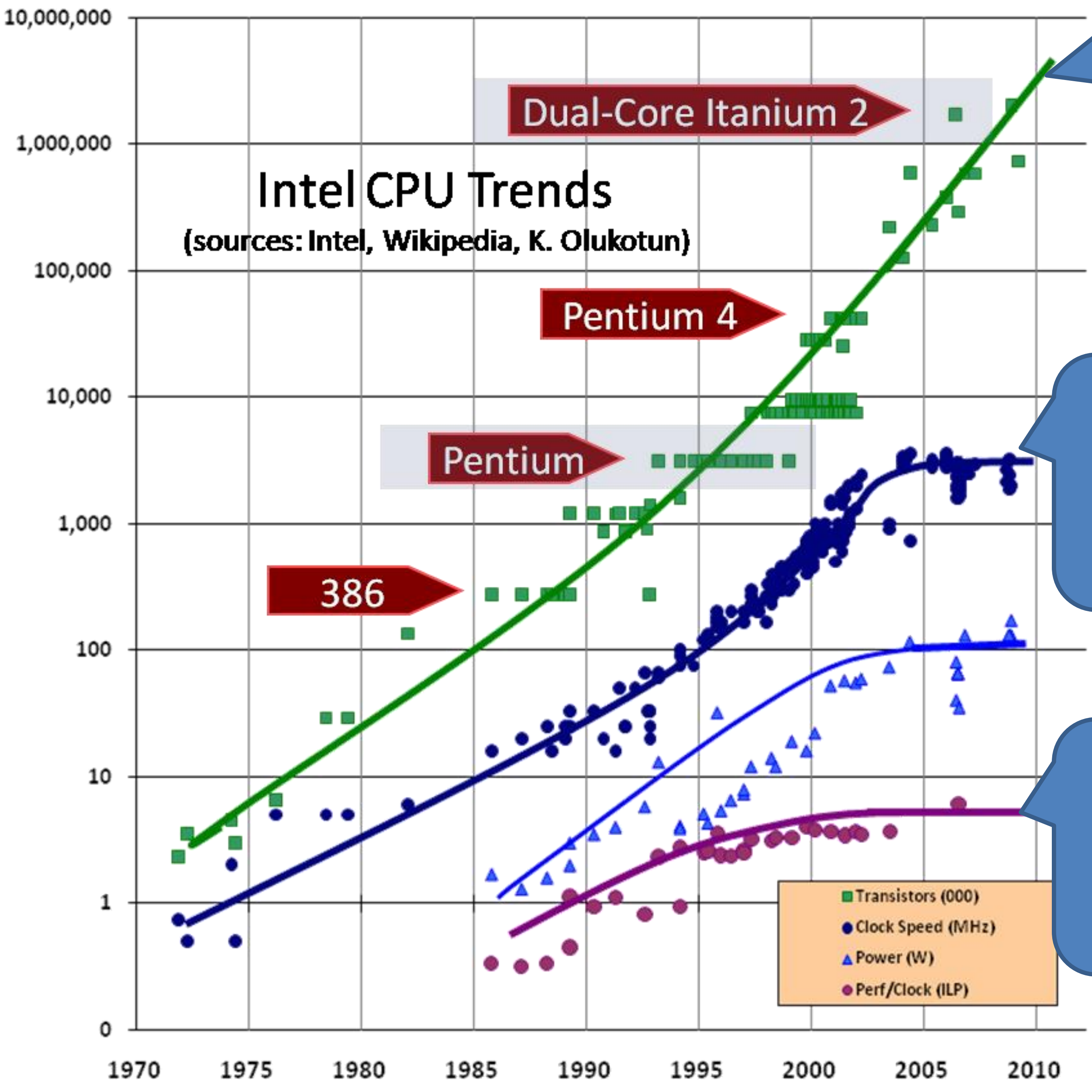
Higher clock frequency ➔ higher power consumption



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



More cores

Stable clock frequency

Stable perf. per clock

Azul Systems Vega 3
Cores per chip: 54
Cores per system: 864

The Future is Parallel

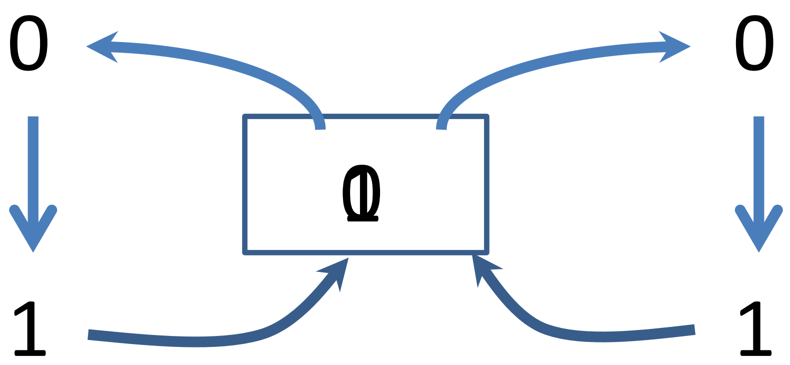
Intel Xeon
10 cores
20 threads

AMD Opteron
16 cores

Tilera Gx-3000
100 cores

Why is parallel programming hard?

`x = x + 1;` || `x = x + 1;`

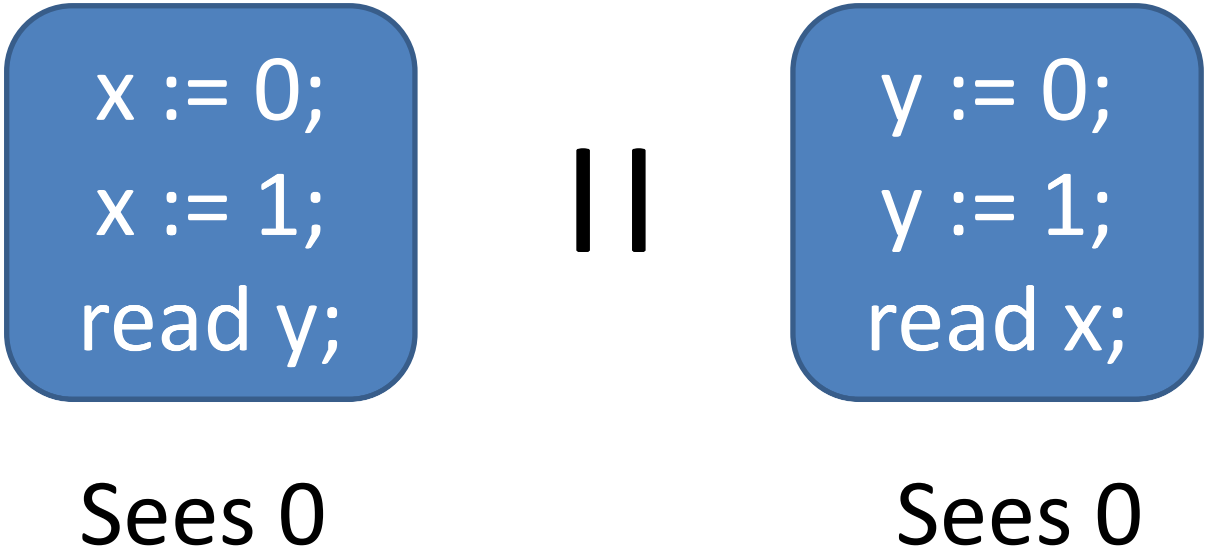


Race conditions lead to *incorrect, non-deterministic* behaviour—a nightmare to debug!

```
x = x + 1;
```

- 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…



```
x := 0;
x := 1;
read y;
```

||

```
y := 0;
y := 1;
read x;
```

Sees 0

Sees 0

- ”Relaxed” memory consistency

Shared Mutable Data

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

```
nfib :: Integer -> Integer
nfib n | n<2 = 1
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!

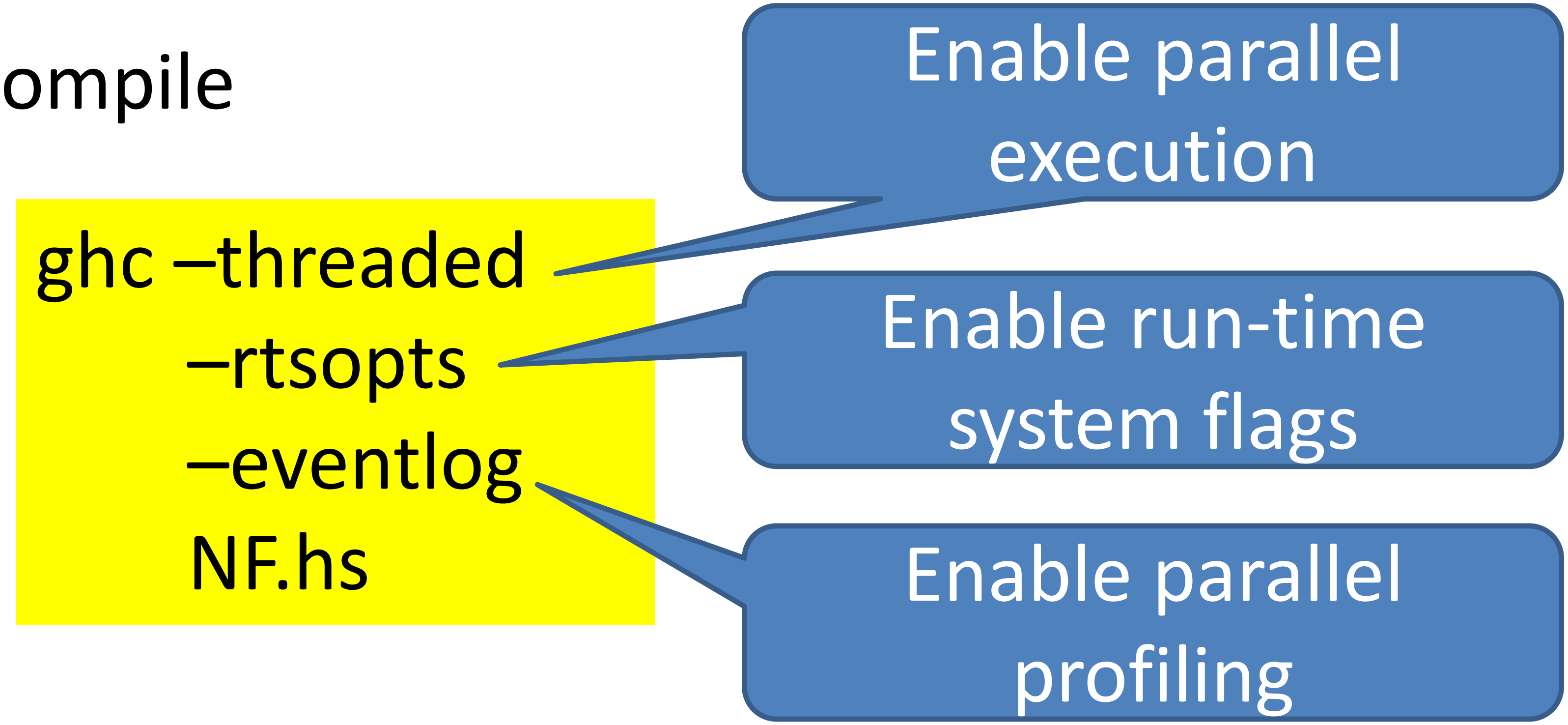
n	nfib n
10	177
20	21891
25	242785
30	2692537

Compiling Parallel Haskell

- Add a main program

```
main = print (nfib 30)
```

- Compile



```
ghc –threaded
      –rtsopts
      –eventlog
      NF.hs
```

Enable parallel execution

Enable run-time system flags

Enable parallel profiling

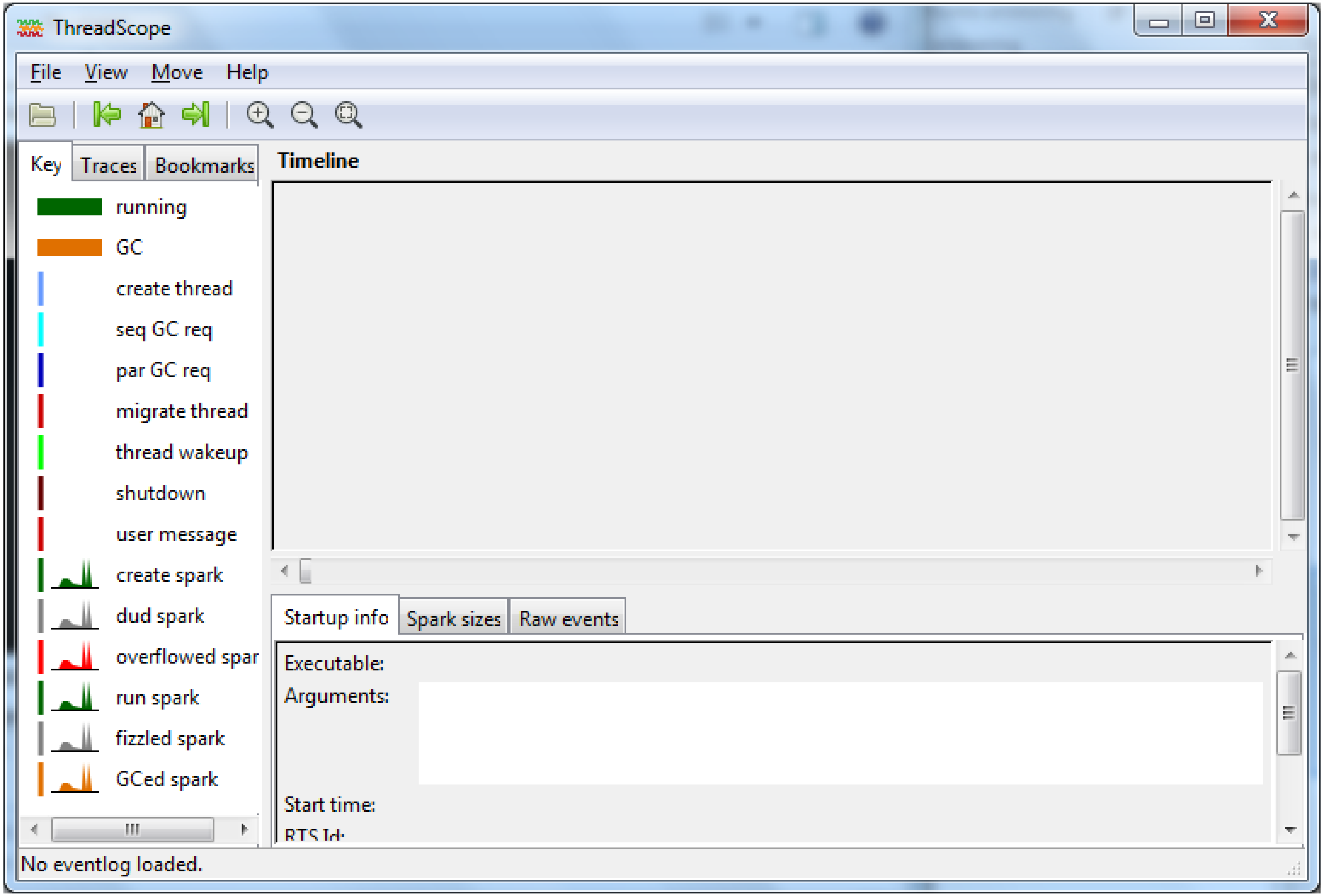
Run the code!

```
➢NF.exe
2692537
➢NF.exe +RTS –N1
2692537
➢NF.exe +RTS –N2
2692537
➢NF.exe +RTS –N4
2692537
➢NF.exe +RTS –N4 –ls
2692537
```

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!



Executable:

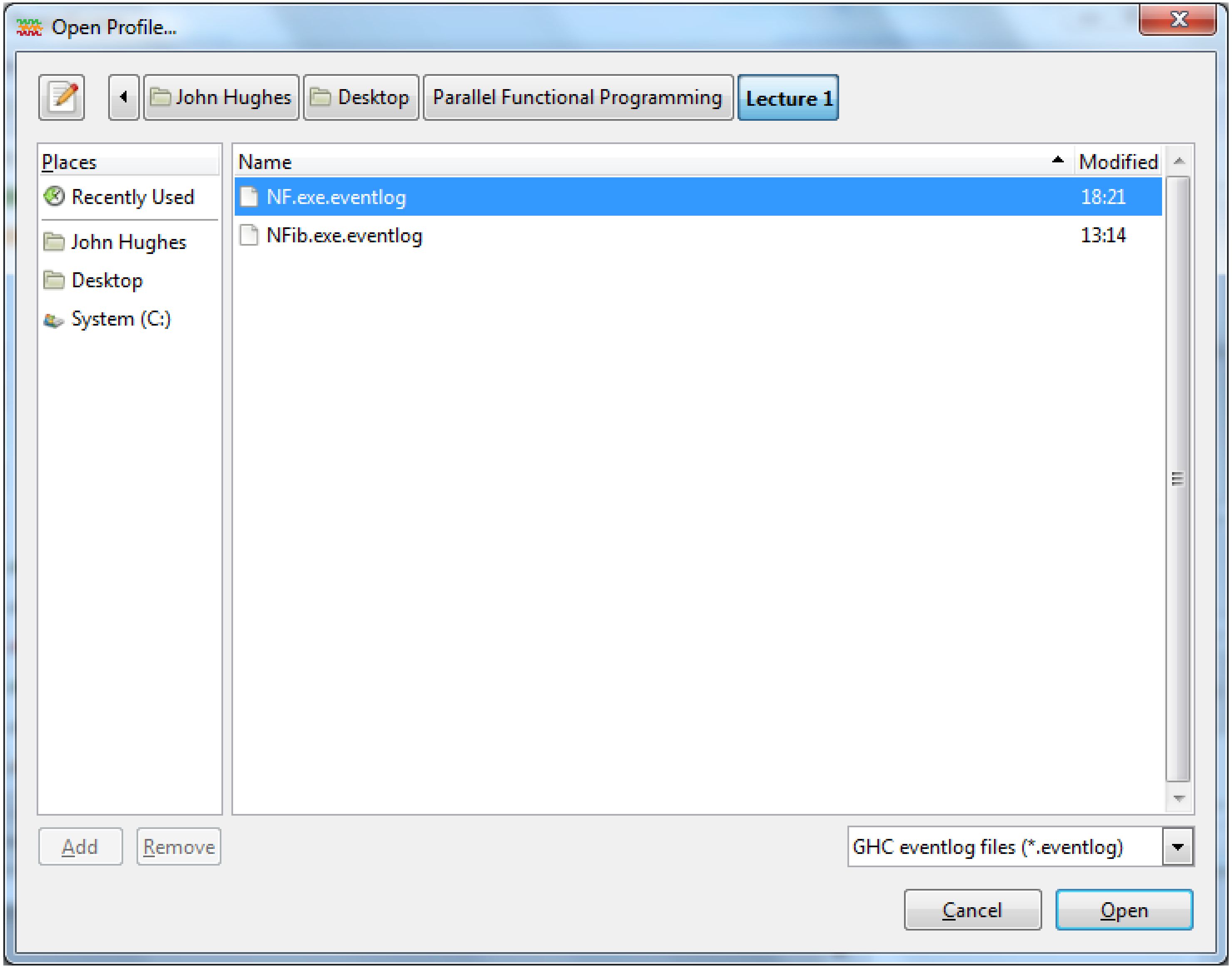
Arguments:

Start time:

RTS Id:

No eventlog loaded.

Look at the event log!

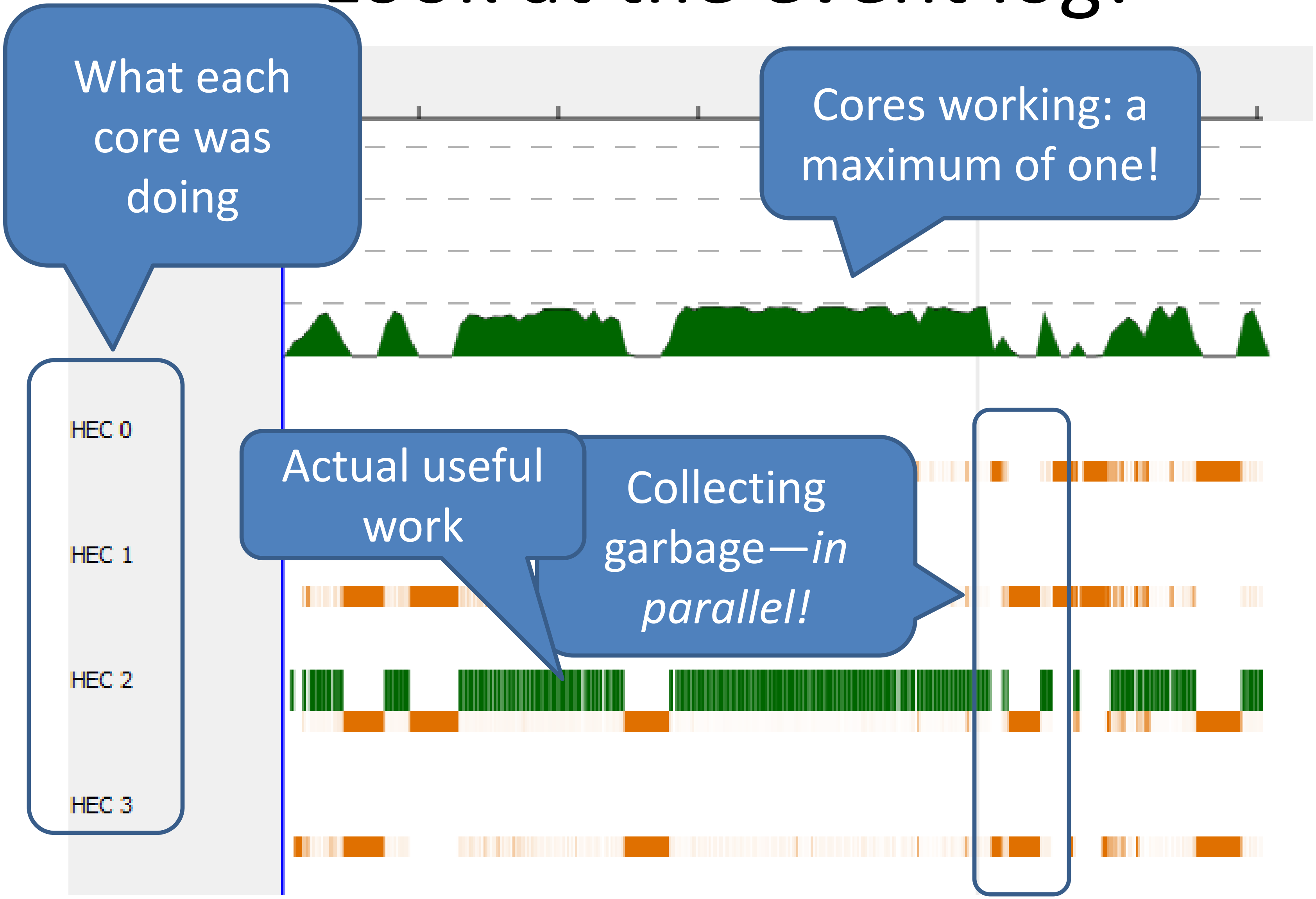


Open Profile...

John Hughes | Desktop | Parallel Functional Programming | Lecture 1

Name	Modified
NF.exe.eventlog	18:21
NFib.exe.eventlog	13:14

Look at the event log!



What each core was doing

Cores working: a maximum of one!

Actual useful work

Collecting garbage—*in parallel!*

Explicit Parallelism

```
par x y
```

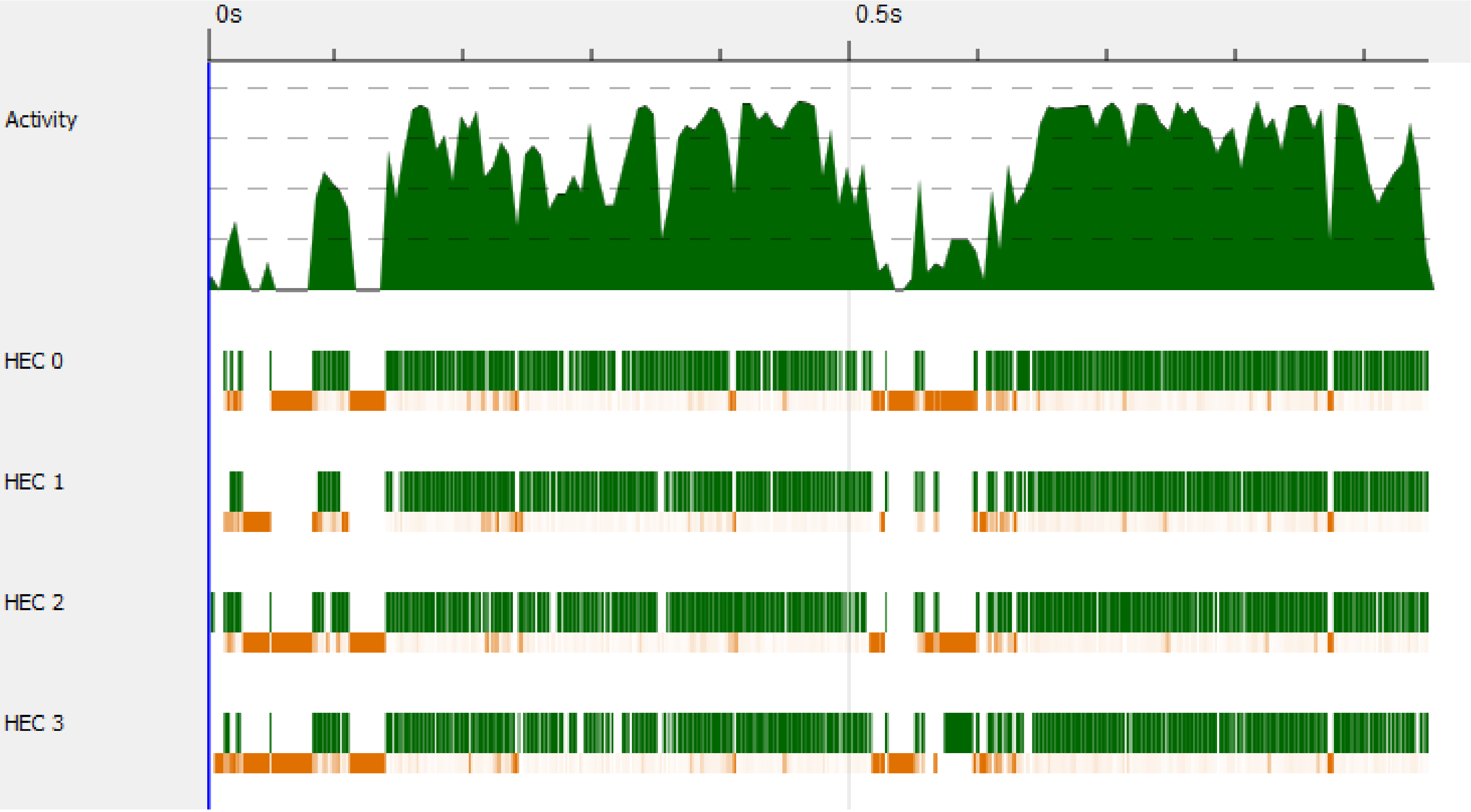
- ”Spark” x in parallel with computing y
  - (and return y)
- The run-time system *may* convert a spark into a parallel task—or it may not
- Starting a task is cheap, but not free

Using par

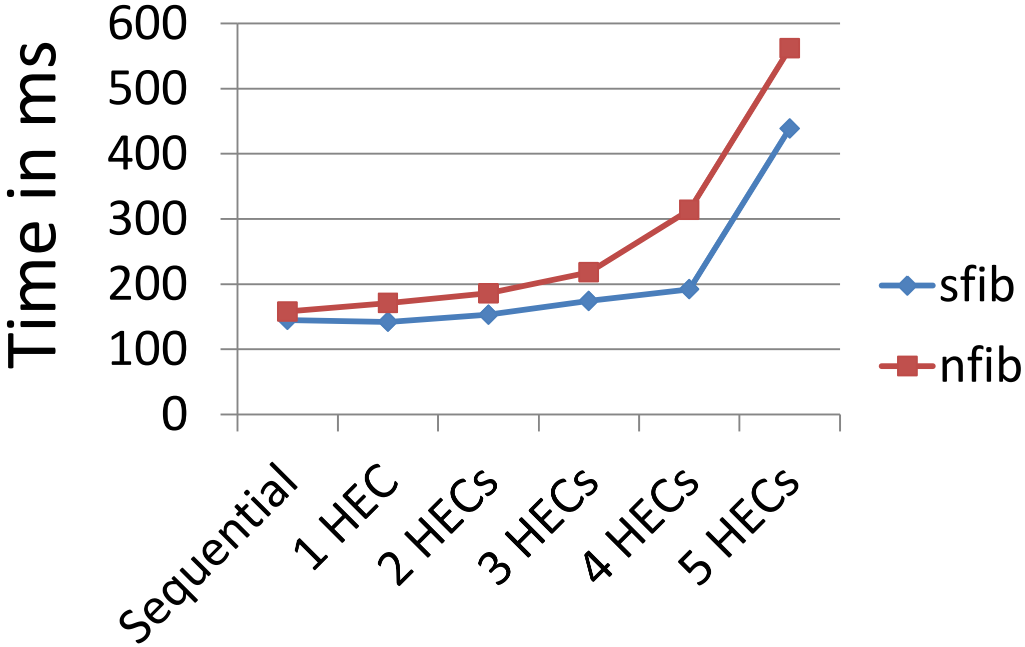
```
import Control.Parallel

nfib :: Integer -> Integer
nfib n | n < 2 = 1
nfib n = par nf (nf + nfib (n-2) + 1)
  where nf = nfib (n-1)
```

- Evaluate nf *in parallel with* the body
- Note lazy evaluation: **where** nf = … binds nf to an *unevaluated* expression

Threadscope again…

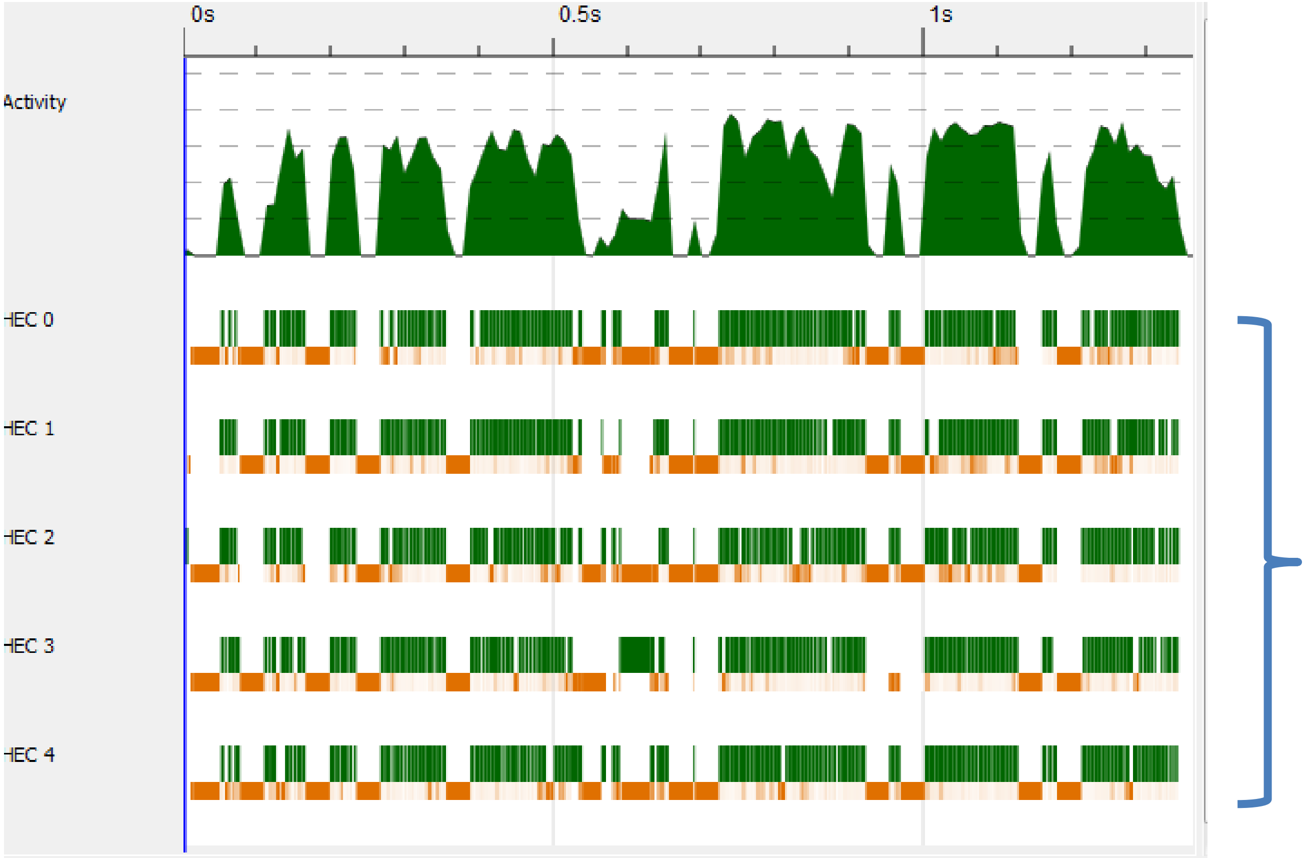


Benchmarks: nfib 30



- Performance is *worse* for the parallel version
- Performance *worsens* as we use more HECs!

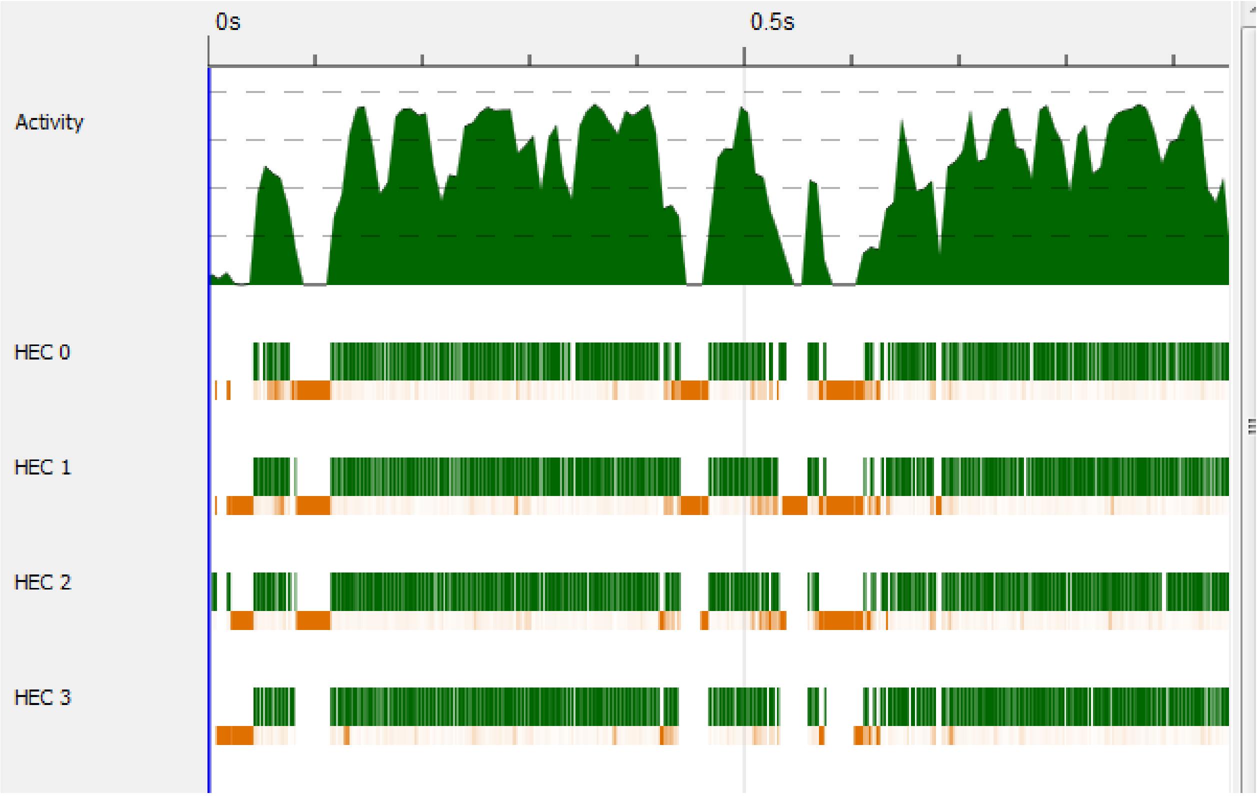
What’s happening?



5 HECs

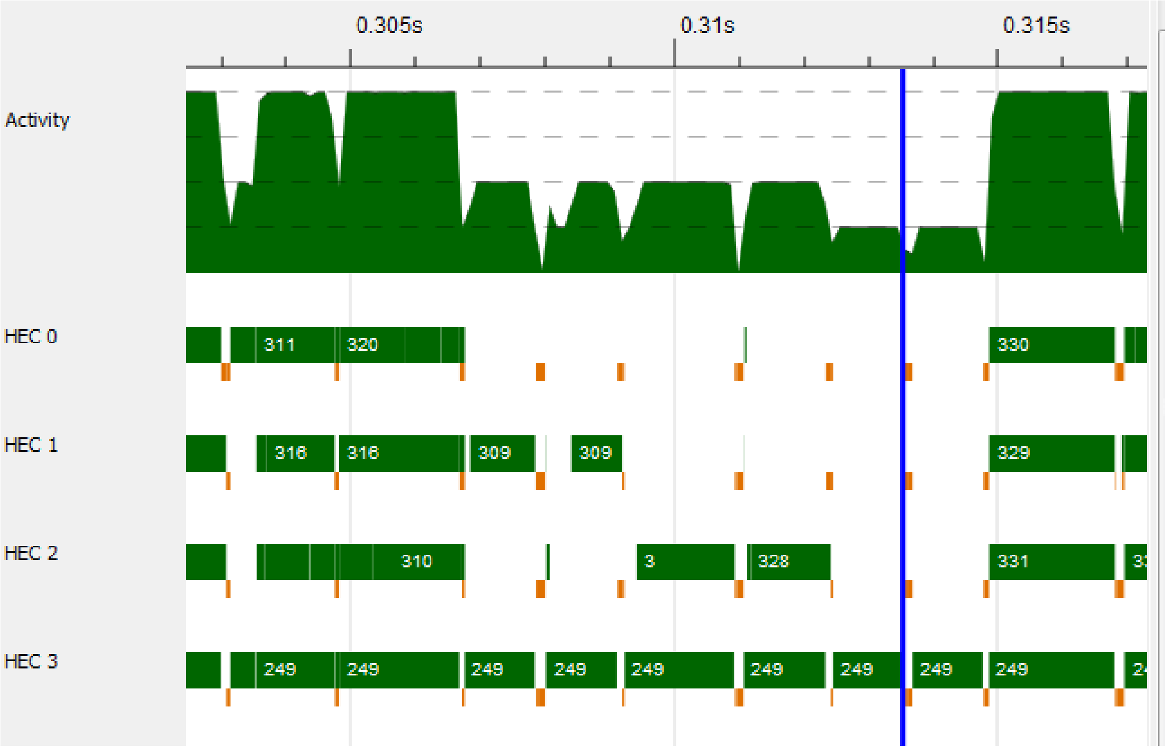
- There *are* only four hyperthreads!
- HECs are being scheduled out, waiting for each other…

With 4 HECs



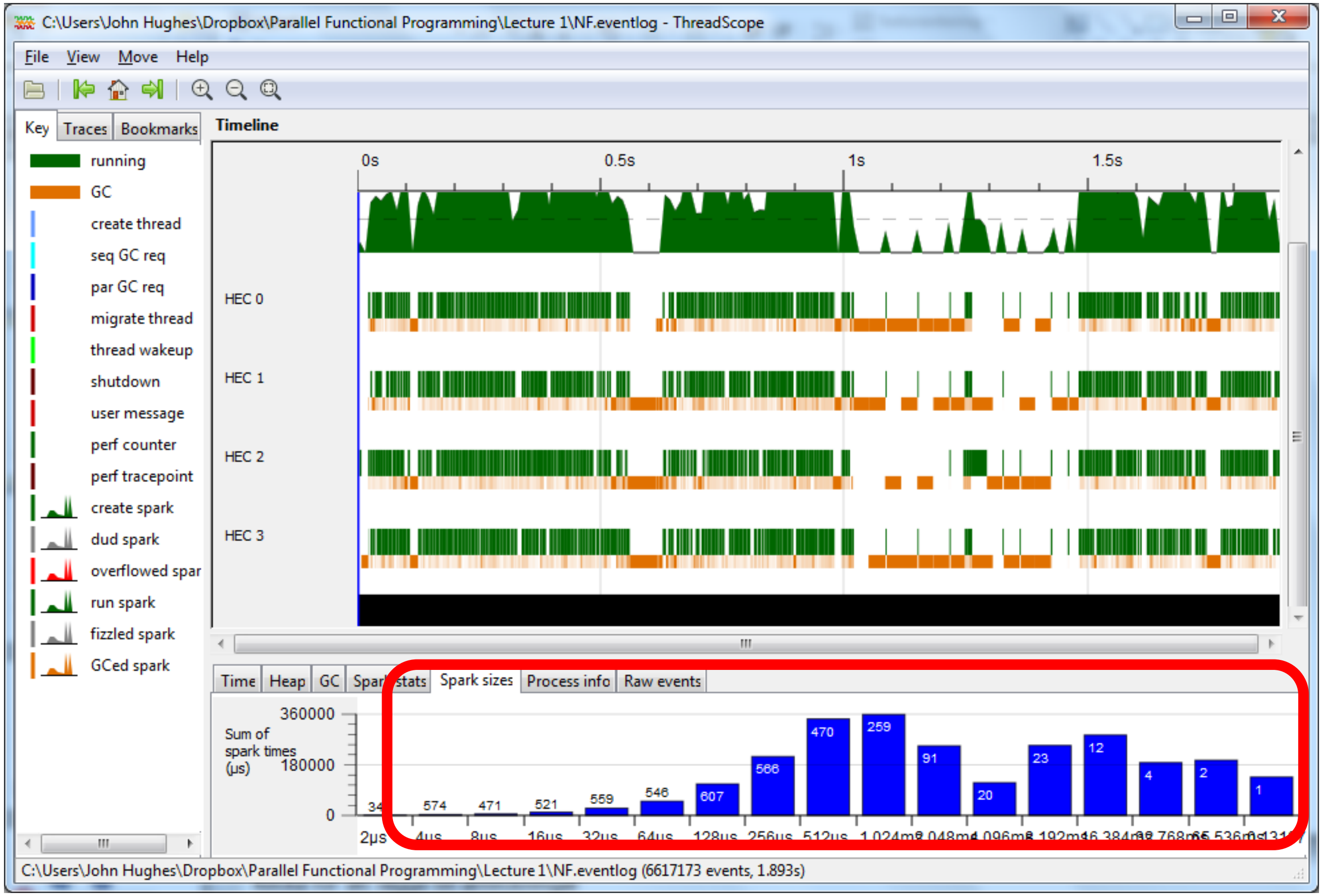
- Looks better (after some GC at startup)
- But let’s zoom in…

Detailed profile



- Lots of idle time!
- Very short tasks

Another clue



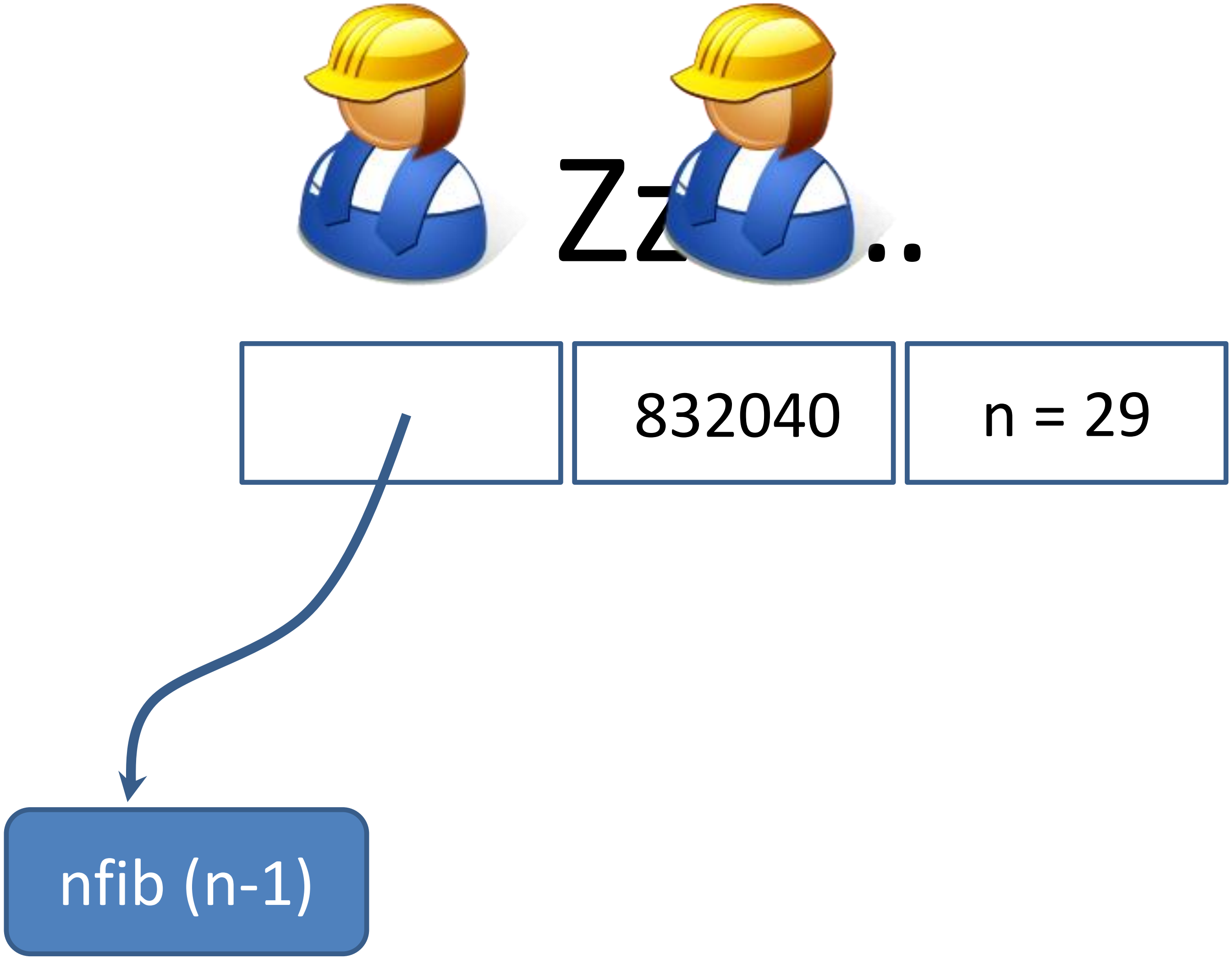
- Many short-lived tasks

What’s wrong?

```
nfib n | n < 2 = 1
nfib n = par nf (nf + nfib (n-2) + 1)
  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



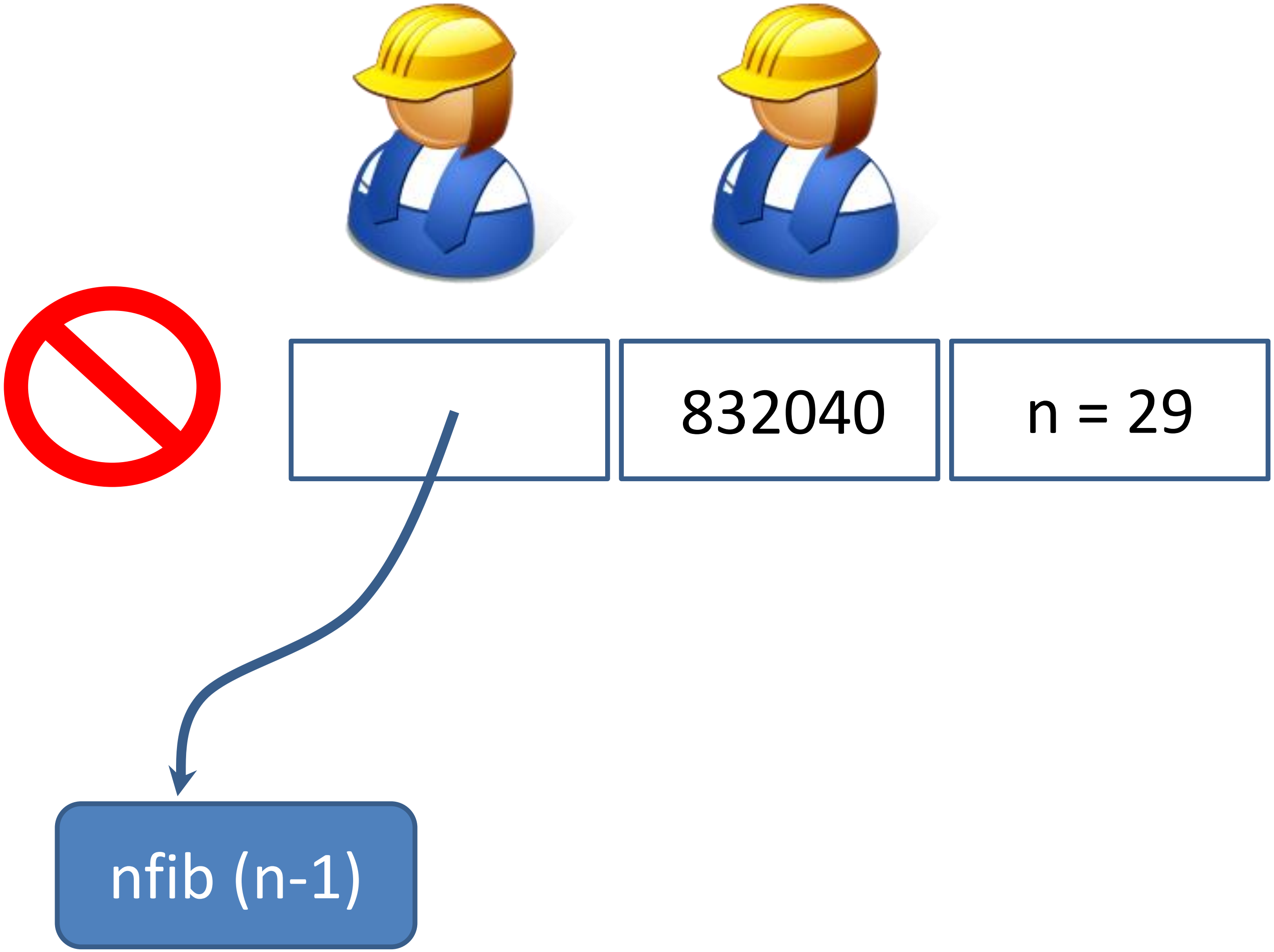
Zz ..

832040

n = 29

nfib (n-1)

Lazy evaluation in parallel Haskell



	832040	n = 29

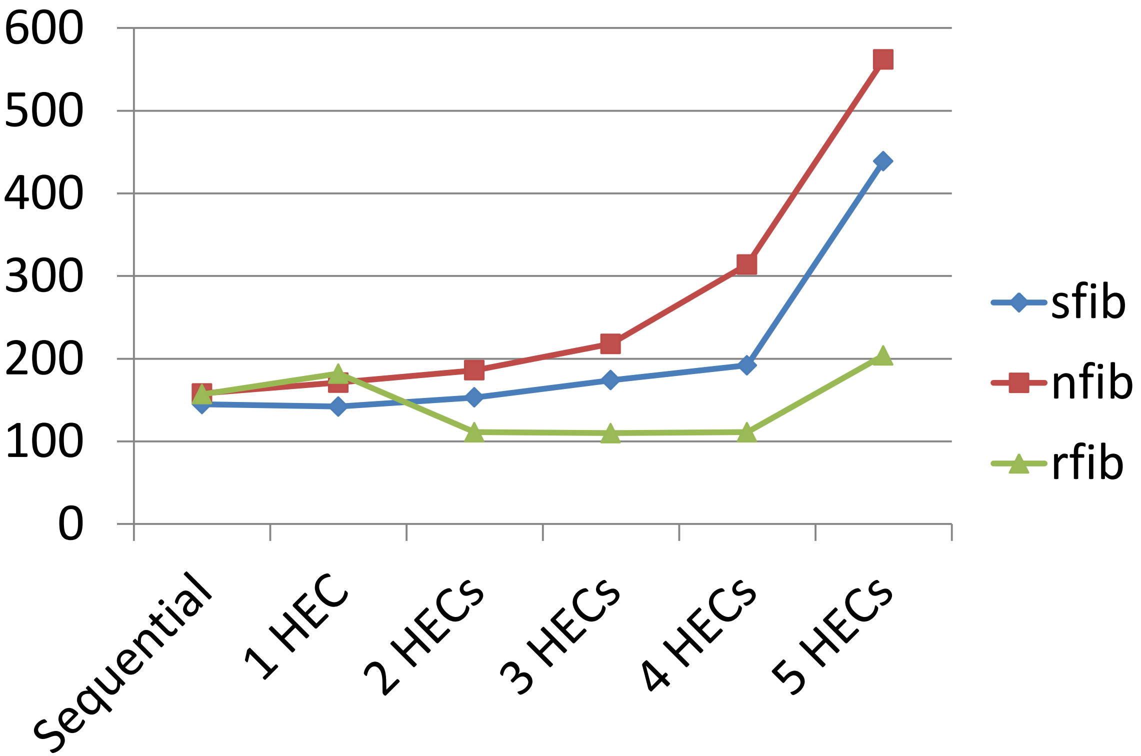
nfib (n-1)

Fixing the bug

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

- Make sure we don’t wait for nf until *after* doing the recursive call

Much better!



- 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

```
pseq x y
```

- Evaluate x *before* y (and return y)

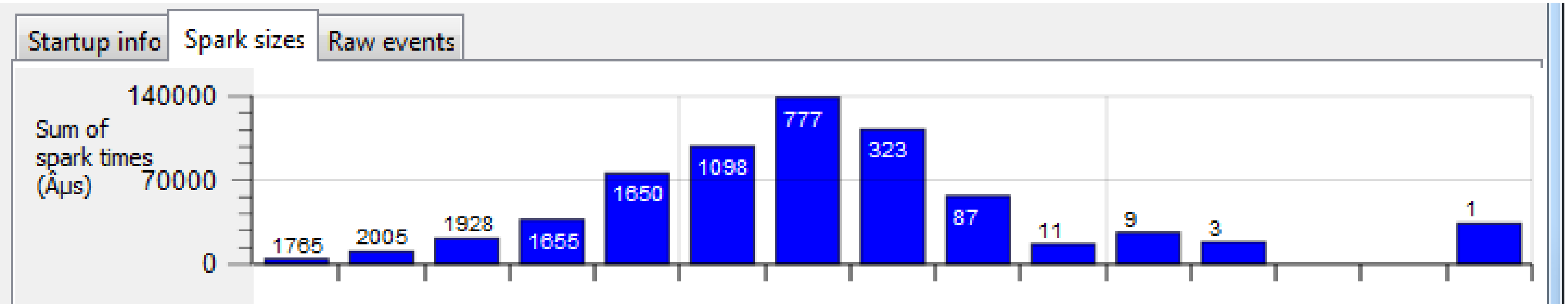
- Used to *ensure* we get the right evaluation order

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 +

Spark Sizes



Spark size on a log scale

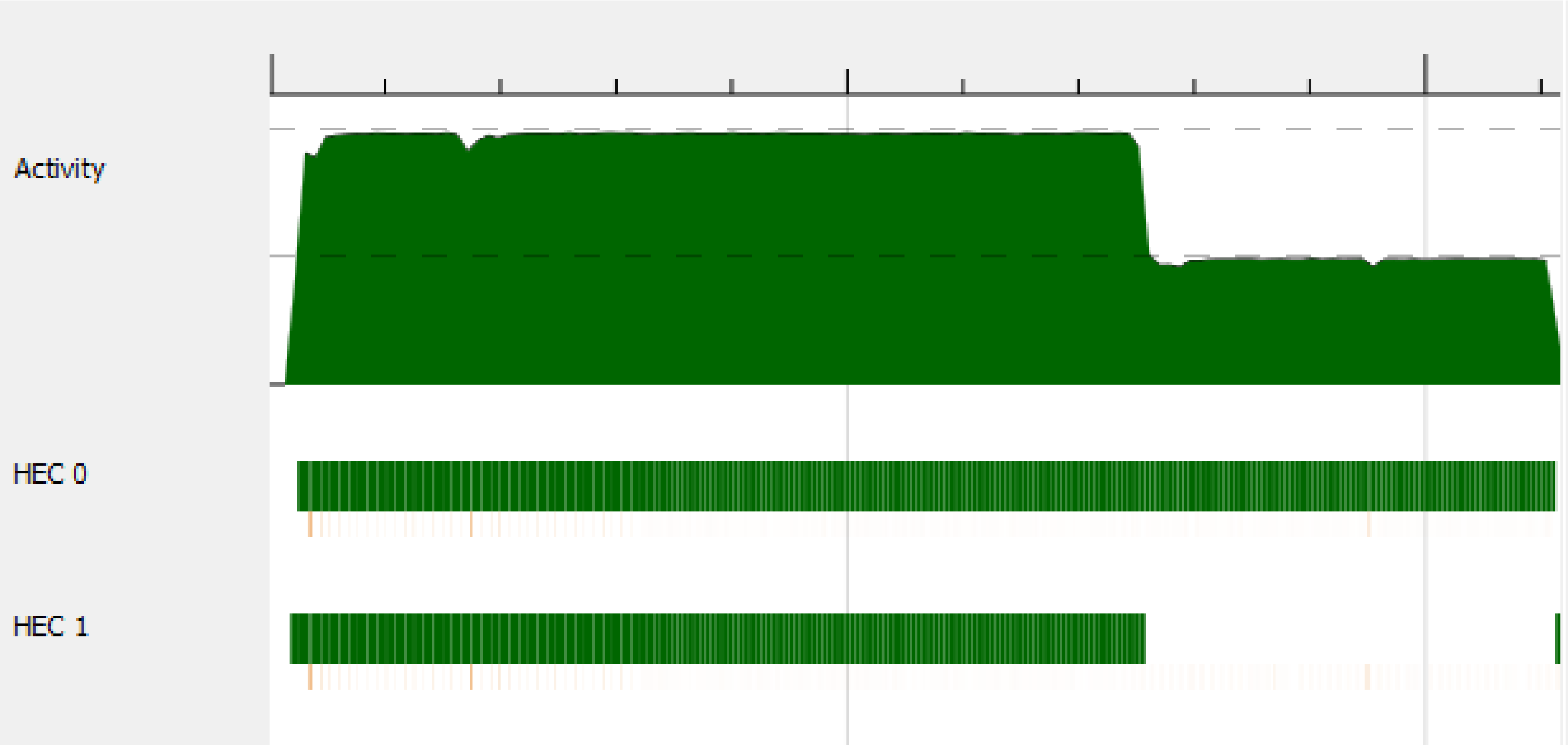
- Most of the sparks are *short*
- Spark *overheads* may dominate!

Controlling Granularity

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

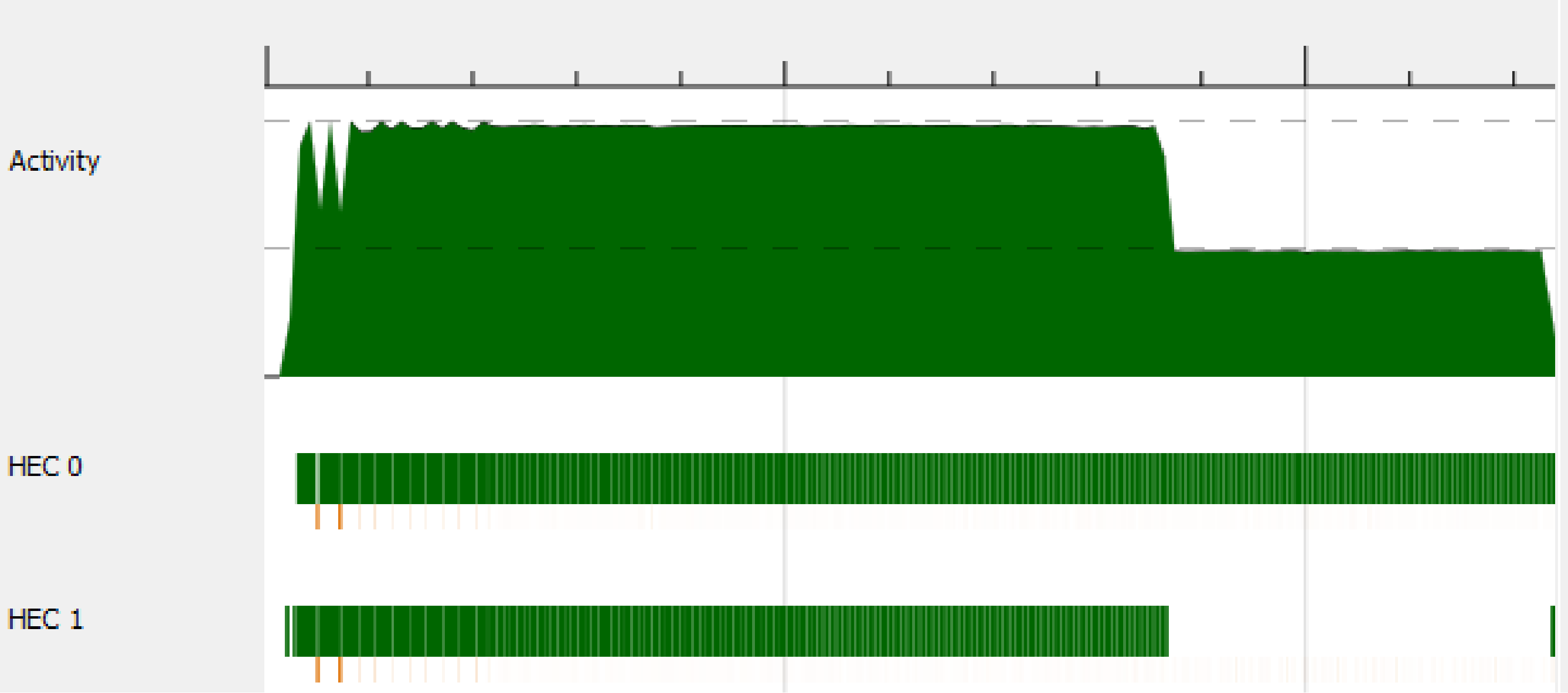
```
pfib :: Integer -> Integer -> Integer
pfib 0 n = sfib n
pfib _ n | n < 2 = 1
pfib d n = par nf1 (pseq nf2 (nf1 + nf2) + 1)
  where nf1 = pfib (d-1) (n-1)
        nf2 = pfib (d-1) (n-2)
```

Depth 1



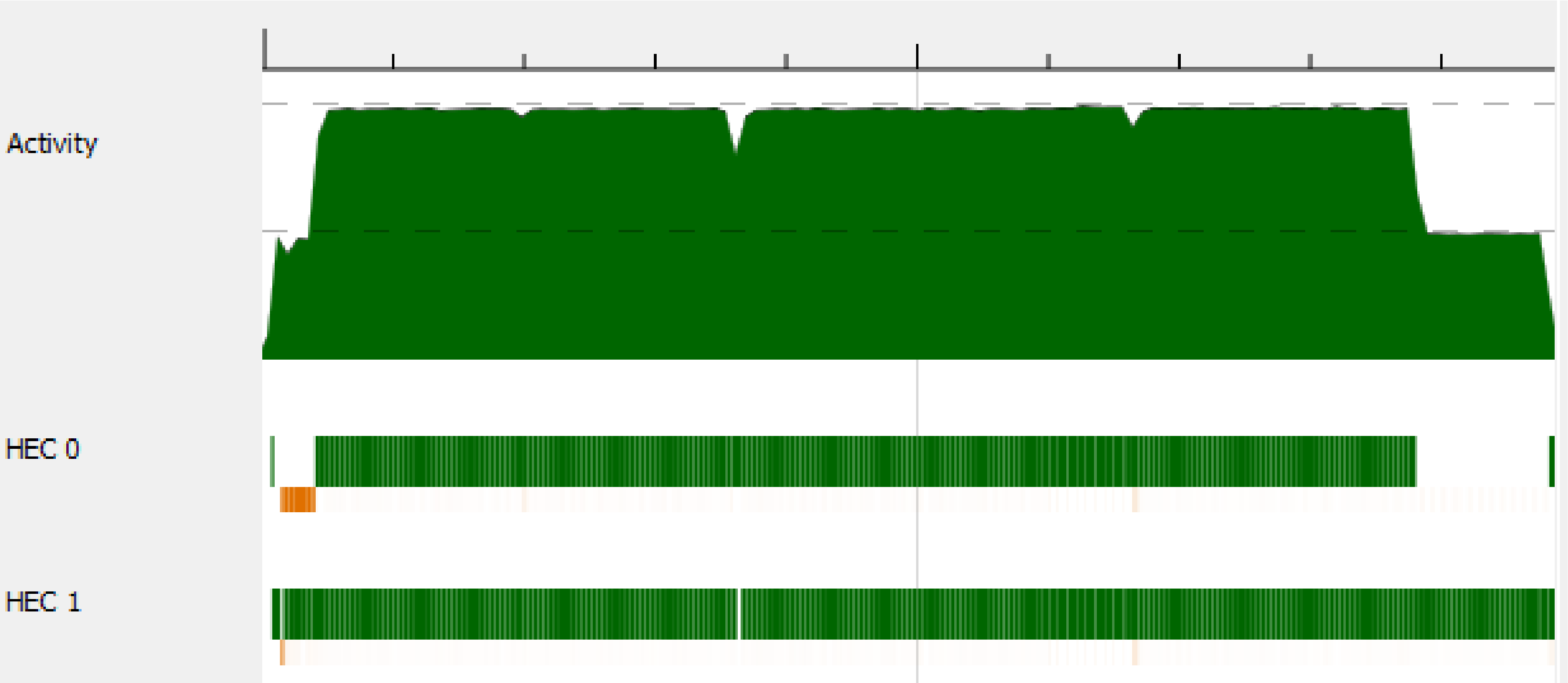
- Two sparks—but uneven lengths leads to waste

Depth 2



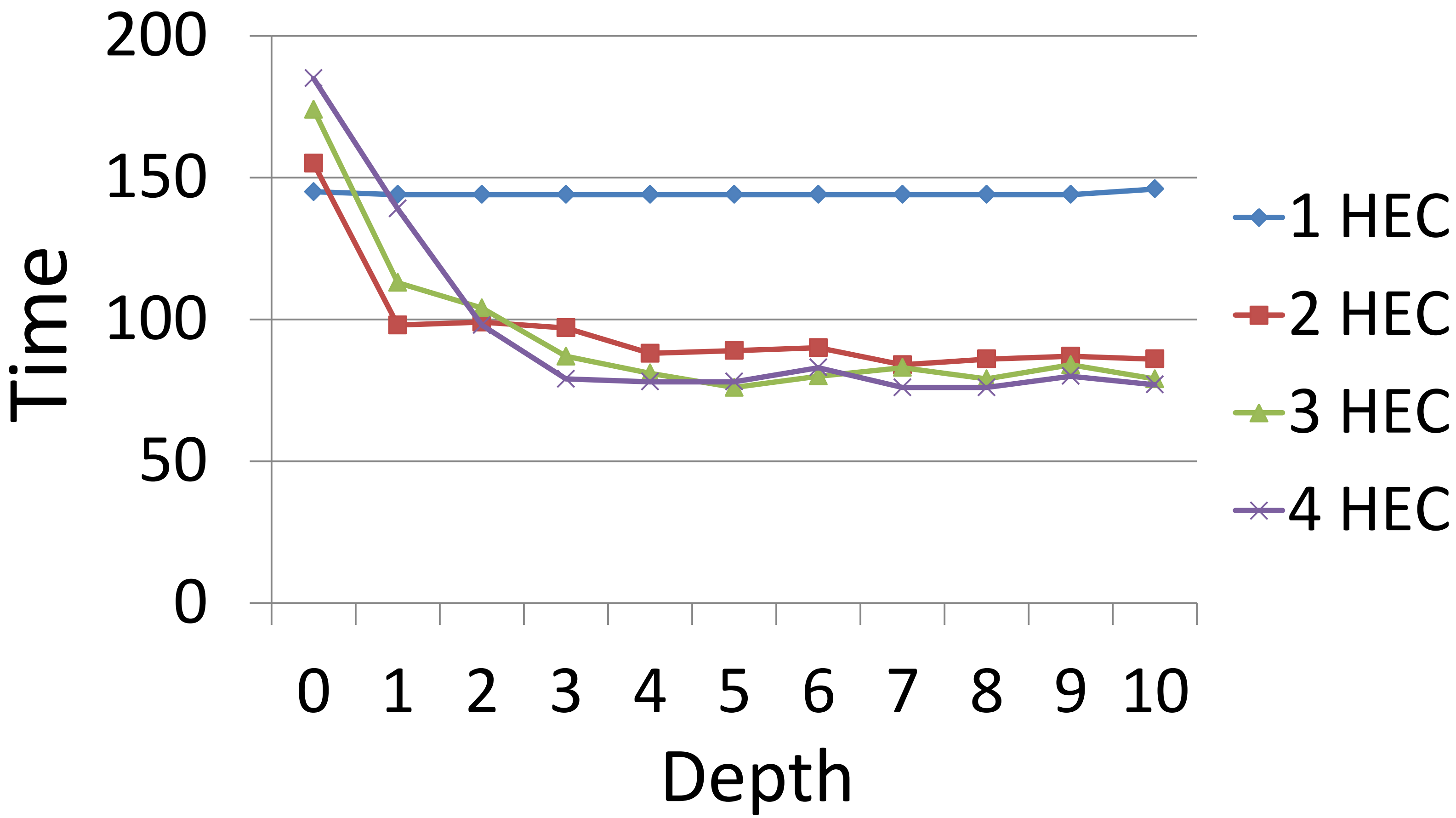
- Four sparks, but uneven sizes still leave HECs idle

Depth 5



- 32 sparks
- Much more even distribution of work

Benchmarks



Best speedup: 1.9x

Another Example: Sorting

```
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

```
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

Criterion

Name a benchmark

Import the

Run a list of benchmarks

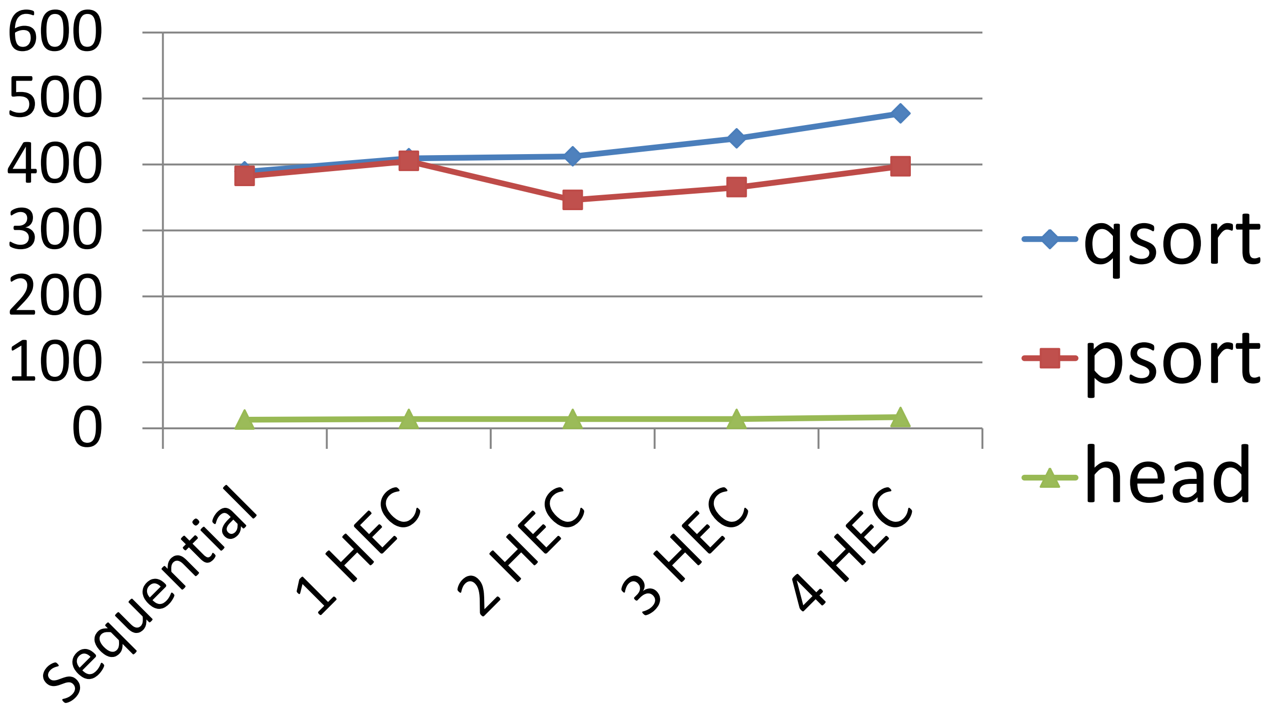
```
import Crit    ion.Main

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

randomInts =
   take 200000 (randoms (mk
      :: [Integer]
```

Call fun on arg *and evaluate result*

Generate a *fixed* list of random integers as test data

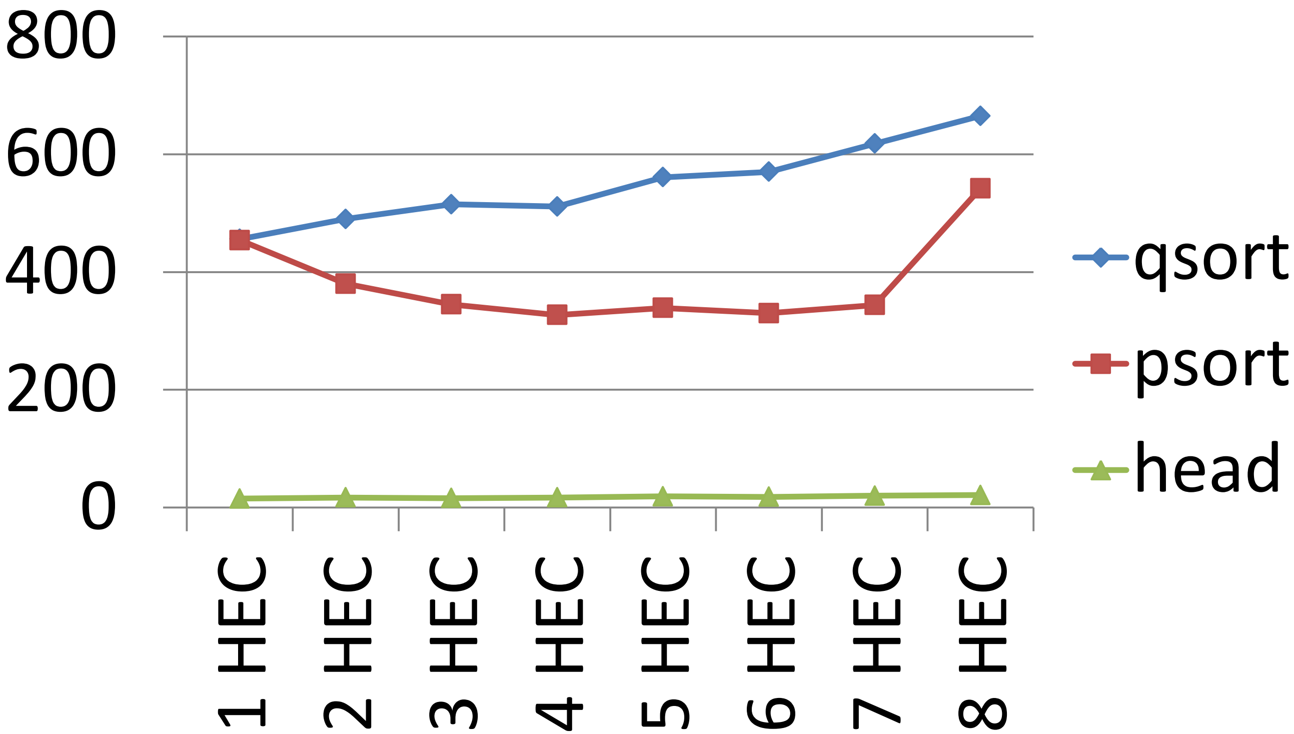
- cabal install criterion

Results



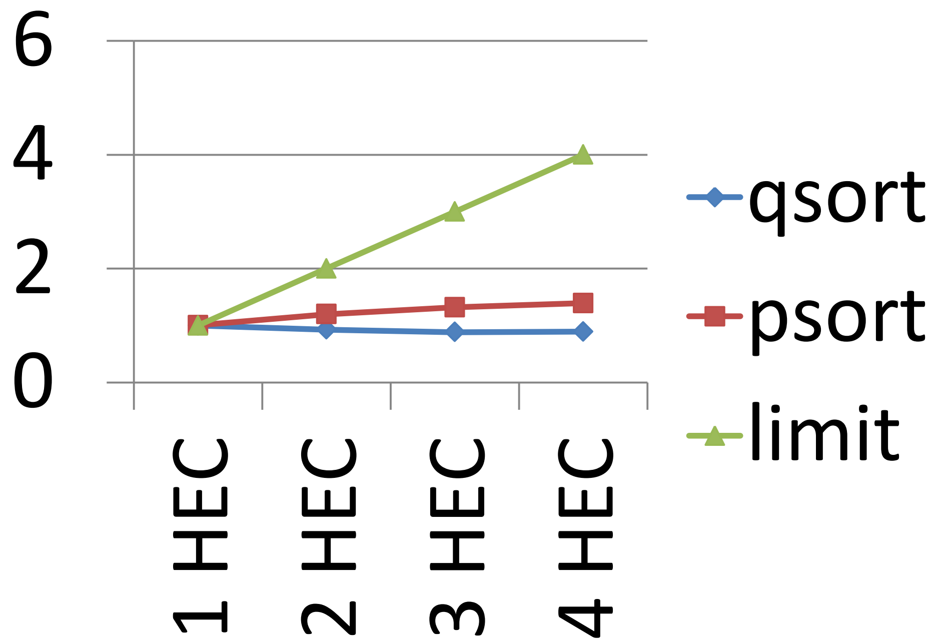
- Only a 12% speedup—but easy to get!
- Note how fast head.qsort is!

Results on i7 4-core/8-thread



Best performance with 4 HECs

Speedup on i7 4-core



- Best speedup: 1.39x on four cores

Too lazy evaluation?

This only evaluates the *first constructor* of the list!

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

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

Notice what's *missing*

- Thread synchronization
- 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

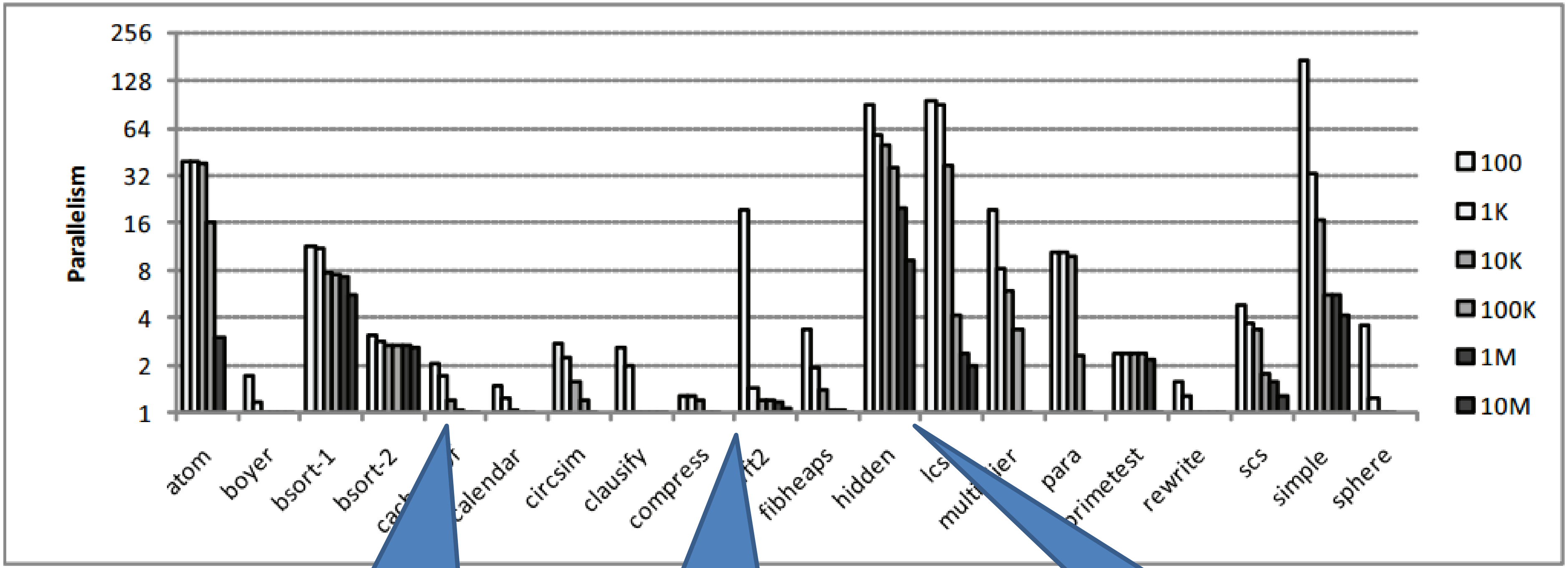


Figure 4. Implicit parallelism st programs with execution thresholds. The y-axis e parallelism achieved, so 1 mea 'twic

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

- *Haskell on a shared-memory multiprocessor,* Tim Harris, Simon Marlow, Simon Peyton Jones, Haskell Workshop, Tallin, Sept 2005. The first paper on multicore Haskell.

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

- *Runtime Support for Multicore Haskell*, Simon Marlow, Simon Peyton Jones, and Satnam Singh. ICFP'09. An overview of GHC's parallel runtime, lots of optimisations, and lots of measurements.

- *Real World Haskell*, by Bryan O'Sullivan, Don Stewart, and John Goerzen. The parallel sorting example in more detail.

Just for fun

- For the fastest Haskell matrix multiplication
- Using par, pseq, but *no mutable data*
- Multiplying two random 200x200 matrices given as lists—[[Int64]]
- On a 4-core Intel i7 with 4 HECs
- **Entries:** by midnight on Monday 8th April