The future is parallel The future of parallel is declarative

Simon Peyton Jones
Microsoft Research

Thesis

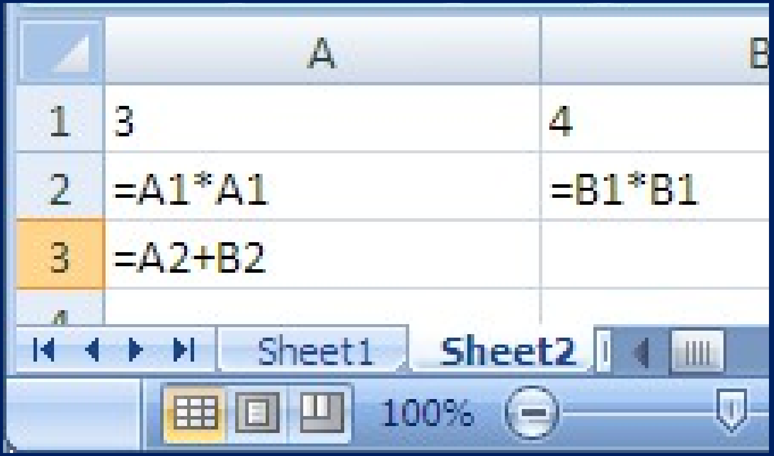
- The free lunch is over. Muticores are here. We have to program them. This is hard. Yada-yada-yada.
- Programming parallel computers
  - **Plan A**. Start with a language whose computational fabric is by-default sequential, and by heroic means make the program parallel
  - **Plan B**. Start with a language whose computational fabric is by-default parallel
- Every successful large-scale application of parallelism has been largely declarative and value-oriented
  - SQL Server
  - LINQ
  - Map/Reduce
  - Scientific computation
- **Plan B will win**. Parallel programming will increasingly mean functional programming

Any effect ← Spectrum → **Pure (no effects)**

C, C++, Java, C#, VB

Excel, Haskell

```
X := In1
X := X*X
X := X + In2*In2
```



	A	B
1	3	4
2	=A1*A1	=B1*B1
3	=A2+B2	

Commands, control flow

- Do this, then do that
- "X" is the name of a cell that has different values at different times

Expressions, data flow

- No notion of sequence
- "A2" is the name of a (single) value

Imperative

C, C++, Java, C#, VB

```
X := In1
X := X*X
X := X + In2*In2
```

Computational model:

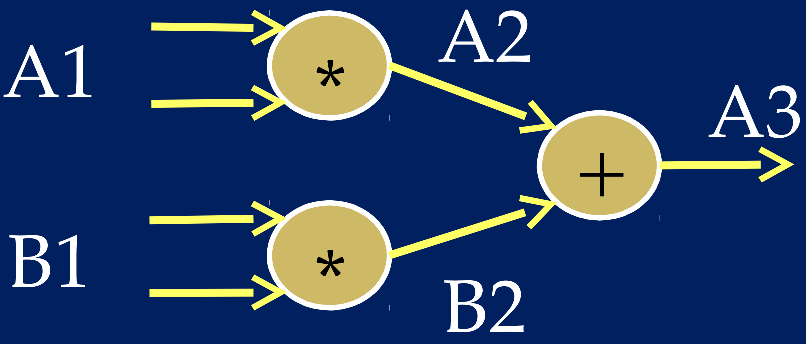
- program counter
- mutable state

Inherently sequential

Commands, control flow

- Do this, then do that
- "X" is the name of a cell that has different values at different times

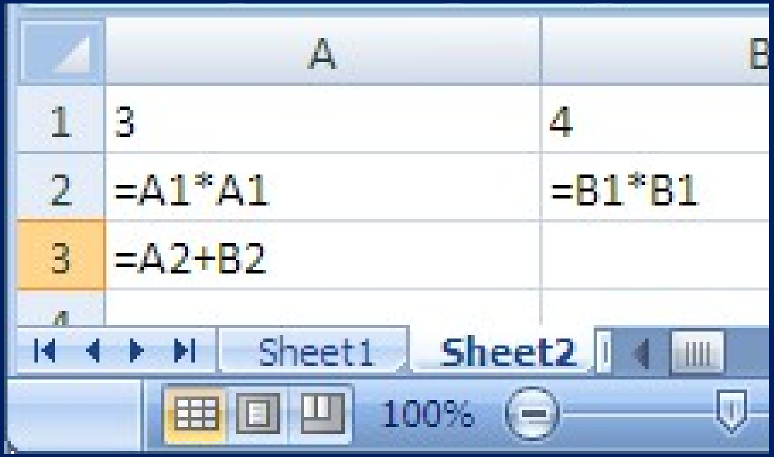
Functional



A2 = A1*A1
B2 = B1*B1
A3 = A2+B2

Computational model: expression evaluation
Inherently parallel

Excel, Haskell



	A	B
1	3	4
2	=A1*A1	=B1*B1
3	=A2+B2	

Expressions, data flow

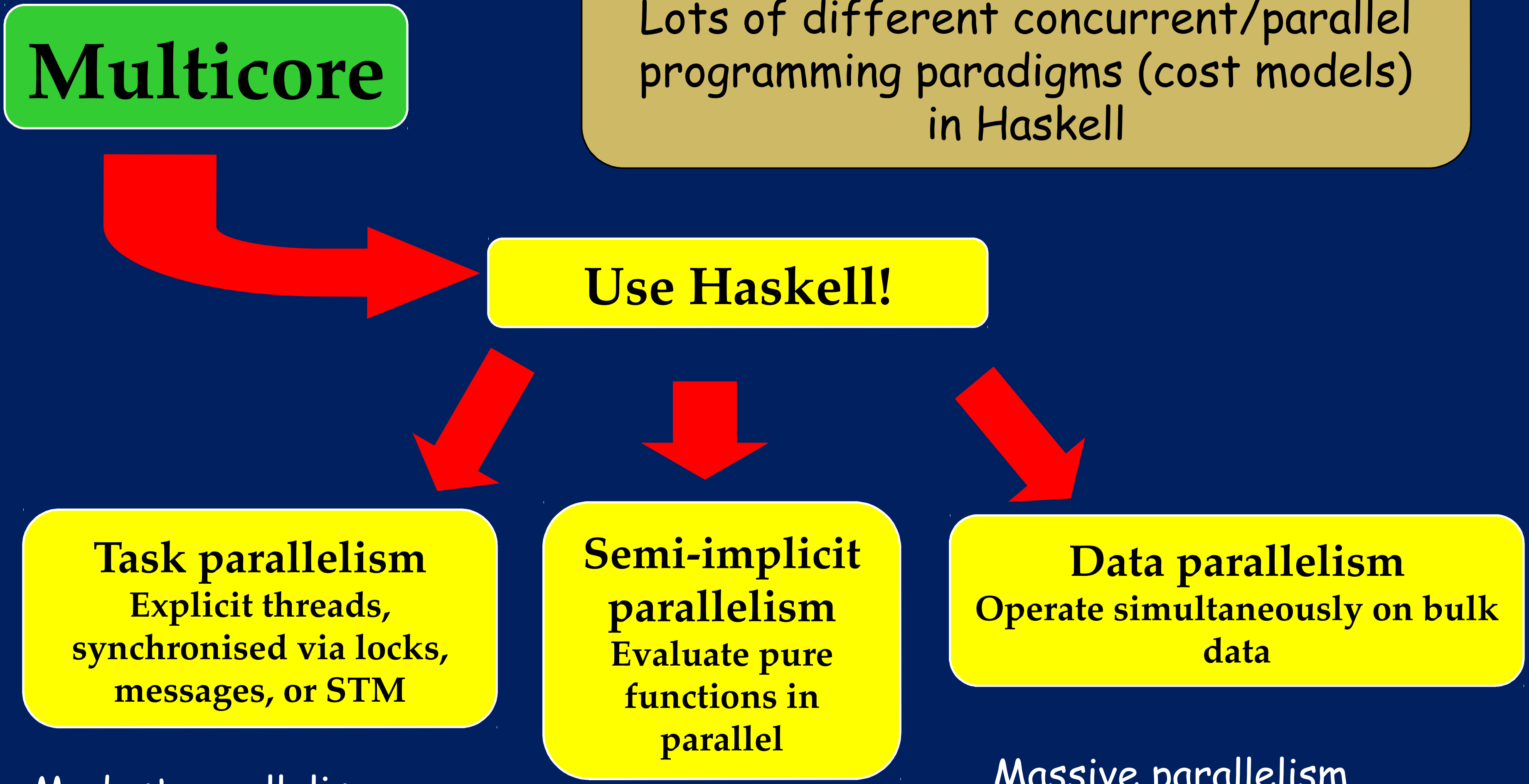
- No notion of sequence
- "A2" is the name of a (single) value

Functional programming to the rescue?

- "Just use a functional language and your troubles are over"
- Right idea:
  - ~~No side effects~~ Limited side effects
  - Strong guarantees that sub-computations do not interfere
- But far too starry eyed. No silver bullet:
  - Need to "think parallel": if the algorithm has sequential data dependencies, no language will save you!
  - Parallelism is complicated: different applications need different approaches.

Haskell

- The only programming language that takes purity really seriously
- 21 years old this year... yet still in a ferment of development
- Particularly good for Domain Specific Embedded Languages (aka libraries that feel easy to use).
- Offers many different approaches to parallel/concurrent programming, each with a different cost model.
  - No up-front choice
  - You can use several paradigms in one program



Multicore

This talk
Lots of different concurrent/parallel programming paradigms (cost models) in Haskell

Use Haskell!

Task parallelism
Explicit threads, synchronised via locks, messages, or STM

Modest parallelism
Hard to program

Semi-implicit parallelism
Evaluate pure functions in parallel

Modest parallelism
Implicit synchronisation
Easy to program

Data parallelism
Operate simultaneously on bulk data

Massive parallelism
Easy to program
Single flow of control
Implicit synchronisation

Slogan: no silver bullet: embrace diversity

No Silver Bullet

Many different parallelism paradigms

One language

One program uses multiple paradigms

Road map

Multicore

Use Haskell!

Semi-implicit parallelism

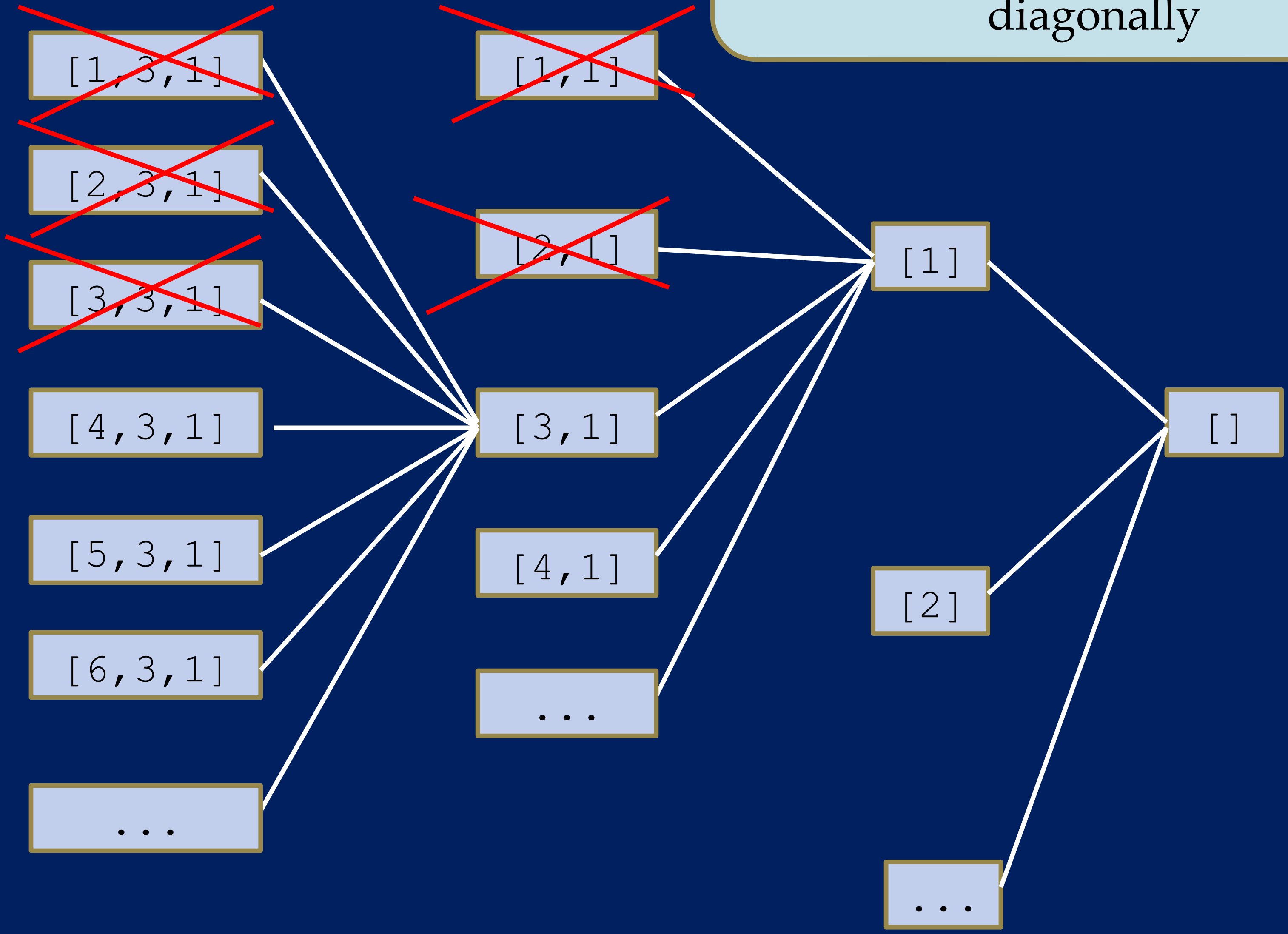
Evaluate pure functions in parallel

Modest parallelism
Implicit synchronisation
Easy to program

Slogan: no silver bullet: embrace diversity

N queens

Place *n* queens on an *n* x *n* board such that no queen attacks any other, horizontally, vertically, or diagonally



~~[1,3,1]~~

~~[2,3,1]~~

~~[3,3,1]~~

[4,3,1]

[5,3,1]

[6,3,1]

...

~~[1,1]~~

~~[2,1]~~

[3,1]

[4,1]

...

[1]

[2]

...

[]

Start here

NQueens

Place *n* queens on an *n* x *n* board such that no queen attacks any other, horizontally, vertically, or diagonally

- Sequential code

```
nqueens :: Int -> [[Int]]
nqueens n = subtree n []

subtree :: Int -> [Int] -> [[Int]]
subtree 0 b = [b]
subtree c b = concat $
              map (subtree (c-1)) (children b)

children :: [Int] -> [[Int]]
children b = [ (q:b) | q <- [1..n],
                       safe q b ]
```

NQueens

Place *n* queens on an *n* x *n* board such that no queen attacks any other, horizontally, vertically, or diagonally

- Parallel code

```
nqueens :: Int -> [[Int]]
nqueens n = subtree n []

subtree :: Int -> [Int] -> [[Int]]
subtree 0 b = [b]
subtree c b = concat $
              parMap (subtree (c-1)) (children b)

children :: [Int] -> [[Int]]
children b = [ (q:b) | q <- [1..n],
                       safe q b ]
```

Works on the sub-trees in parallel

- Speedup: 3.5x on 6 cores

Semi-implicit parallelism

```
map    :: (a->b) -> [a] -> [b]
parMap :: (a->b) -> [a] -> [b]
```

Good things

- Parallel program guaranteed not to change the result
- Deterministic: same result every run
- Very low barrier to entry
- "Strategies" to separate algorithm from parallel structure
- Implementation free to map available parallelism to actual architecture

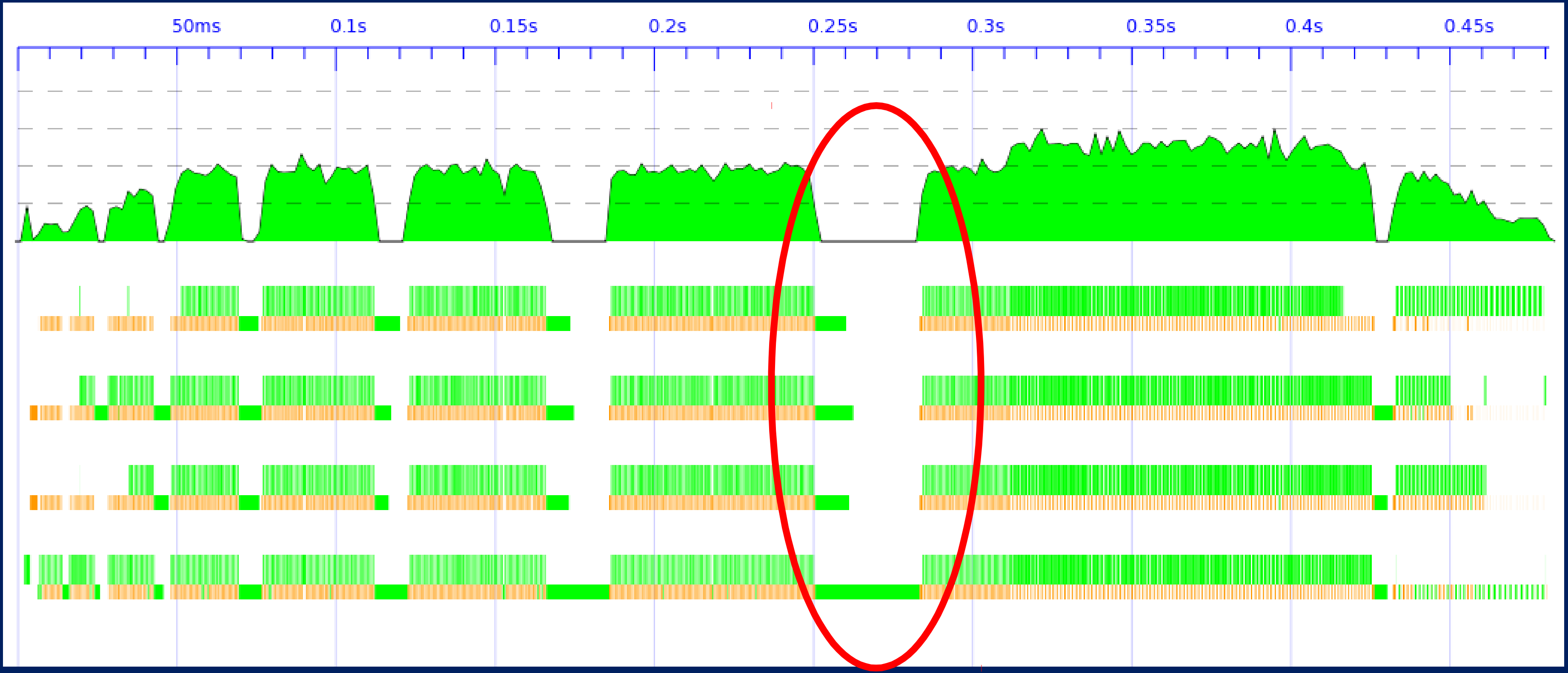
Semi-implicit parallelism

Bad things

- Poor cost model; all too easy to fail to evaluate something and lose all parallelism
- Not much locality; shared memory
- Over-fine granularity can be a big issue

Profiling tools can help a lot

ThreadScope



- As usual, watch out for Amdahl's law!

Cryptographic Protocol Shapes Analyzer (CPSA)

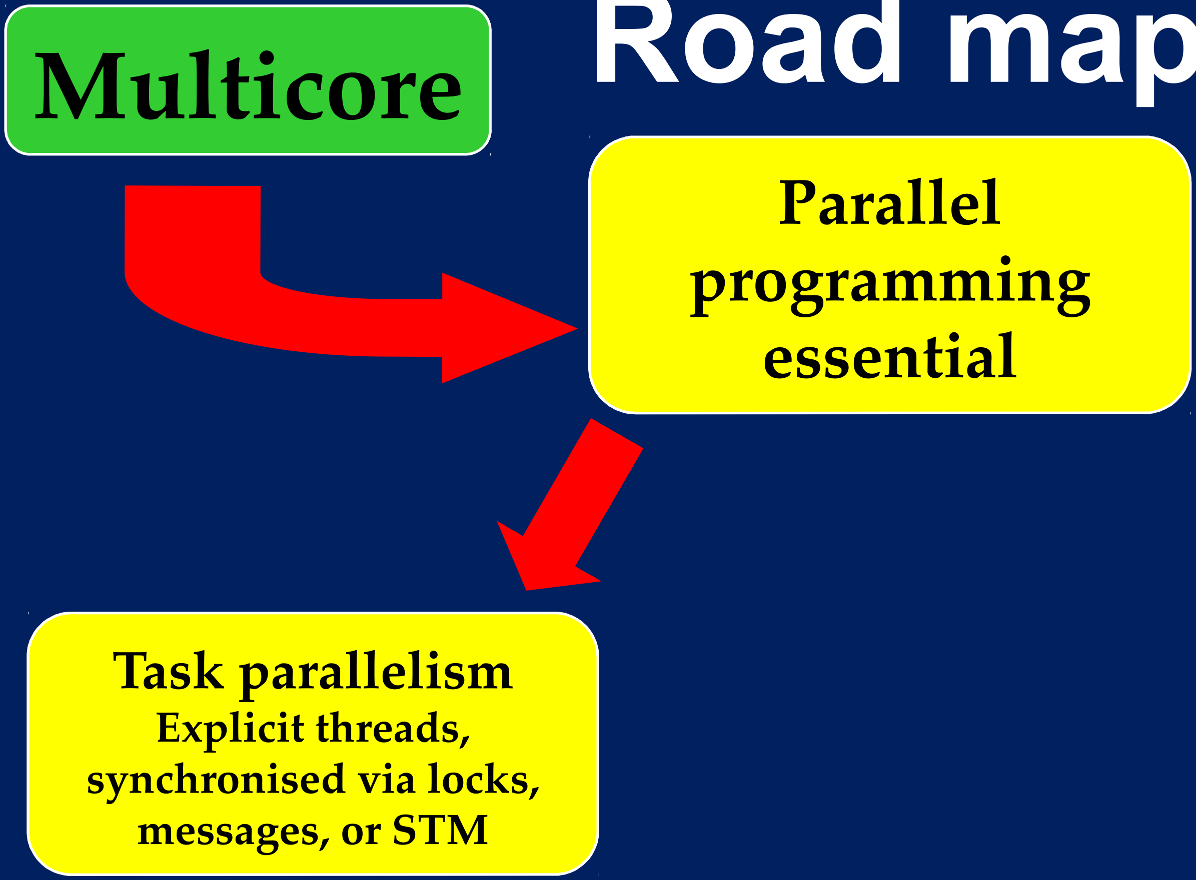
http://hackage.haskell.org/package/cpsa

- Find authentication or secrecy failures in cryptographic protocols. (Famous example: authentication failure in the Needham-Schroeder public key protocol.)
- About 6,500 lines of Haskell
- "I think it would be moronic to code CPSA in C or Python. The algorithm is very complicated, and the leap between the documented design and the Haskell code is about as small as one can get, because the design is functional."
- **One call to parMap**
- Speedup of 3x on a quad-core --- worthwhile when many problems take 24 hrs to run.

Summary of semi-implicit

- Modest but worthwhile speedups (3-10) for very modest investment
- Limited to shared memory; 10's not 1000's of processors
- You still have to think about a parallel algorithm! (Eg John Ramsdell had to refactor his CPSA algorithm a bit.)

Road map



Multicore

Parallel programming essential

Task parallelism
Explicit threads, synchronised via locks, messages, or STM

Expressing concurrency

- Lots of threads, all performing I/O
  - GUIs
  - Web servers (and other servers of course)
  - BitTorrent clients
- Non-deterministic by design
- Needs
  - Lightweight threads
  - A mechanism for threads to coordinate/share
  - Typically: pthreads/Java threads + locks/condition variables

What you get in Haskell

- Very very lightweight threads
  - Explicitly spawned, can perform I/O
  - Threads cost a few hundred bytes each
  - You can have (literally) millions of them
  - I/O blocking via epoll => OK to have hundreds of thousands of outstanding I/O requests
  - Pre-emptively scheduled
- Threads share memory
- Coordination via Software Transactional Memory (STM)

I/O in Haskell

```
main = do { putStr (reverse "yes")
          ; putStr "no" }
```

- Effects are explicit in the type system
  - `(reverse "yes") :: String` -- No effects
  - `(putStr "no") :: IO ()` -- Can have effects
- The main program is an effect-ful computation
  - `main :: IO ()`

Mutable state

```
newRef :: a -> IO (Ref a)
readRef :: Ref a -> IO a
writeRef :: Ref a -> a -> IO ()
```

```
main = do { r <- newRef 0
          ; incR r
          ; s <- readRef r
          ; print s }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v+1)
            }
```

Reads and writes are 100% explicit!

You can't say (r + 6), because r :: Ref Int

Concurrency in Haskell

```
forkIO :: IO () -> IO ThreadId
```

- forkIO spawns a thread
- It takes an action as its argument

```
webServer :: RequestPort -> IO ()
webServer p = do { conn <- acceptRequest p
                 ; forkIO (serviceRequest conn)
                 ; webServer p }

serviceRequest :: Connection -> IO ()
serviceRequest c = do { … interact with client … }
```

No event-loop spaghetti!

Coordination in Haskell

- How do threads coordinate with each other?

```
main = do { r <- newRef 0
          ; forkIO (incR r)
          ; incR r
          ; ... }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v+1) }
```

Aargh!
A race

What’s wrong with locks?

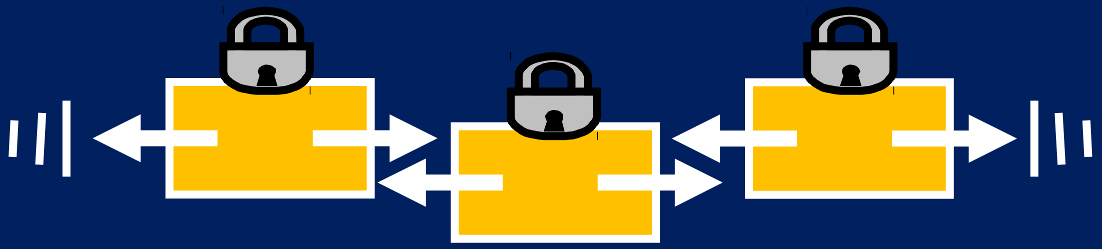
A 10-second review:

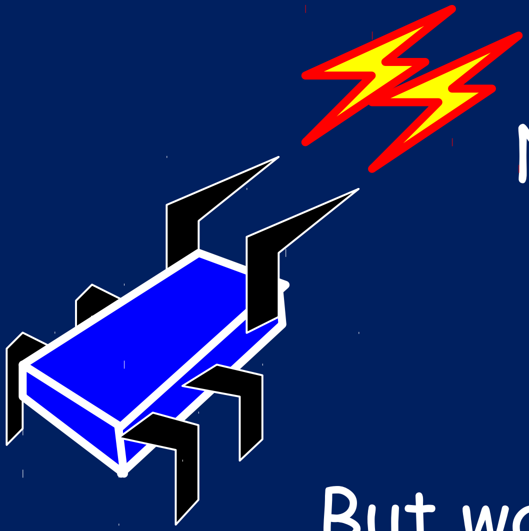
- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in “wrong” order.
- **Lost wakeups:** forgotten notify to condition variable
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...

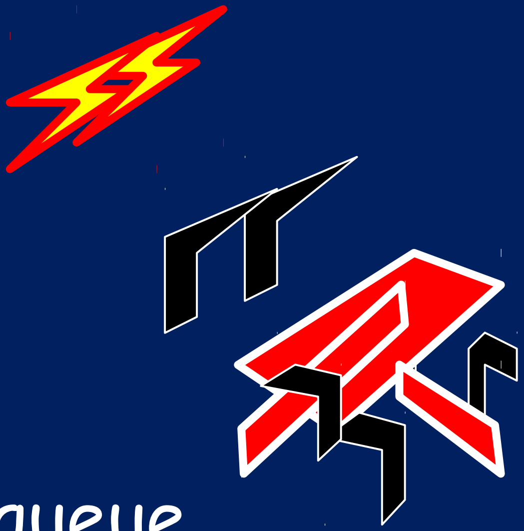
Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell





No interference if ends "far enough" apart



But watch out when the queue is 0, 1, or 2 elements long!

Locks are absurdly hard to get right

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate

Locks are absurdly hard to get right

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference

Atomic memory transactions

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference
Atomic blocks	**Undergraduate**

Atomic memory transactions

atomically { ... sequential get code ... }

- To a first approximation, just write the sequential code, and wrap **atomically** around it
- All-or-nothing semantics: **Atomic** commit
- Atomic block executes in **Isolation**
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the **get** code)

AcId

Atomic blocks in Haskell

```
atomically :: IO a -> IO a
```

```
main = do { r <- newRef 0
          ; forkIO (atomically (incR r))
          ; atomically (incR r)
          ; ... }
```

- atomically is a function, not a syntactic construct
- A worry: what stops you doing incR outside atomically?

STM in Haskell

- Better idea:

```
atomically :: STM a -> IO a
newTVar    :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar      :: TVar a -> a -> STM
()
```

```
incT :: TVar Int -> STM ()
incT r = do { v <- readTVar r; writeTVar r (v+1) }
main = do { r <- atomically (newTVar 0)
            ; forkIO (atomically (incT r))
            ; atomic (incT r)
            ; ... }
```

STM in Haskell

```
atomic :: STM a -> IO a
newTVar :: a -> STM (TVar a)
readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
```

- Can't fiddle with TVars outside atomic block [good]
- Can't do IO inside atomic block [sad, but also good]
- No changes to the compiler (whatsoever). Only runtime system and primops.

Lots more…

http://research.microsoft.com/~simonpj/papers/stm

- STM composes beautifully
- MVars for efficiency in (very common) special cases
- Blocking (retry) and choice (orElse) in STM
- Exceptions in STM

Example: Warp

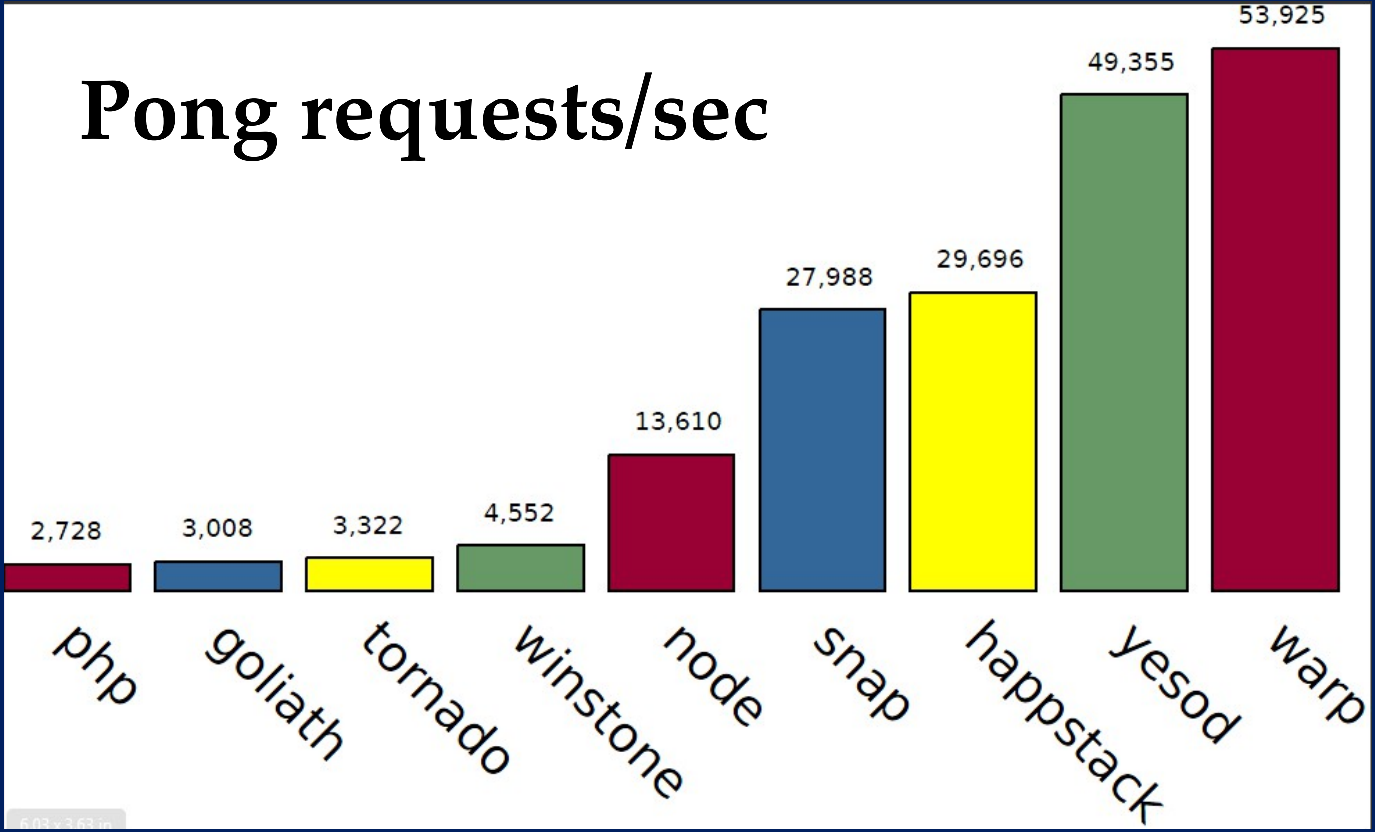
http://docs.yesodweb.com/blog/announcing-warp

- A very simple web server written in Haskell
  - full HTTP 1.0 and 1.1 support,
  - handles chunked transfer encoding,
  - uses sendfile for optimized static file serving,
  - allows request bodies and response bodies to be processed in constant space
- Protection for all the basic attack vectors: overlarge request headers and slow-loris attacks
- 500 lines of Haskell (building on some amazing libraries: bytestring, blaze-builder, iteratee)

Example: Warp

http://docs.yesodweb.com/blog/announcing-warp

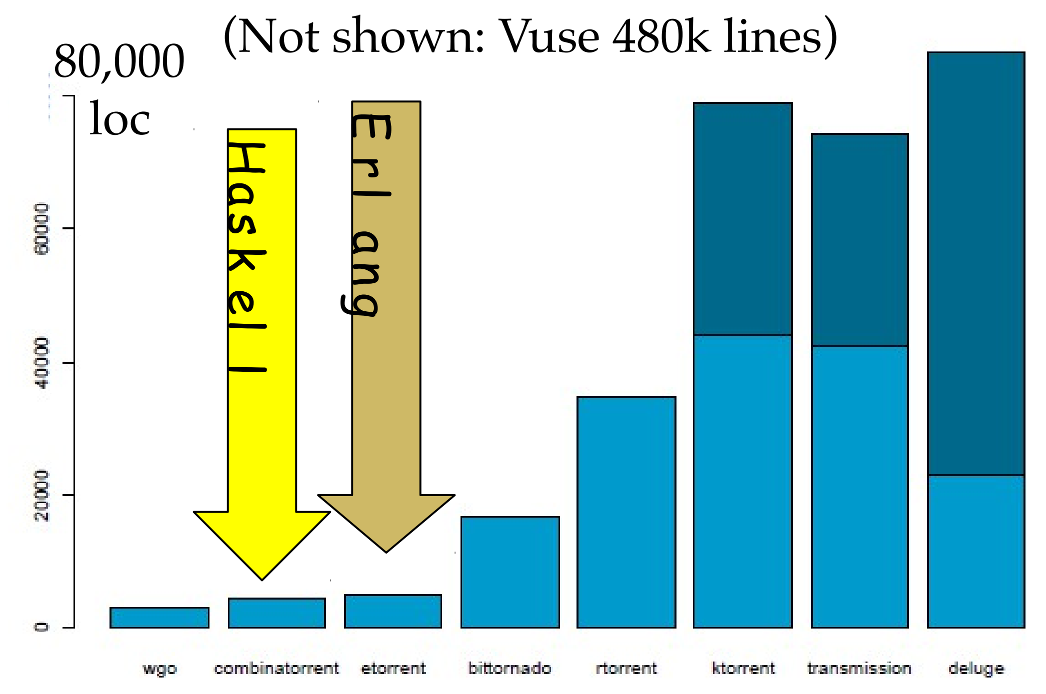
- A new thread for each user request
- Fast, fast



Example: Combinatorrent

http://jlouis.github.com/combinatorrent/

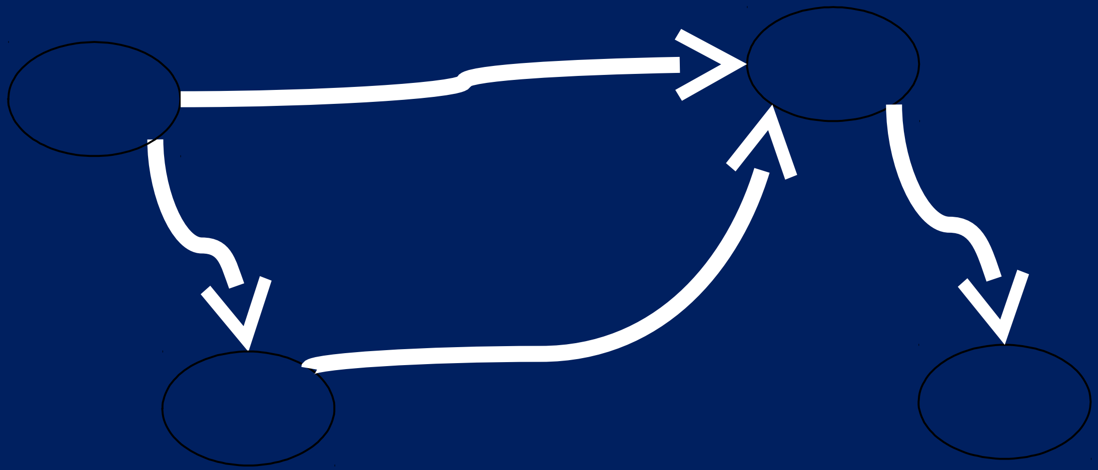
- Again, lots of threads: 400-600 is typical
- Significantly bigger program: 5000 lines of Haskell – but way smaller than the competition
- Built on STM
- Performance: roughly competitive



(Not shown: Vuse 480k lines)

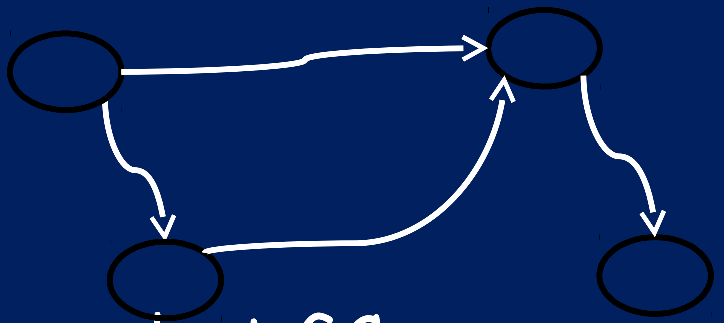
Distributed memory

- So far everything is shared memory
- Distributed memory has a different cost model



- Think message passing…
- Think Erlang…

Erlang

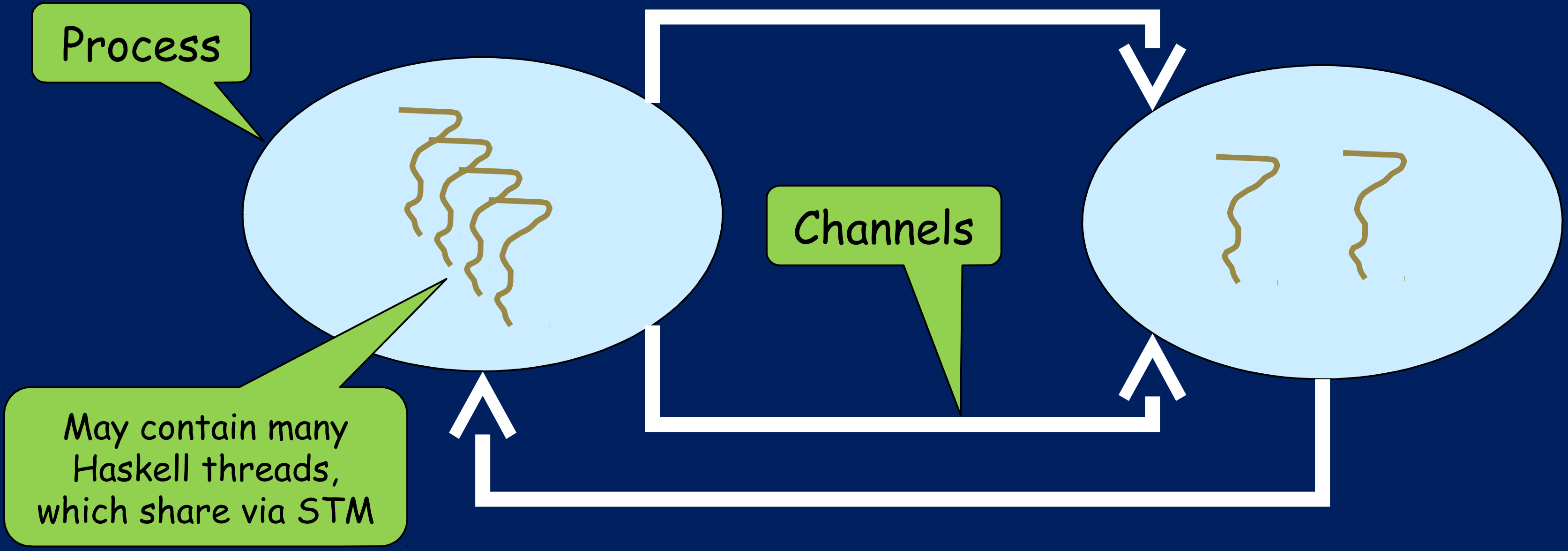


- Processes share nothing; independent GC; independent failure
- Communicate over channels
- Message communication = serialise to bytestream, transmit, deserialise
- Comprehensive failure model
  - A process P can "link to" another Q
  - If Q crashes, P gets a message
  - Use this to build process monitoring apparatus
  - Key to Erlang's 5-9's reliability

Cloud Haskell

- Provide Erlang as a library – no language extensions needed

```
newChan :: PM (SPort a, RPort a)
send     :: Serialisable a => SPort a -> a -> PM a
receive  :: Serialisable a => RPort a -> PM a

spawn :: NodeId -> PM a -> PM PId
```



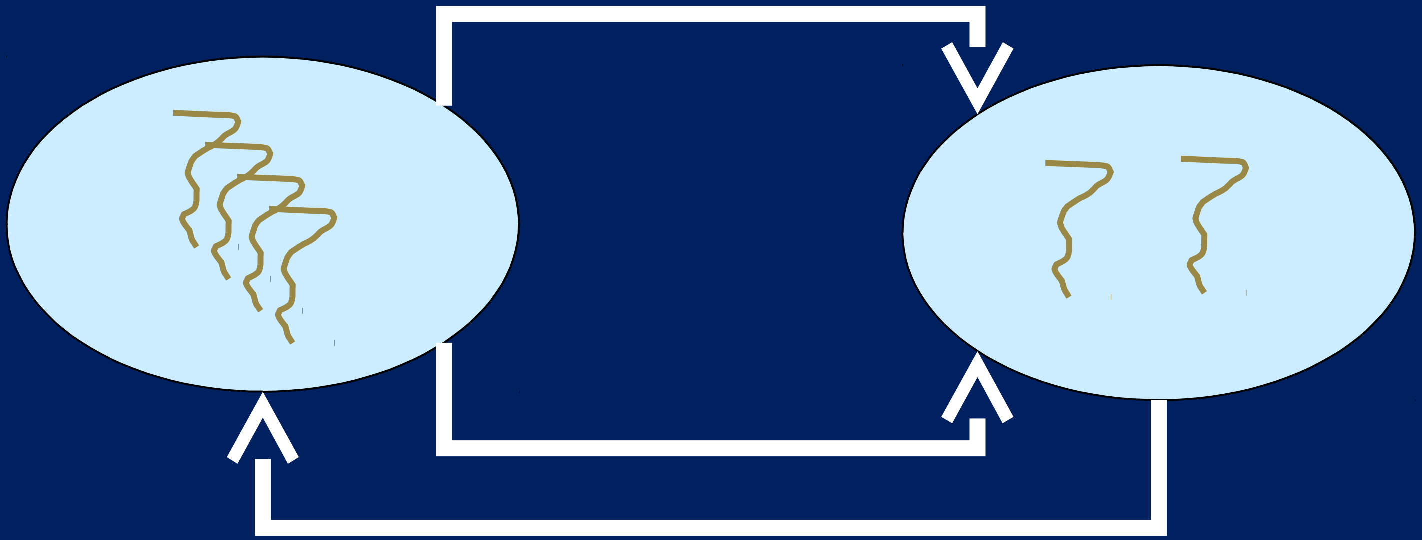
Process

Channels

May contain many
Haskell threads,
which share via STM

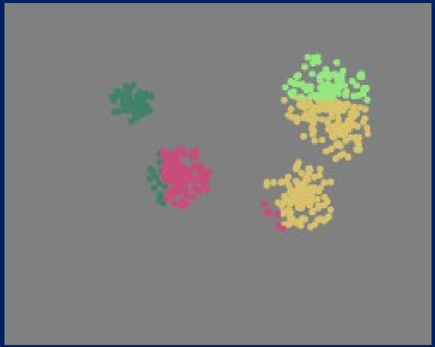
Cloud Haskell

- Many static guarantees for cost model:
  - (SPort a) is serialisable, but not (RPort a)
    => you always know where to send your message
  - (TVar a) not serialisable
    => no danger of multi-site STM

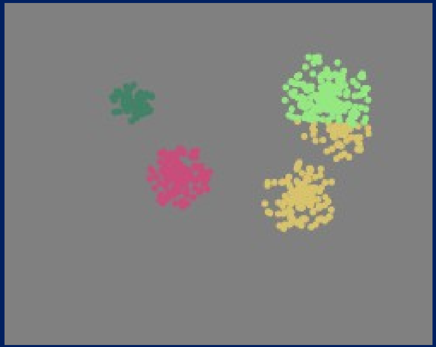


K-means clustering

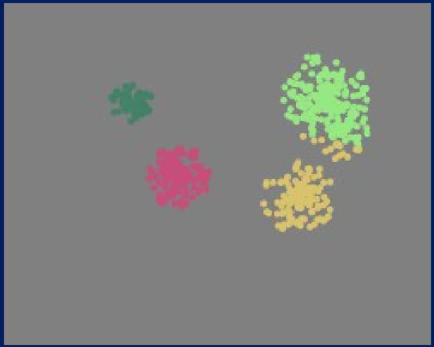
The k-means clustering algorithm takes a set of data points and groups them into clusters by spatial proximity.



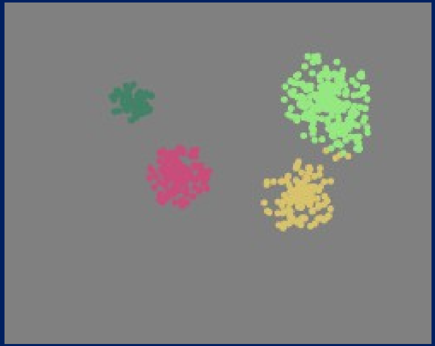
Initial clusters have random centroids



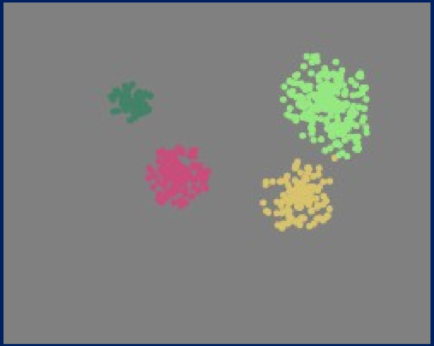
After first iteration



After second iteration



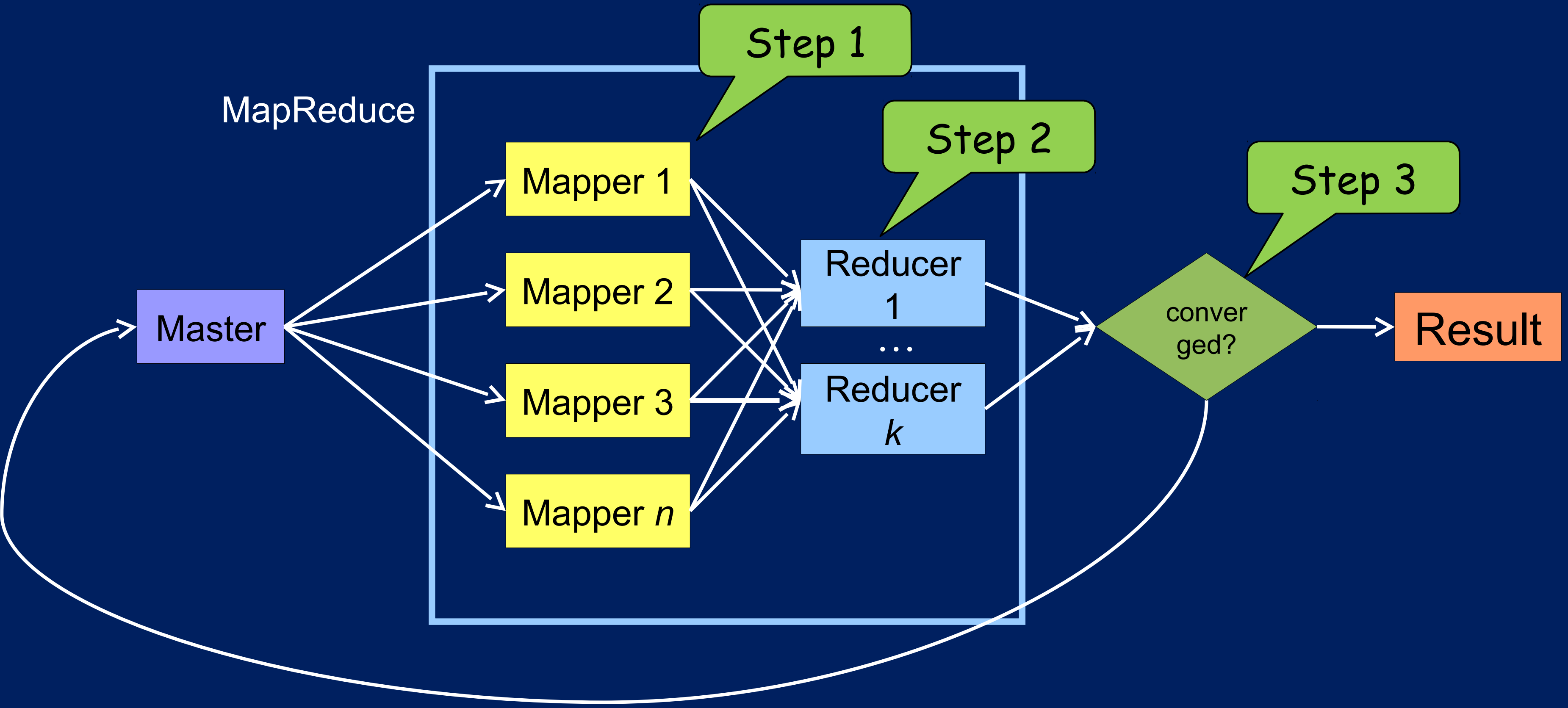
After third iteration



Converged

- Start with Z lots of data points in N-dimensional space
- Randomly choose k points as ”centroid candidates”
- Repeat:
    1. For each data point, find the nearerst ”centroid candidate”
    2. For each candidate C, find the centroid of all points nearest to C
    3. Make those the new centroid candidates, and repeat

- Start with Z lots of data points in N-dimensional space
- Randomly choose k points as ”centroid candidates”
- Repeat:
    1. For each data point, find the nearerst ”centroid candidate”
    2. For each candidate C, find the centroid of all points nearest to C
    3. Make those the new centroid candidates, and repeat if necessary



Running today in Haskell on an Amazon EC2 cluster [current work]

Summary so far

Highly concurrent applications are a killer app for Haskell

Summary so far

Highly concurrent applications are a killer app for Haskell

But wait… didn't you say that Haskell was a **functional** language?

Value oriented programming => better concurrent programs

- Side effects are inconvenient

```
do { v <- readTVar r; writeTVar r (v+1) }
```

  vs

```
r++
```

- Result: almost all the code is functional, processing immutable data
- **Great for avoiding bugs**: no aliasing, no race hazards, no cache ping-ponging.
- **Great for efficiency**: only TVar access are tracked by STM

Multicore

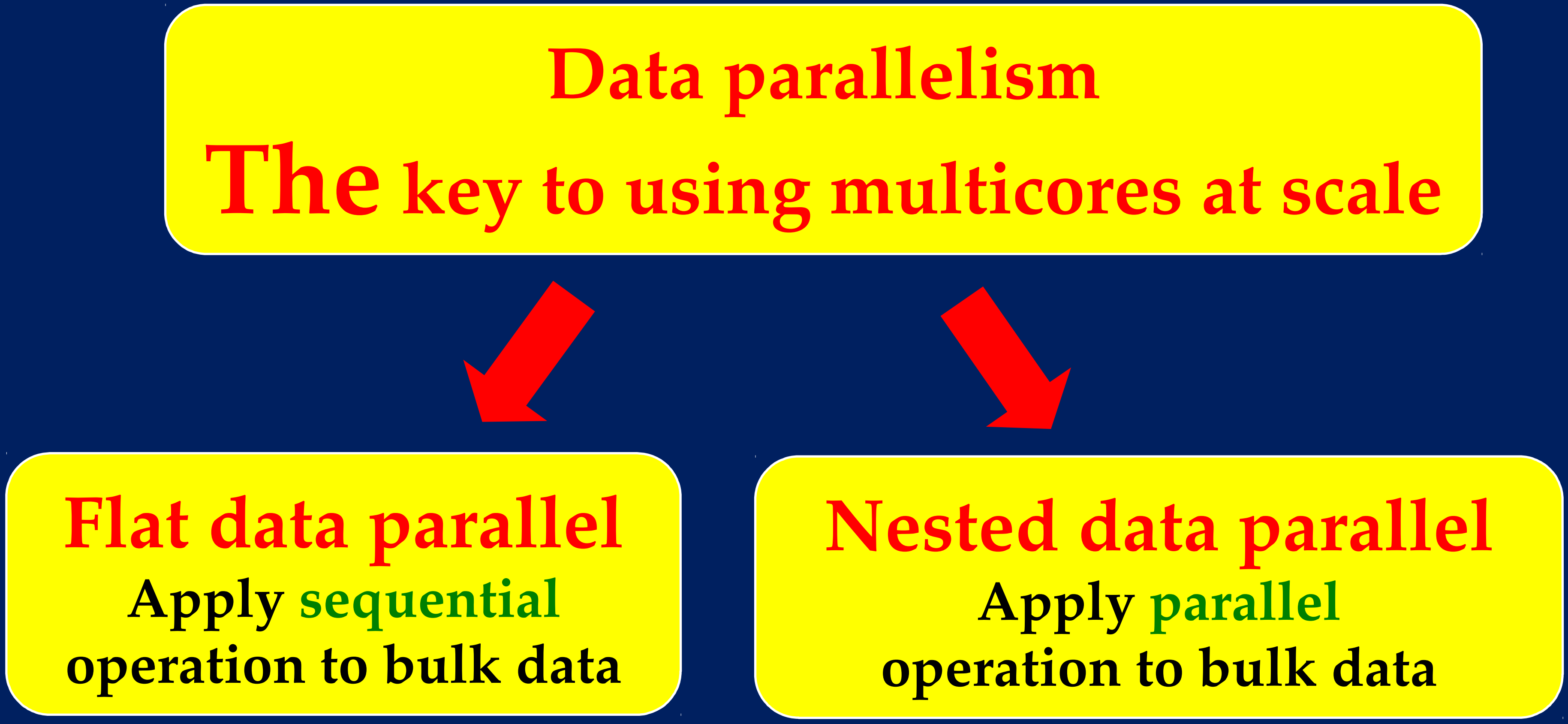
Road map

Use Haskell!

Data parallelism
Operate simultaneously on bulk data

Massive parallelism
Easy to program
Single flow of control
Implicit synchronisation

Slogan: no silver bullet: embrace diversity



Data parallelism
The key to using multicores at scale

Flat data parallel
Apply sequential
operation to bulk data

Nested data parallel
Apply parallel
operation to bulk data

Very widely used

Research project

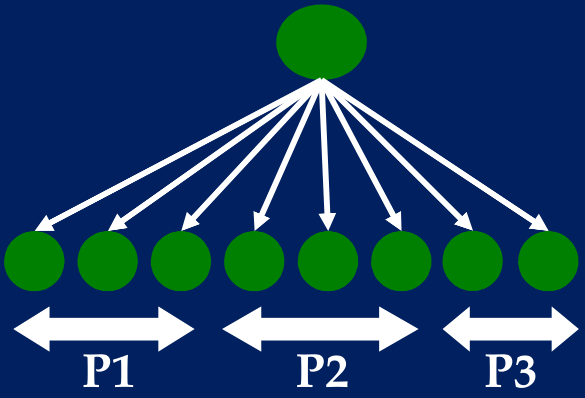
Flat data parallel

e.g. Fortran(s), *C
MPI, map/reduce

- The brand leader: widely used, well understood, well supported

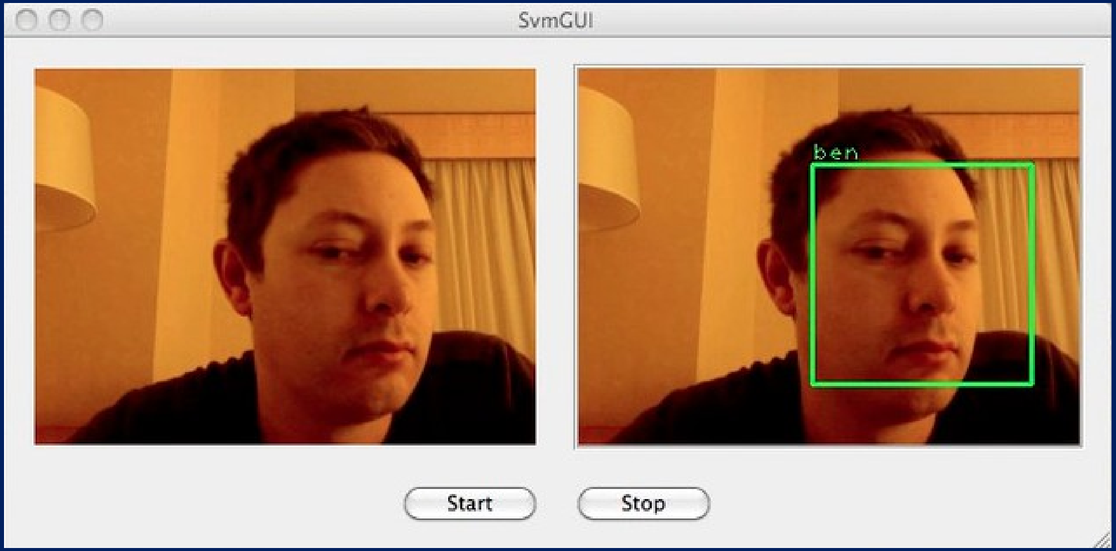
```
foreach i in 1..N {
    ...do something to A[i]...
}
```

- BUT: "**something**" *is sequential*
- Single point of concurrency
- Easy to implement: use "chunking"
- Good cost model (both granularity and locality)



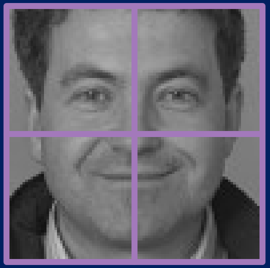
1,000,000's of (small) work items

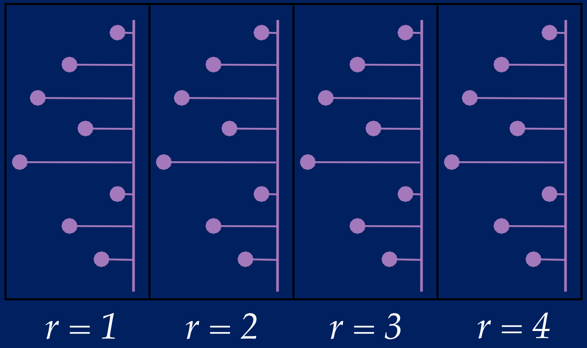
Face Recognition (NICTA, Sydney)



Faces are compared by computing a *distance* between their *multi-region histograms*.

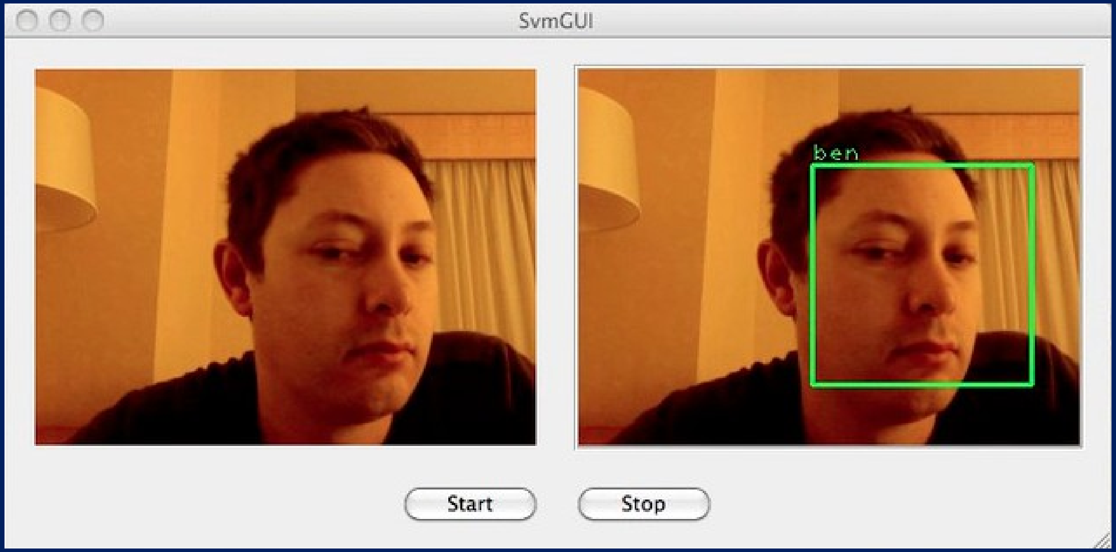
$$\mathrm{dist}(A,B)=\frac{1}{R}\sum_{r=1}^{R}\left\|\vec{h}_r^{[A]}-\vec{h}_r^{[B]}\right\|_1$$

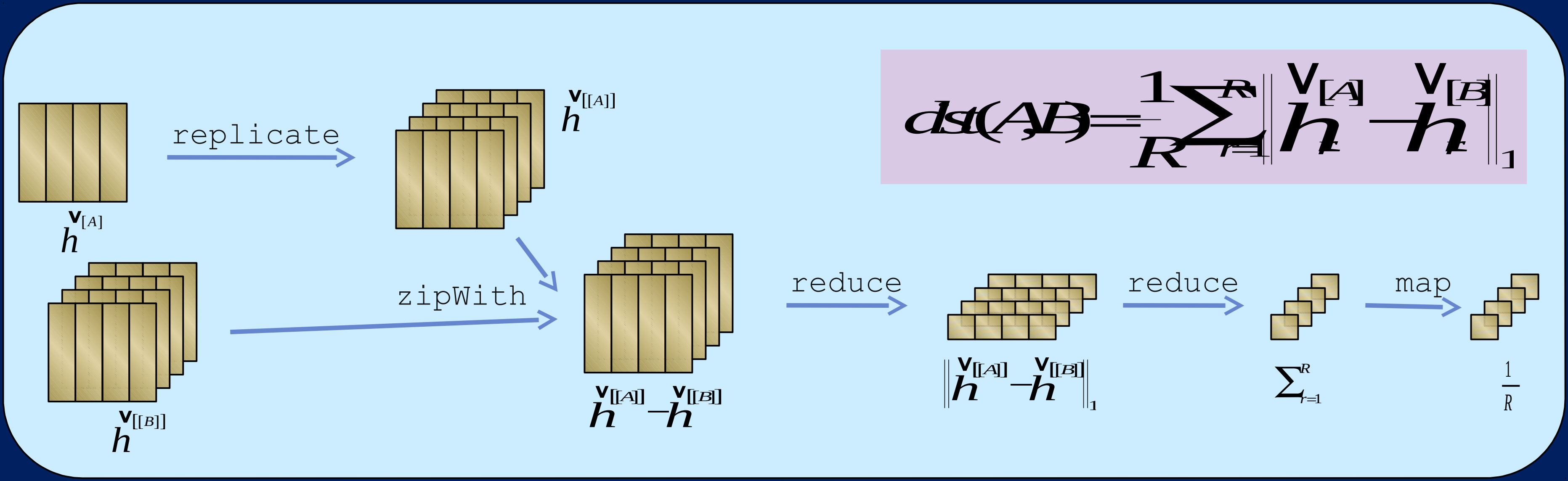




Multi-region histogram for **candidate face** as an array.

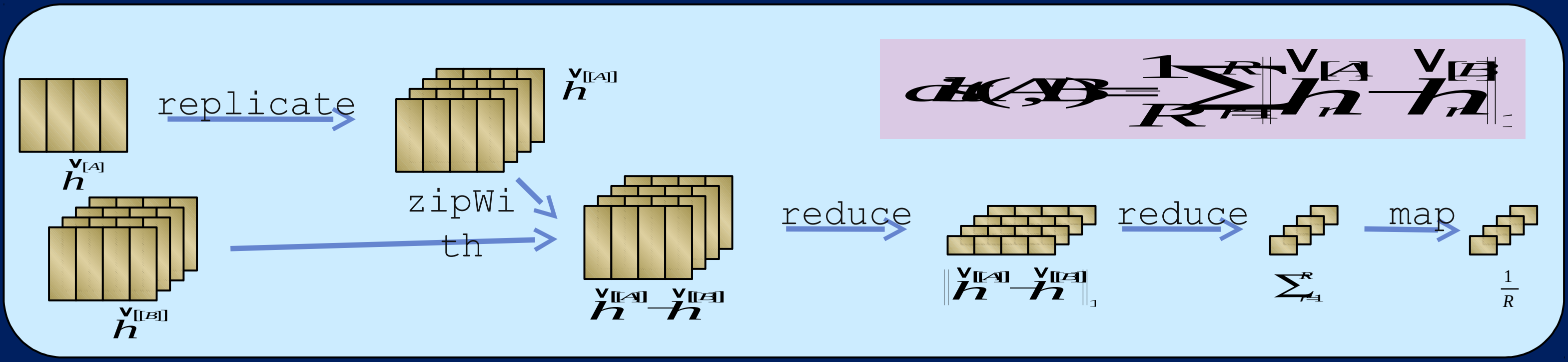
Face Recognition: Distance calculation





$$dist(A,B) = \frac{1}{R}\sum_{r=1}^{R}\left\| \vec{h}_r^{[A]} - \vec{h}_r^{[B]} \right\|_1$$

Face Recognition: Distance calculation

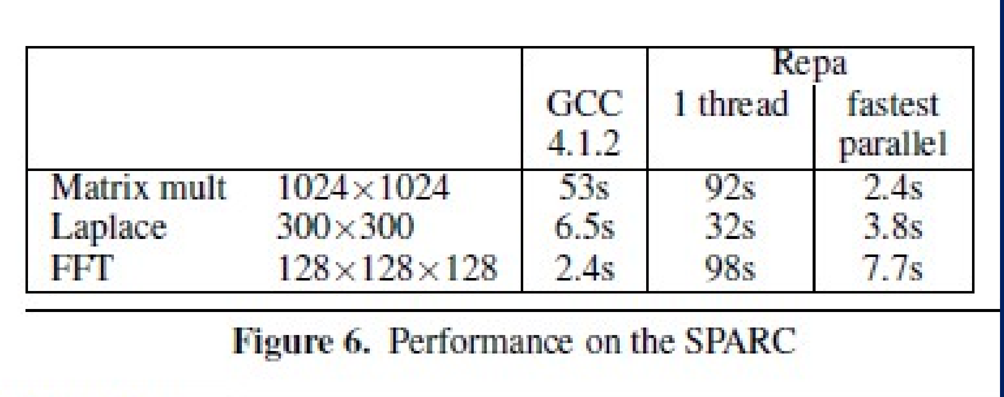


```
distances :: Array DIM2 Float -> Array DIM3 Float
          -> Array DIM1 Float
distances histA histBs = dists
  where
    histAs = replicate (constant (All, All, f)) histA

    diffs  = zipWith (-) histAs histBs
    l1norm = reduce (\a b -> abs a + abs b) (0) diffs
    regSum = reduce (+) (0) l1norm
    dists  = map (/ r) regSum

     (h, r, f) = shape histBs
```

Repa: regular, shape-polymorphic parallel arrays in Haskell

http://justtesting.org/regular-shape-polymorphic-parallel-arrays-in

- Arrays as values: virtually no element-wise programming (for loops).
- Think APL, but with much more polymorphism
- Performance is (often) comparable to C
- AND it auto-parallelises



		GCC 4.1.2	Repa 1 thread	 fastest parallel
Matrix mult	1024×1024	53s	92s	2.4s
Laplace	300×300	6.5s	32s	3.8s
FFT	128×128×128	2.4s	98s	7.7s

Figure 6. Performance on the SPARC

Warning: take all such figures with buckets of salt

GPUs

http://www.cse.unsw.edu.au/~chak/project/accelerate/

- GPUs are massively parallel processors, and are rapidly de-specialising from graphics
- Idea: your program (when run) generates a GPU program

```
distances :: Acc (Array DIM2 Float)
          -> Acc (Array DIM3 Float)
          -> Acc (Array DIM1 Float)
distances histA histBs = dists
  where
    histAs = replicate (constant (All, All, f)) histA
    diffs  = zipWith (-) histAs histBs
    l1norm = reduce (\a b -> abs a + abs b) (0) diffs
    regSum = reduce (+) (0) l1norm
    dists  = map (/ r) regSum
```

GPUs

http://www.cse.unsw.edu.au/~chak/project/accelerate/

- An (Acc a) is a syntax tree for a program computing a value of type a, ready to be compiled for GPU
- The key trick: (+) :: **Num a =>** a -> a -> a

```
distances :: Acc (Array DIM2 Float)
          -> Acc (Array DIM3 Float)
          -> Acc (Array DIM1 Float)
distances histA histBs = dists
  where
    histAs = replicate (constant (All, All, f)) histA
    diffs  = zipWith (-) histAs histBs
    l1norm = reduce (\a b -> abs a + abs b) (0) diffs
    regSum = reduce (+) (0) l1norm
    dists  = map (/ r) regSum
```

GPUs

http://www.cse.unsw.edu.au/~chak/project/accelerate/

- An (Acc a) is a syntax tree for a program computing a value of type a, ready to be compiled for GPU

```
CUDA.run :: Acc (Array a b) -> Array a b
```

- CUDA.run
  - takes the syntax tree
  - compiles it to CUDA
  - loads the CUDA into GPU
  - marshals input arrays into GPU memory
  - runs it
  - marshals the result array back into Haskell memory

Main point

- The code for Repa (multicore) and Accelerate (GPU) is virtually identical
- Only the types change

- Other research projects with similar approach
  - Nicola (Harvard)
  - Obsidian/Feldspar (Chalmers)
  - Accelerator (Microsoft .NET)
  - Recursive islands (MSR/Columbia)

Data parallelism

The key to using multicores at scale

Nested data parallel

Apply parallel operation to bulk data

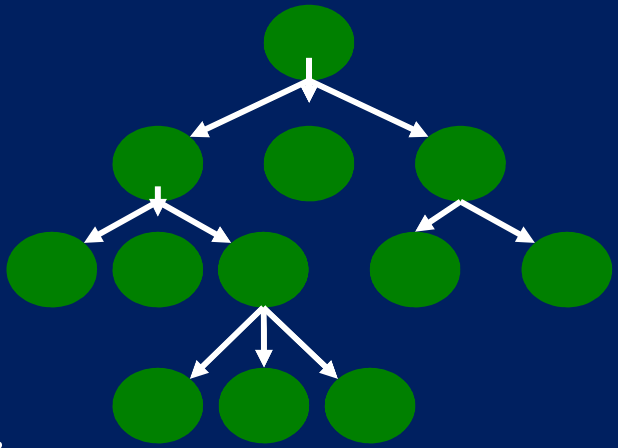
Research project

Nested data parallel

- Main idea: allow "something" to be parallel

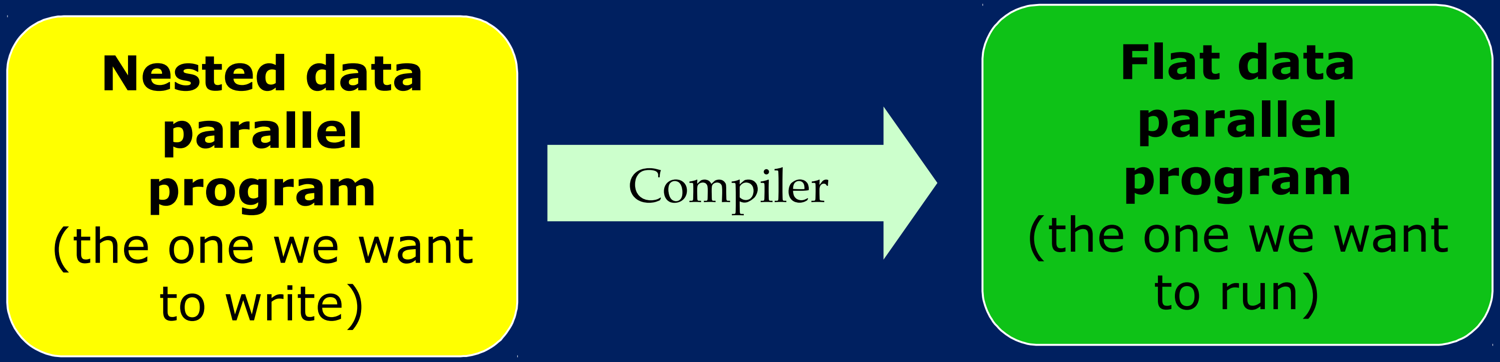
```
foreach i in 1..N {
   ...do something to A[i]...
}
```

- Now the parallelism structure is recursive, and un-balanced
- Much more expressive
- Much harder to implement



Still 1,000,000's of (small) work items

Amazing idea



Nested data parallel program
(the one we want to write)

Compiler

Flat data parallel program
(the one we want to run)

- Invented by Guy Blelloch in the 1990s
- We are now working on embodying it in GHC: Data Parallel Haskell
- Turns out to be jolly difficult in practice (but if it was easy it wouldn't be research). Watch this space.

Glorious Conclusion

- No single cost model suits all programs / computers. It's a complicated world. Get used to it.
- For concurrent programming, functional programming is already a huge win
- For parallel programming at scale, we're going to end up with data parallel functional programming
- Haskell is super-great because it hosts multiple paradigms. Many cool kids hacking in this space.
- But other functional programming languages are great too: Erlang, Scala, F#

Antithesis

Parallel functional programming was tried in the 80's, and basically failed to deliver

Then	Now
Uniprocessors were getting faster really, really quickly.	Uniprocessors are stalled
Our compilers were ~~crappy~~ naive, so constant factors were bad	Compilers are pretty good
The parallel guys were a dedicated band of super-talented programmers who would burn any number of cycles to make their supercomputer smoke.	They are regular Joe Developers
Parallel computers were really expensive, so you needed 95% utilisation	Everyone will has 8, 16, 32 cores, whether they use them or not. Even using 4 of them (with little effort) would be a Jolly Good Thing

Antithesis

Parallel functional programming was tried in the 80’s, and basically failed to deliver

Then

We had no story about
(a) locality,
(b) exploiting regularity, and
(c) granularity

Now

Lots of progress

- Software transactional memory
- Distributed memory
- Data parallelism
- Generating code for GPUs

This talk