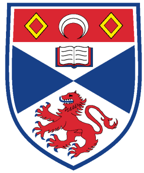


Patterns and Functional Programming

Kevin Hammond

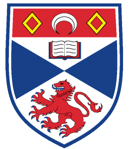
University of St Andrews, Scotland

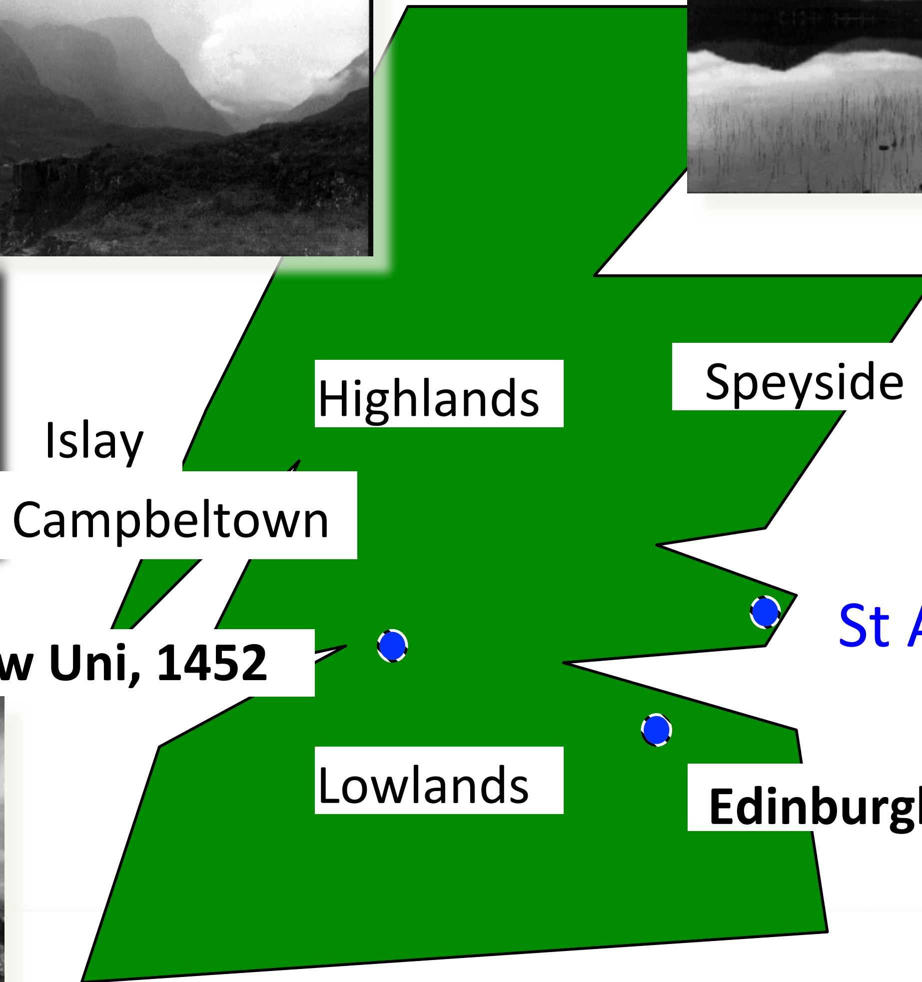
http://www.paraphrase-ict.eu

http://www.project-advance.eu

@paraphrase_fp7

Scotland





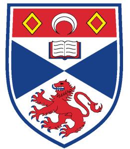
What is Multicore?

- More than one processor on a single chip
- Shared memory
- Common address space
- Shared data bus
- Independent instruction streams
- Shared cache

But haven't we had shared-memory before?

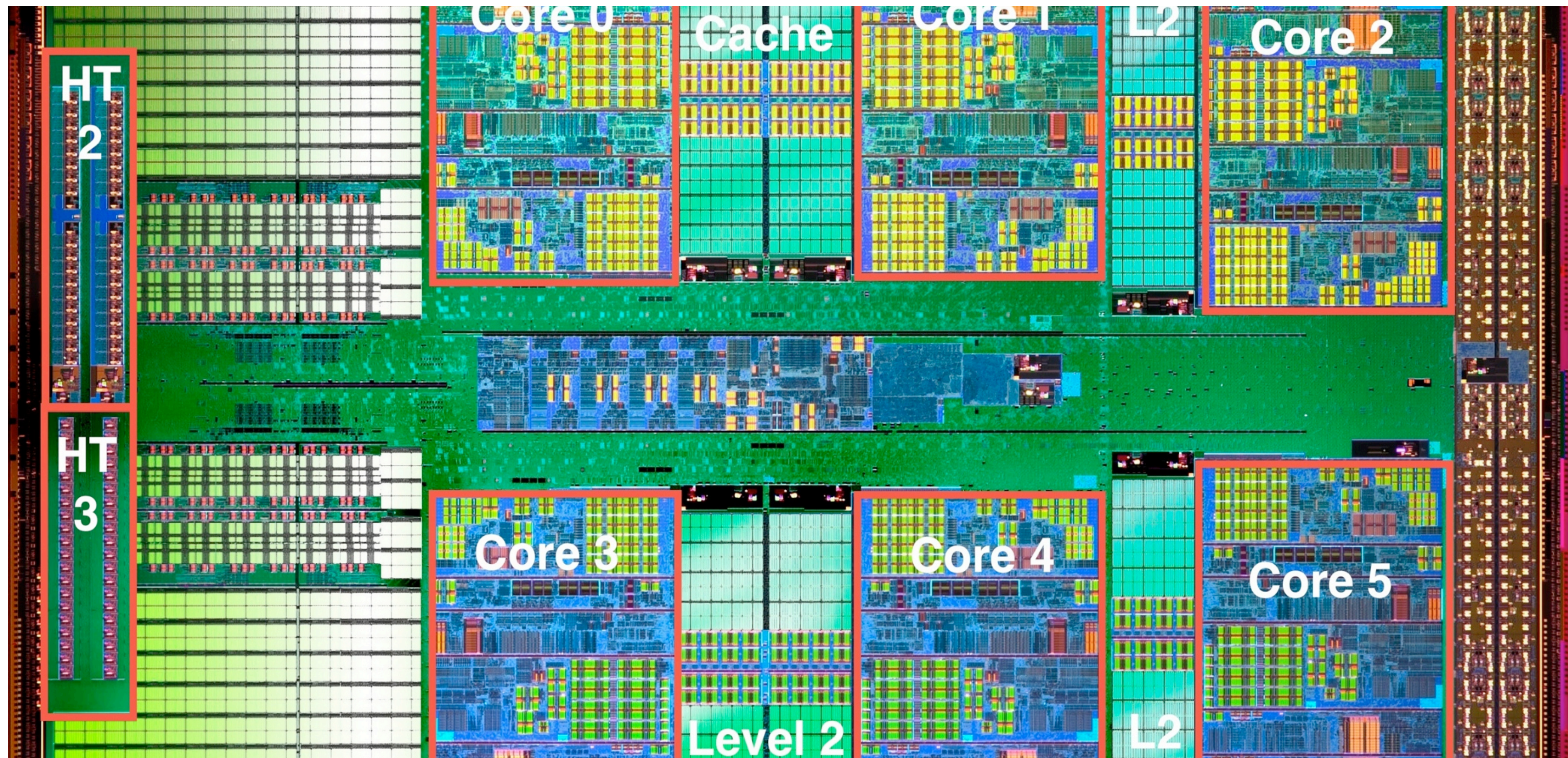
University of St Andrews

So why is multicore important??



The Dawn of a New Age

University of St Andrews

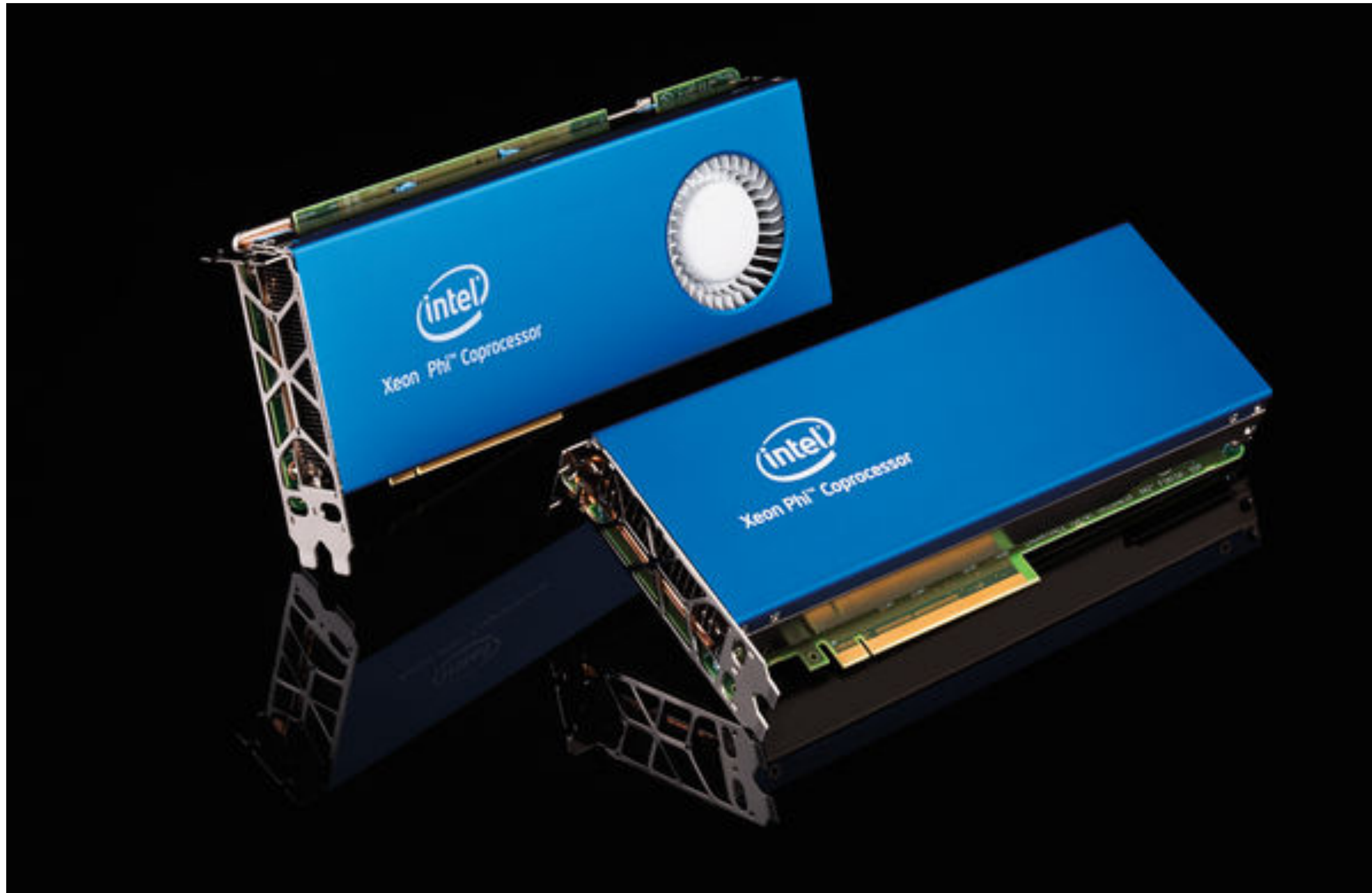


2013: a ManyCore Odyssey

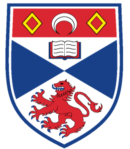
University of St Andrews

2013: a ManyCore Odyssey

University of St Andrews



The Future: “megacore” computers?



- *Hundreds of thousands, or millions, of cores*
- *Highly heterogeneous*
- *NOT uniform memory access*

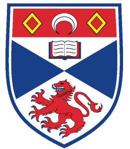
Core Core Core Core

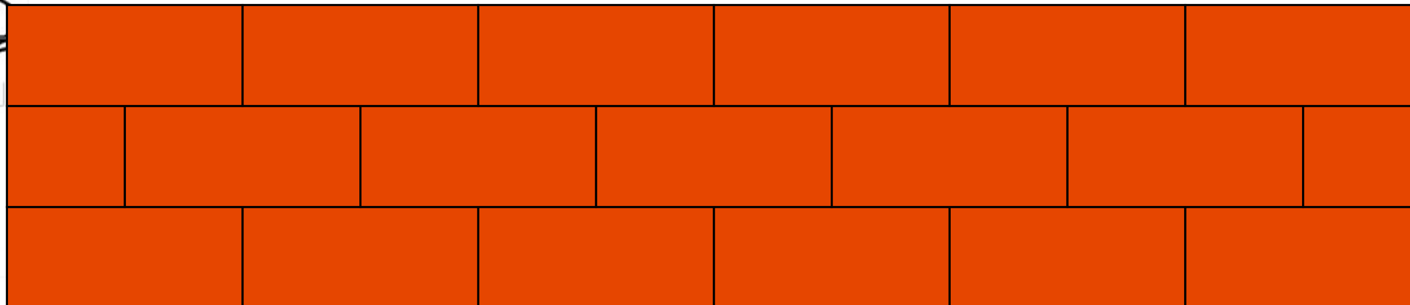
Programming Issues

University of St Andrews

- We can muddle through on 2-8 cores
  - maybe even 16 or so
  - modified sequential code may work
  - we may be able to use multiple programs to soak up cores
  - BUT larger systems are *much* more challenging
    - ***typical concurrency techniques will not scale (threads, locking, shared memory, explicit communication etc.)***

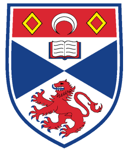
How to build a wall

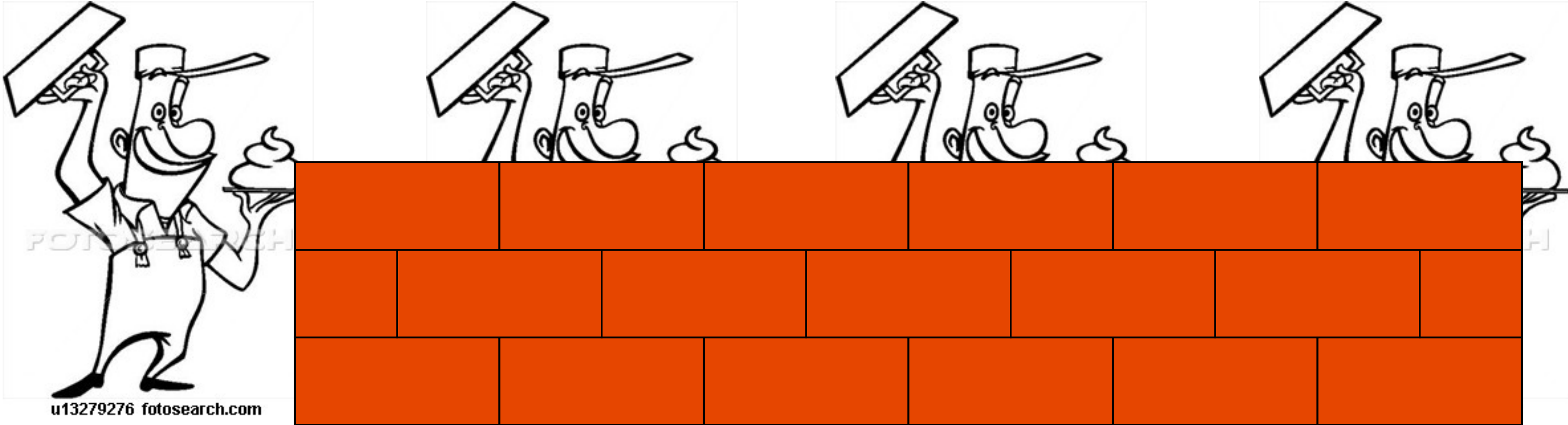




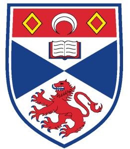
(with apologies to Ian Watson, Univ. Manchester)

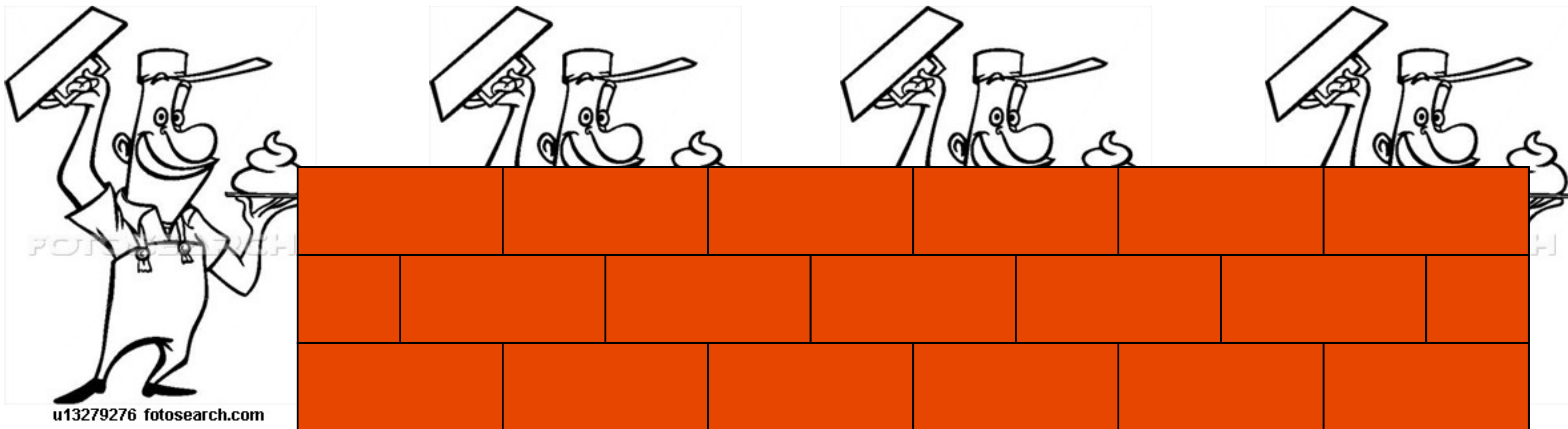
How to build a wall faster





How NOT to build a wall



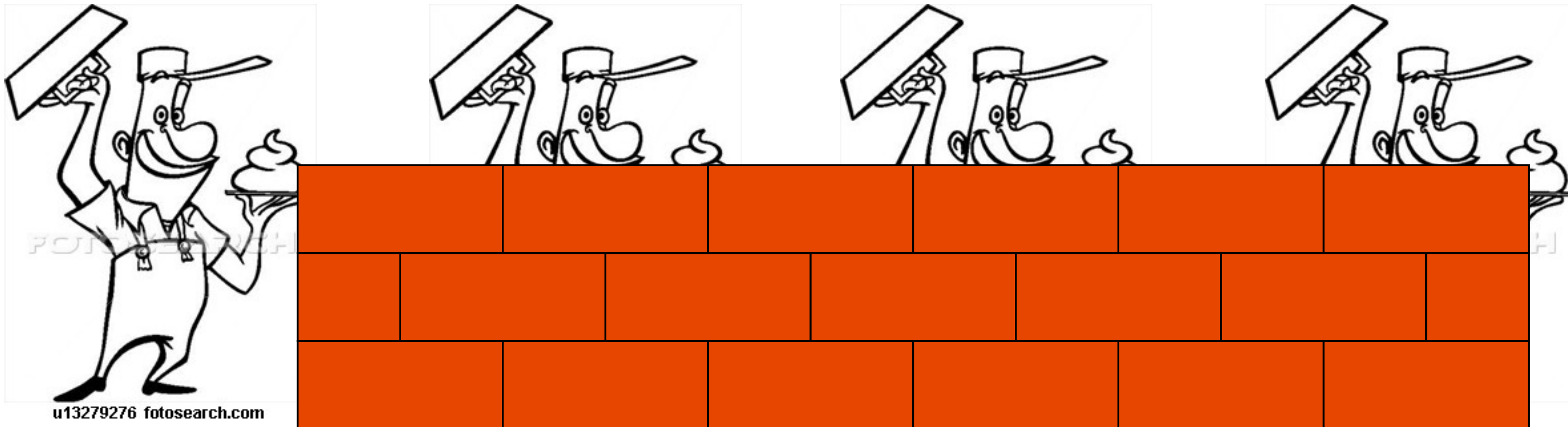


Task identification is not the only problem…

Must also consider Coordination, communication, placement, scheduling, …

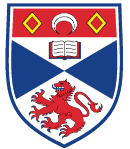
Scheduling

University of St Andrews



First row of bricks has to be built before the second
Second row of bricks has to be built before the third

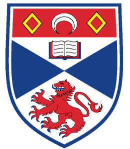
Placement

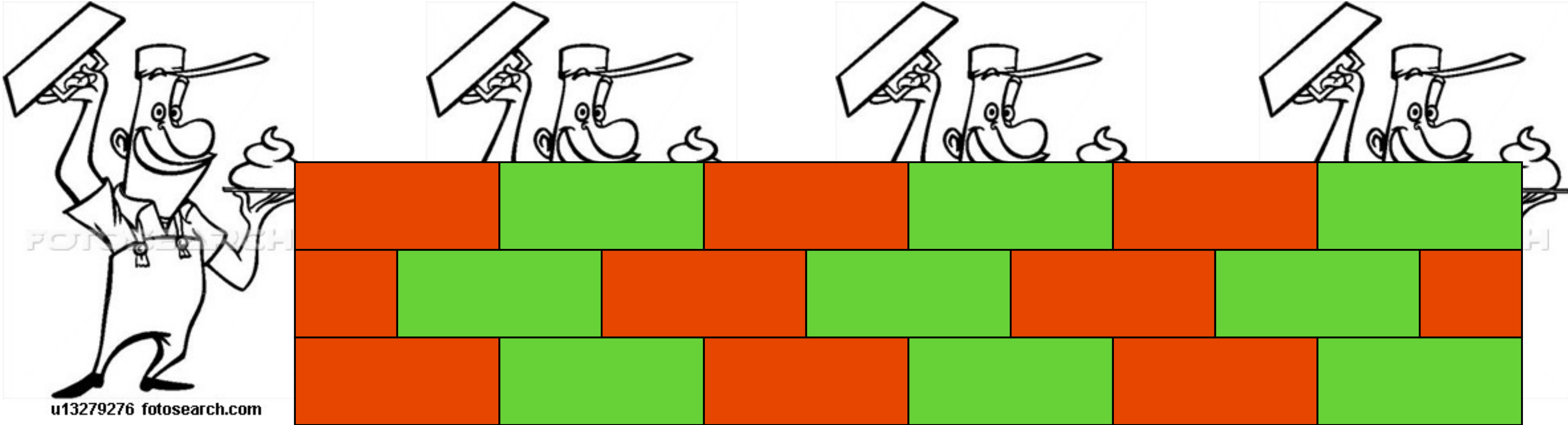


u13279276 fotosearch.com

Which builder places which brick?

Coordination

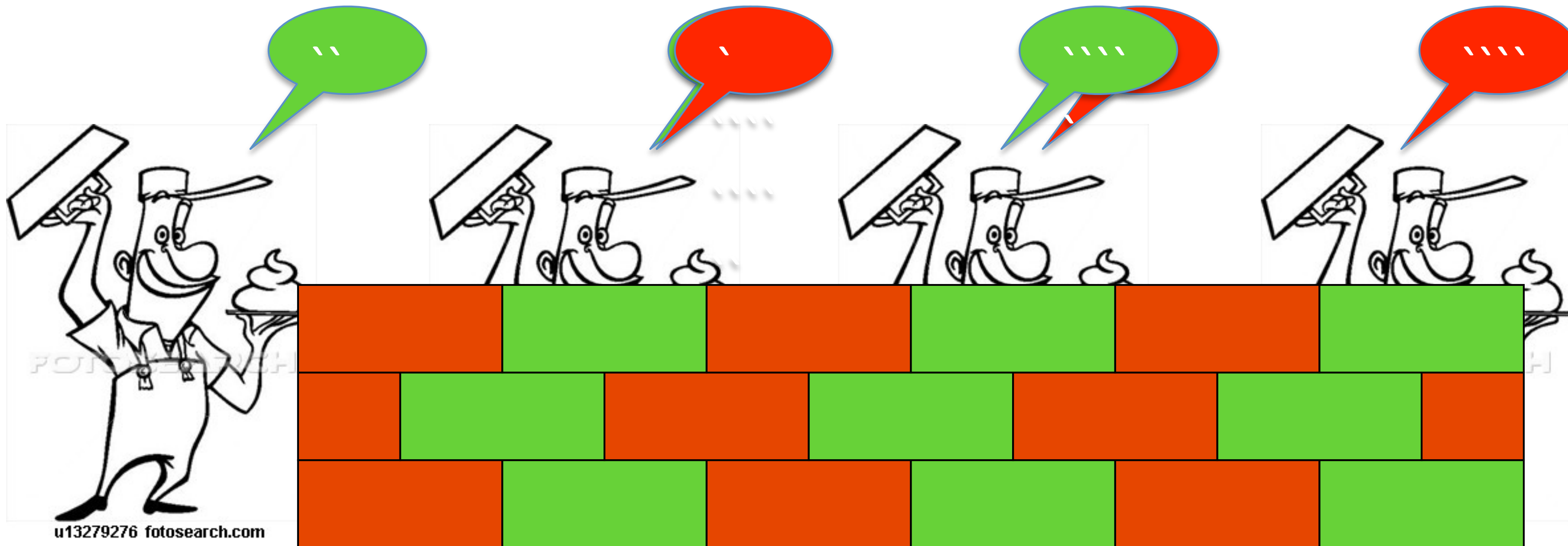




Two adjacent bricks cannot be built at the same time

Communication

University of St Andrews



All builders need to agree when to build a brick...

A-E, F-J, K-N, O-Z

We need structure
We need abstraction

We don't need another brick in the wall

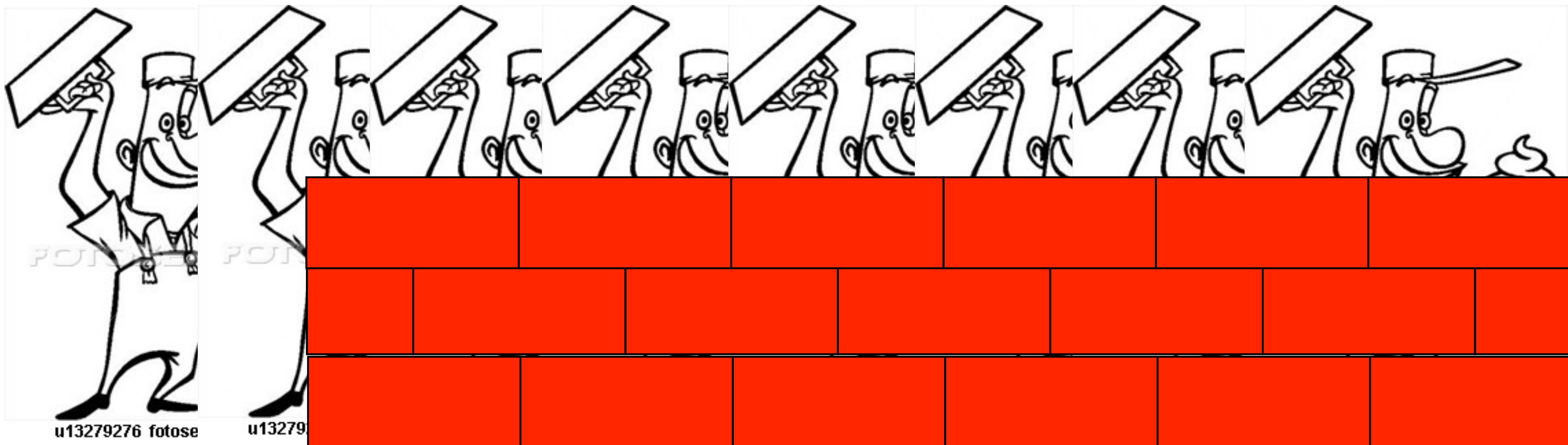
Parallel Patterns

- A *pattern* is a common way of introducing parallelism
  - helps with program design
  - helps guide implementation
- Often a pattern may have several different implementations
  - e.g. a *map* may be implemented by a *farm*
  - these implementations may have different performance characteristics

Patterns and Structure

University of St Andrews

bsp(bricks, 3)



This is an instance of *Bulk Synchronous Parallelism (bsp)*

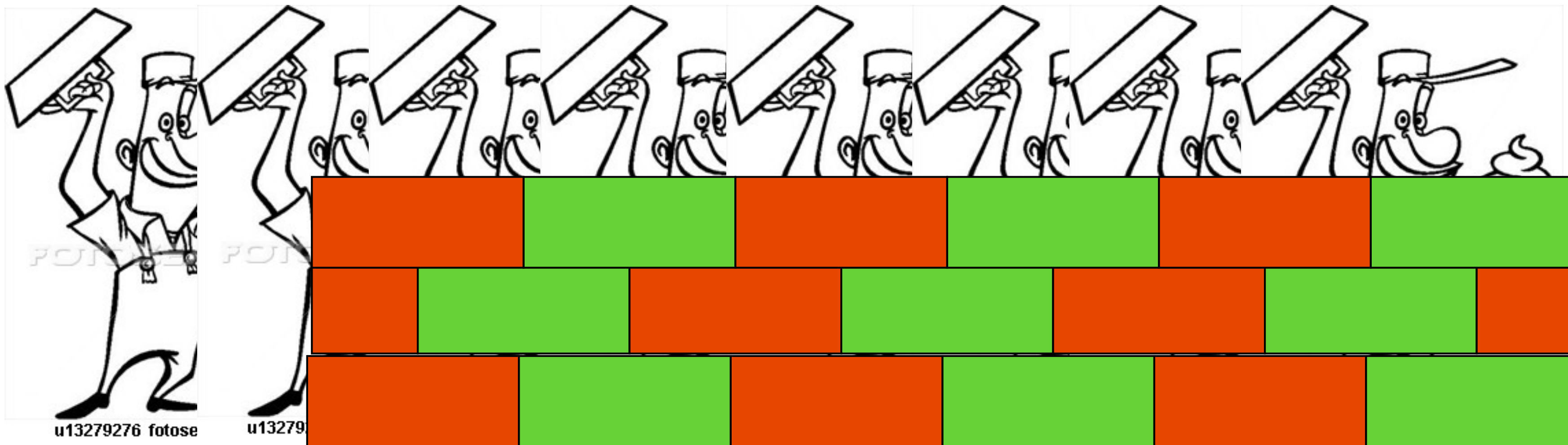
Each level is built before the next

Bricks in a level can be placed in parallel

Patterns can be Composed

University of St Andrews

bsp(stripe(bricks, 2), 3)



This refines *Bulk Synchronous Parallelism*
using a *striping* technique

Mapping can be to GPU/CPU using components

Graph Reduction

University of St Andrews

- *Graph Reduction* involves representing a program as an interconnected graph of heap objects.
- An unevaluated function call (a *thunk)* is represented as a *closure* with pointers to the function and its arguments
- When the result of a function call is produced, the thunk is *updated* with that result

1024: | pfib | → 1030 |

1030: | INT | 15 |

⇒

1024: | INT | 1973 |

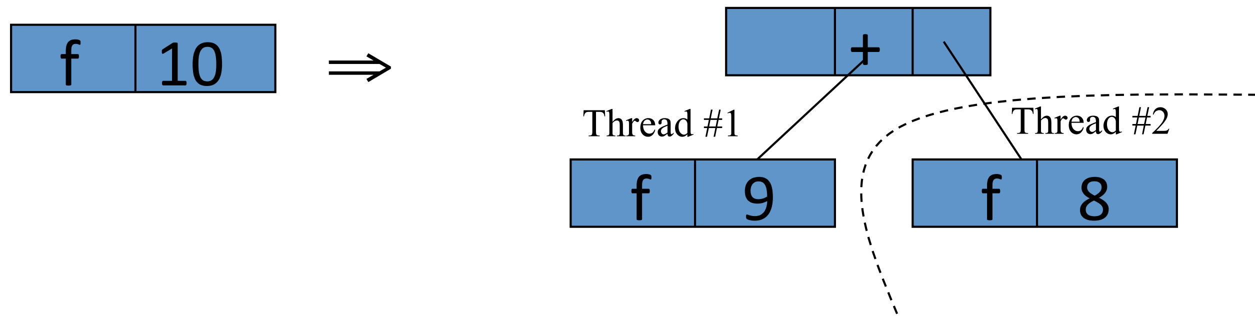
1030: | INT | 15 |

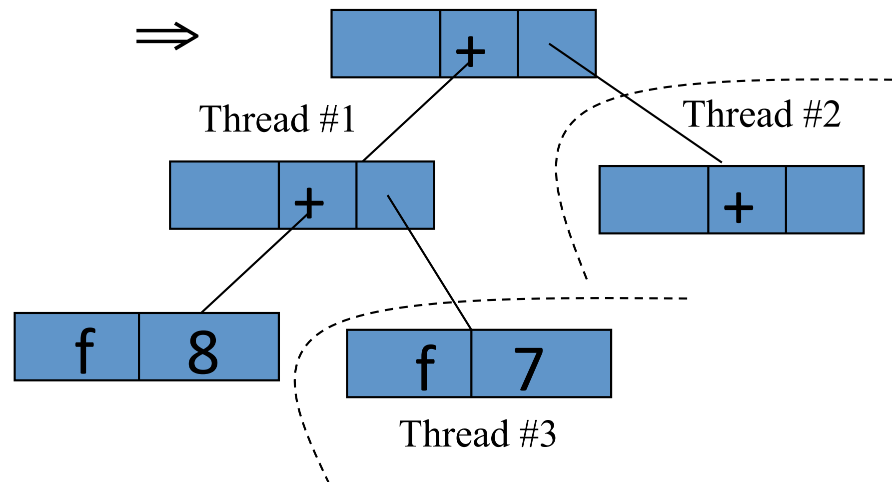
Parallel Graph Reduction

- Graph reduction is well-suited to parallelism since
  - there is no concept of a central program counter
  - concurrent reductions cannot affect the result of a program
  - all communication and synchronisation takes place via the graph

- The program graph acts as a shared resource
  - threads evaluate graph nodes (closures) and share results by updating the graph

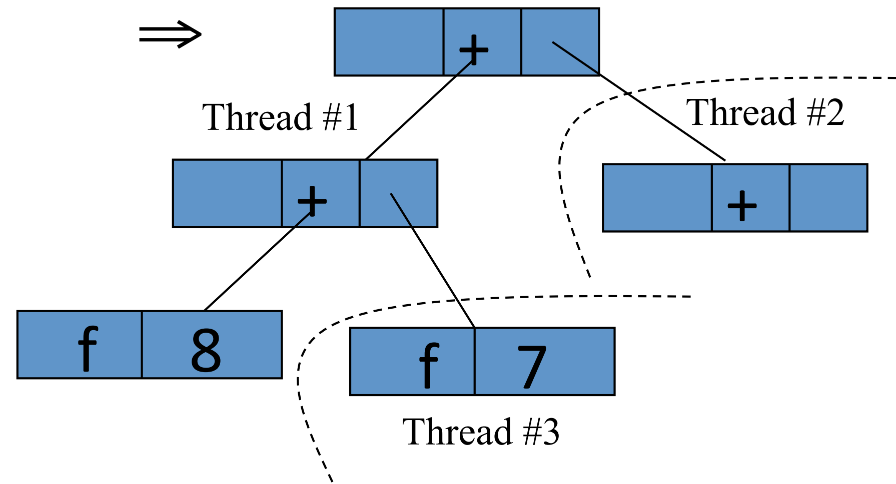
Parallel Graph Reduction Example

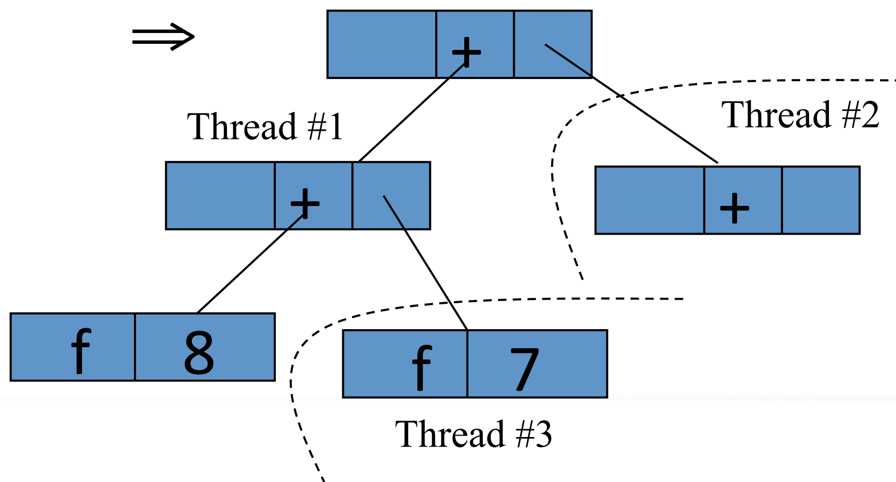


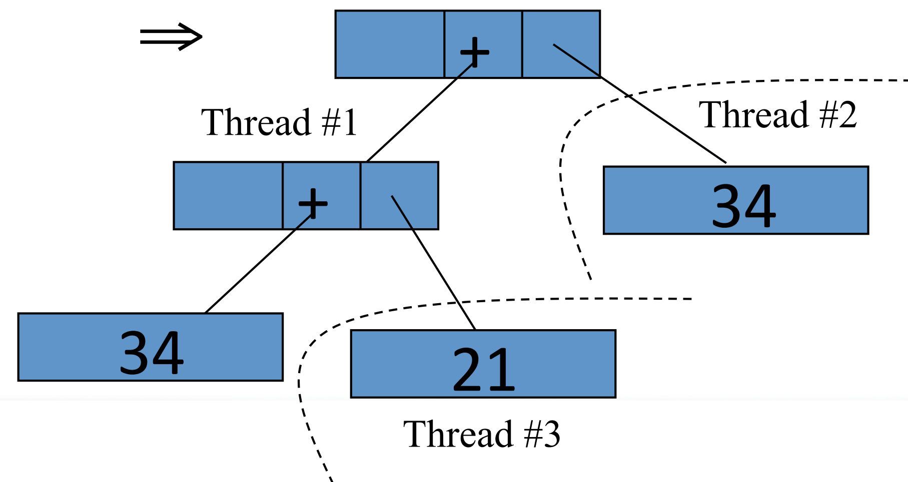


Parallel Graph Reduction Example (2)

University of St Andrews

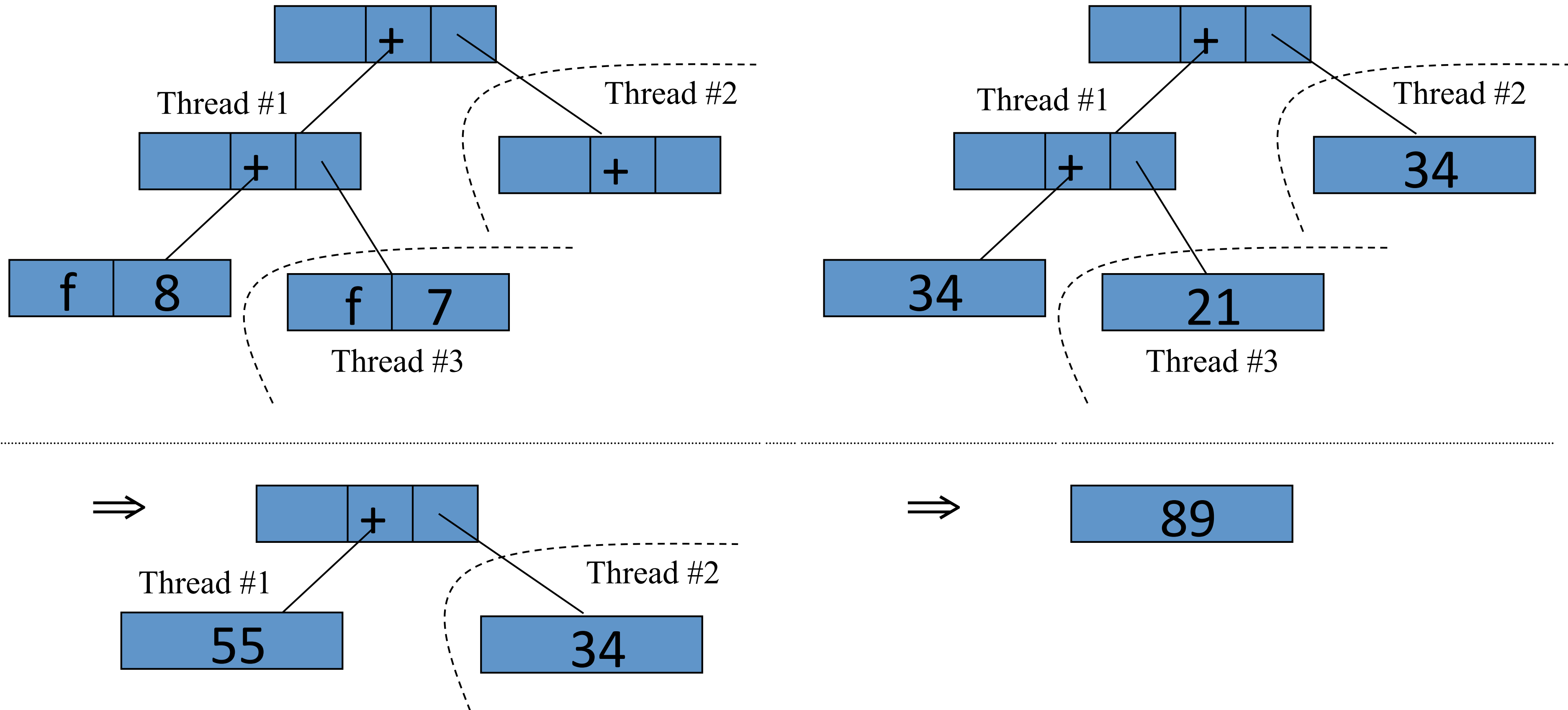






Parallel Graph Reduction Example (3)

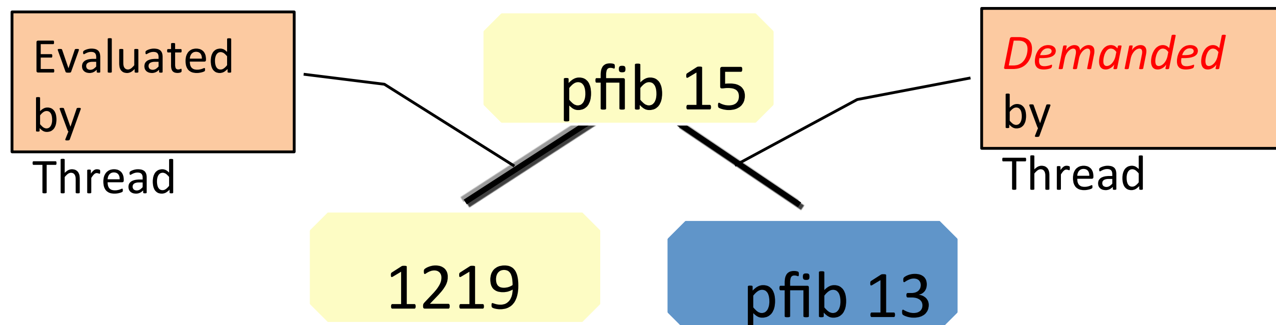
University of St Andrews



Synchronisation in Evaluate-and-Die

University of St Andrews

- Synchronisation happens when values are *demanded*
- This is the main way to introduce *communication* in GpH

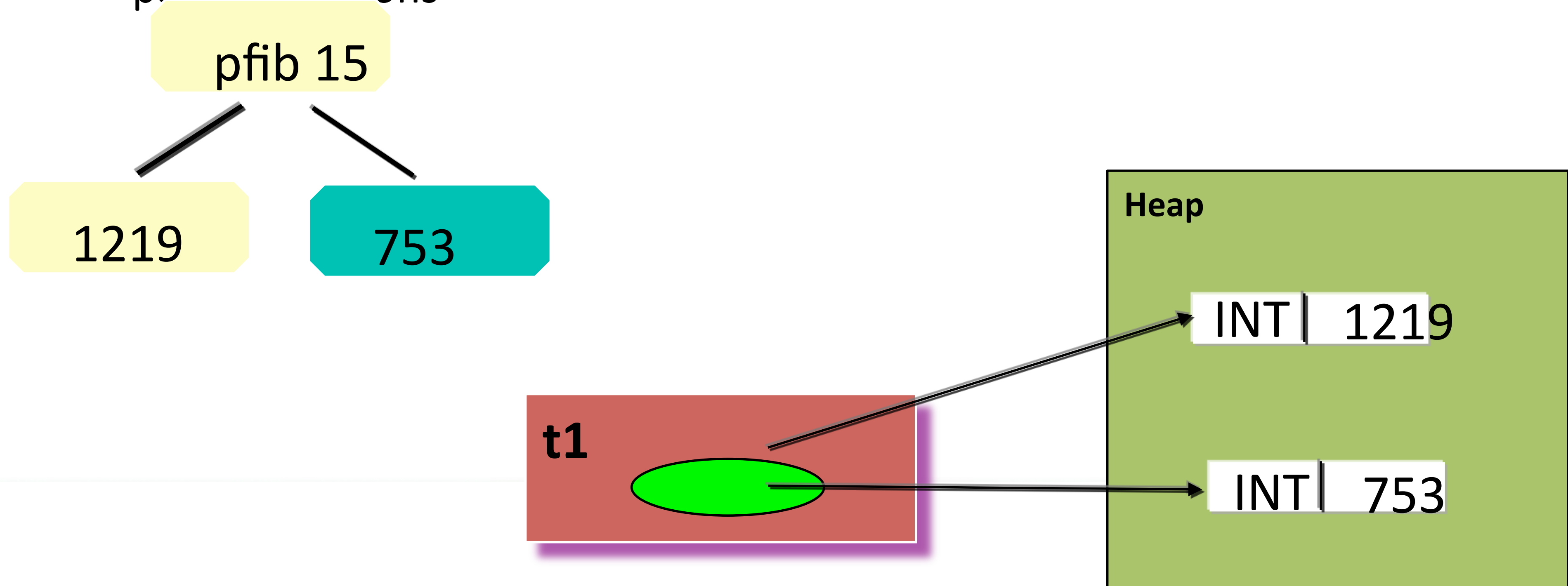


- There are three basic cases
  - The demanded value has already been evaluated (it is a *normal form*)
  - The demanded value has not yet been evaluated (it is a *thunk)*
  - The demanded value is under evaluation by another thread (it is a *black hole*)

Synchronisation: Normal Form

University of St Andrews

- A value is in *normal form* if it has already been evaluated
- Normal forms include
  - constants
  - data structures
  - partial functions

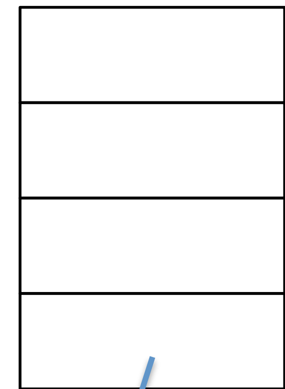


Synchronisation: Thunk

University of St Andrews

- *Thunks* can be evaluated by the running thread
  - *Even if* the thunk is the target of a spark
  - This automatically increases thread granularity!
  - No need to create a new thread to produce the value

Spark Pool



pfib 15

1219

pfib 13

Heap

INT 1219

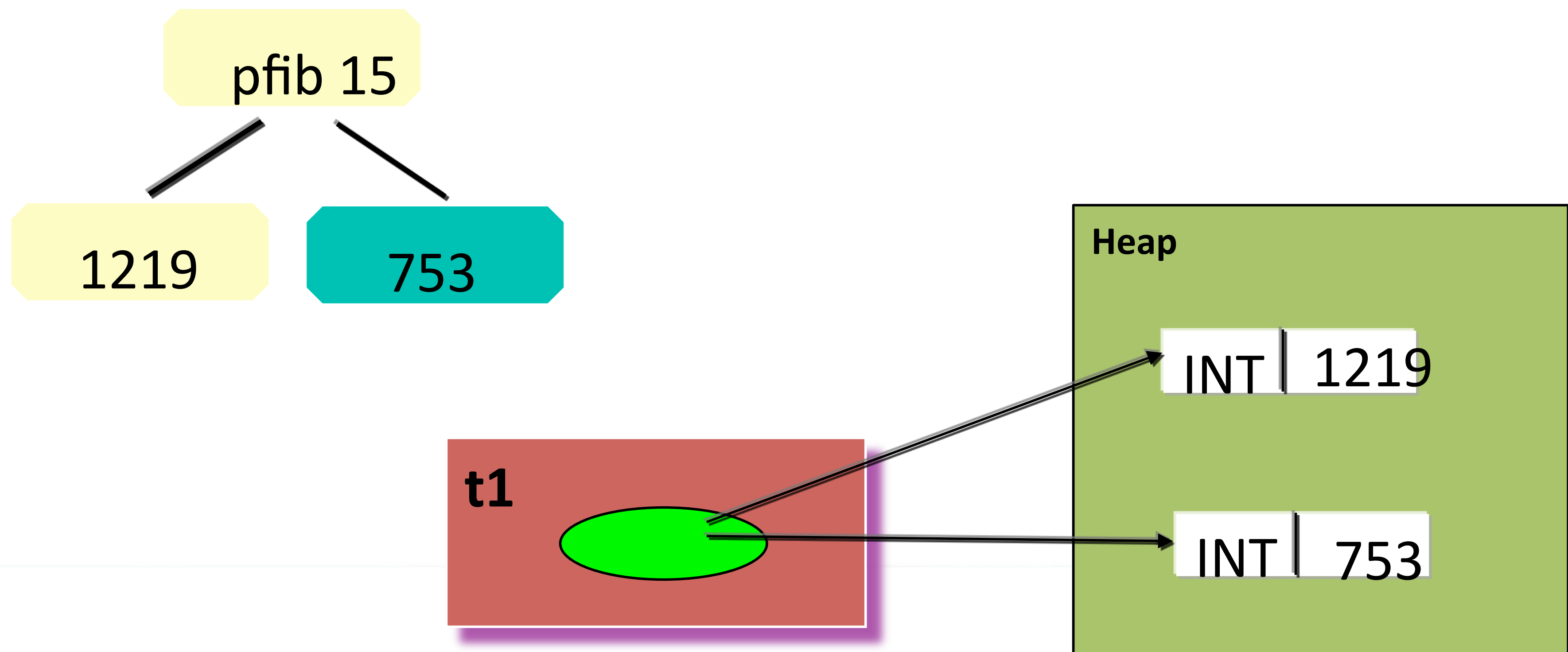
pfib 13

t1

Synchronisation: Thunk

University of St Andrews

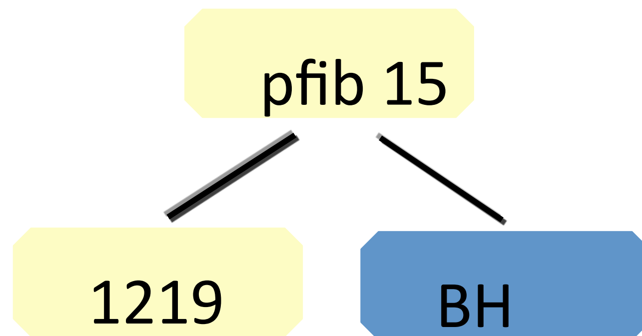
- Once a thunk is *evaluated* by the running thread the situation is as before
  - The result of evaluation is always a normal form

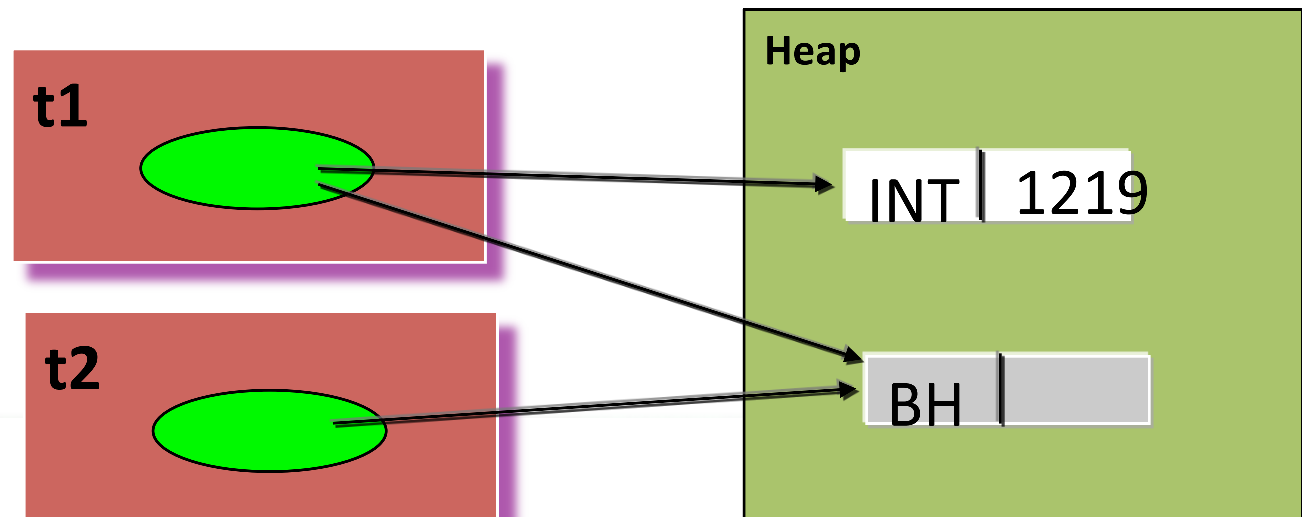


Synchronisation: Black Hole

University of St Andrews

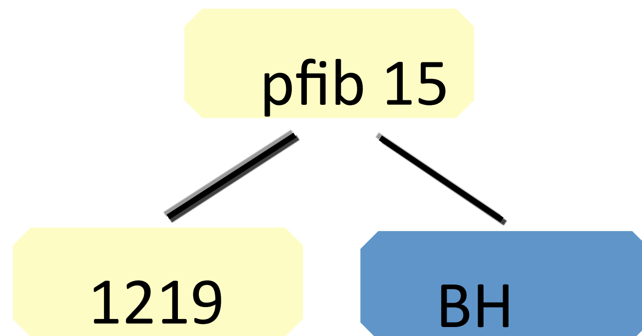
- *Black holes* are thunks that are already under evaluation
  - another thread will produce the value

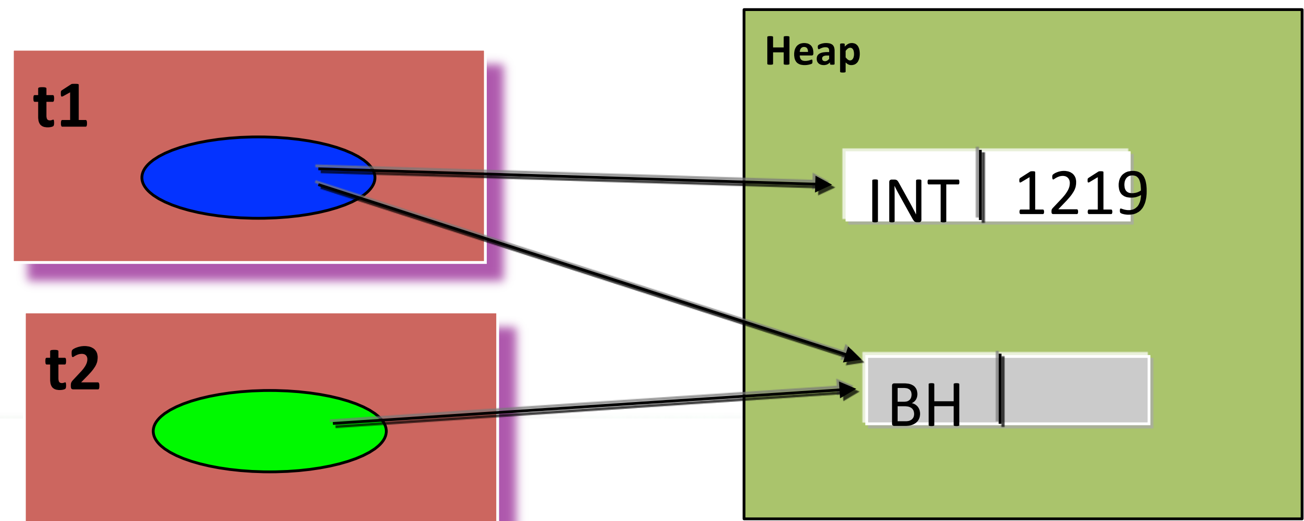




Synchronisation: Black Hole

- The demanding thread must *block* until the value is produced

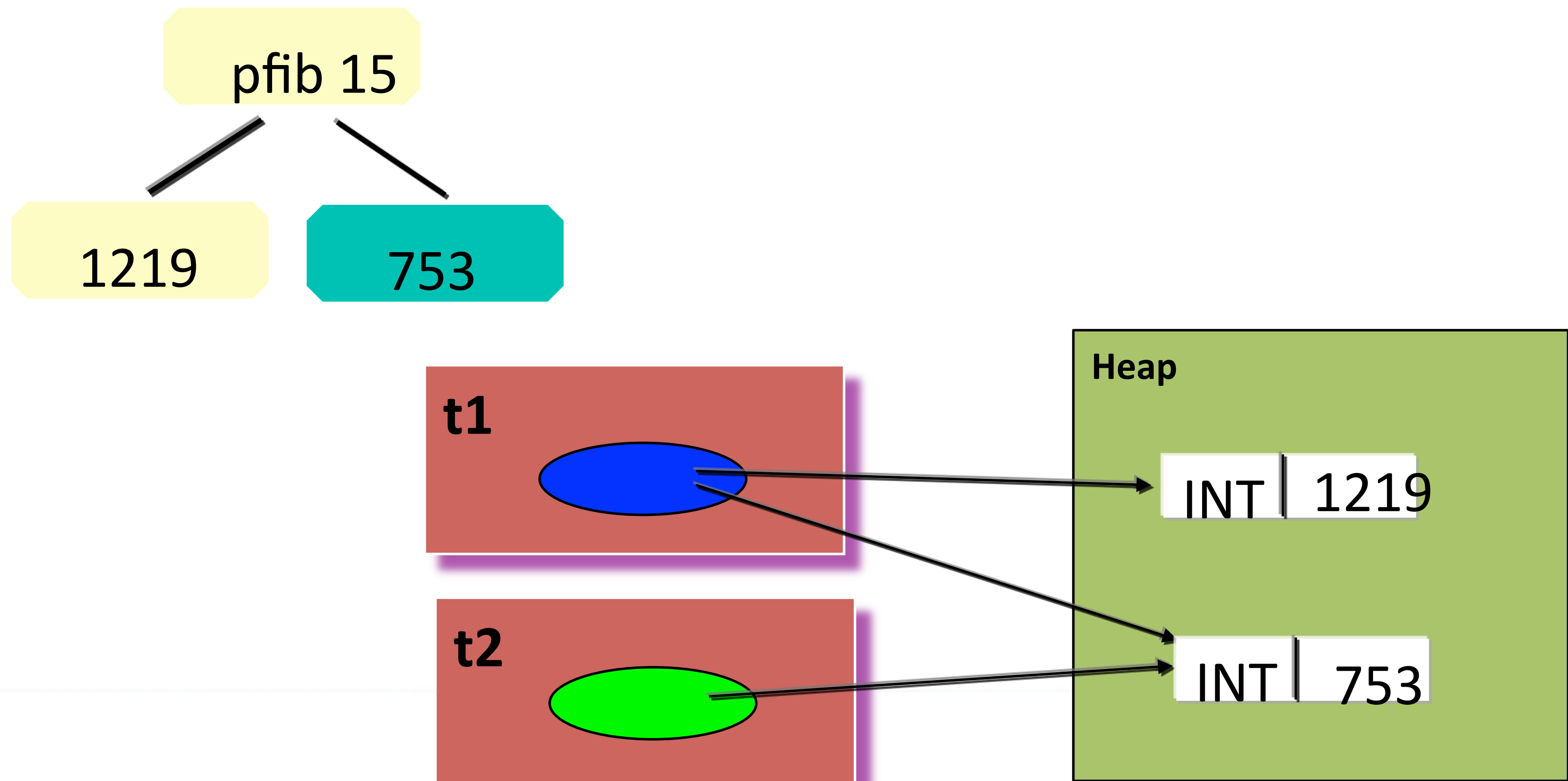




Synchronisation: Black Hole

University of St Andrews

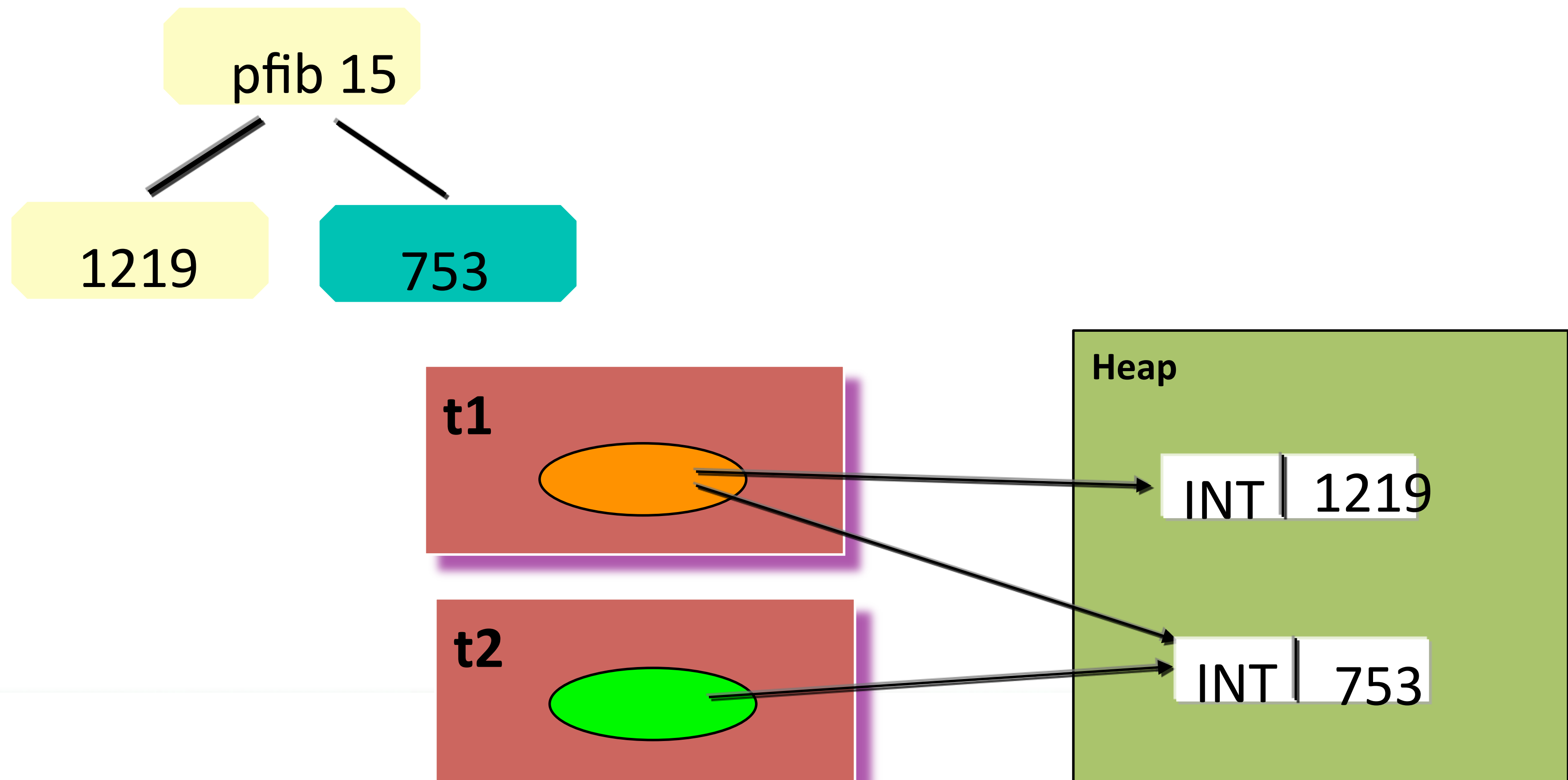
- Once the other thread yields a normal form, it can be *woken*



Synchronisation: Black Hole

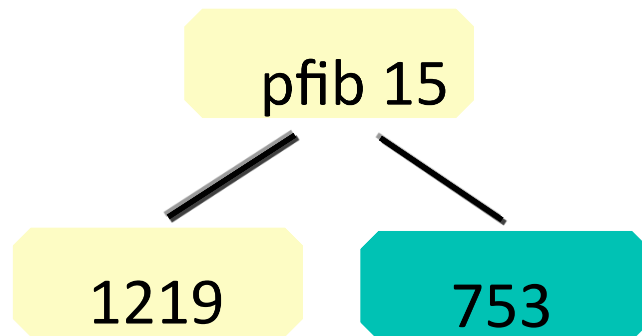
University of St Andrews

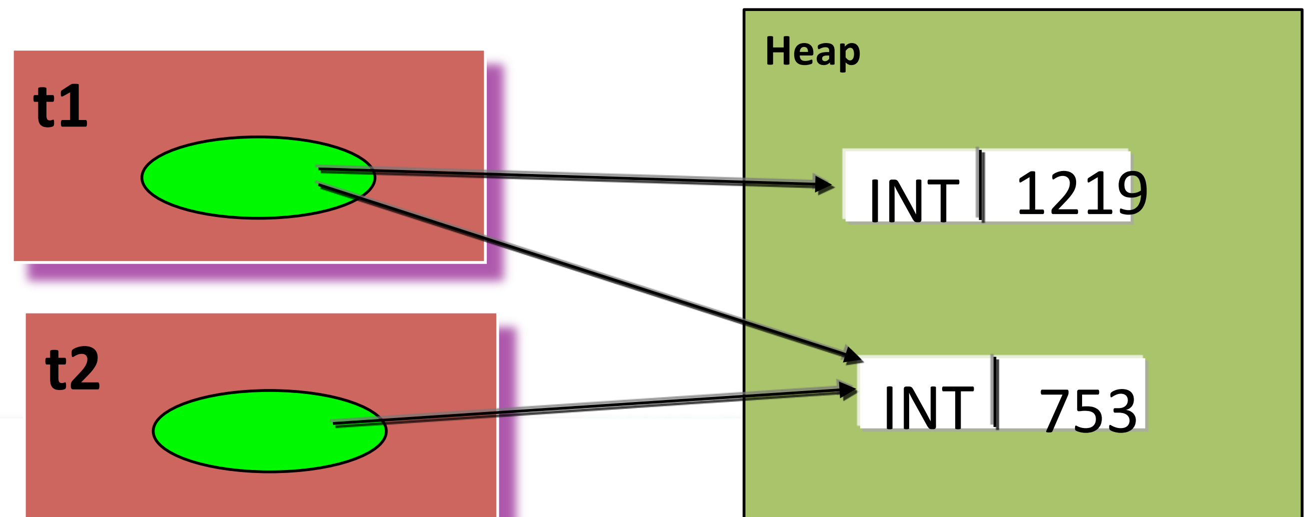
- Once the other thread yields a normal form, it can be *woken*



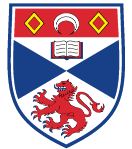
Synchronisation: Black Hole

- And then produce its result





Data Parallel Patterns



Data Parallelism

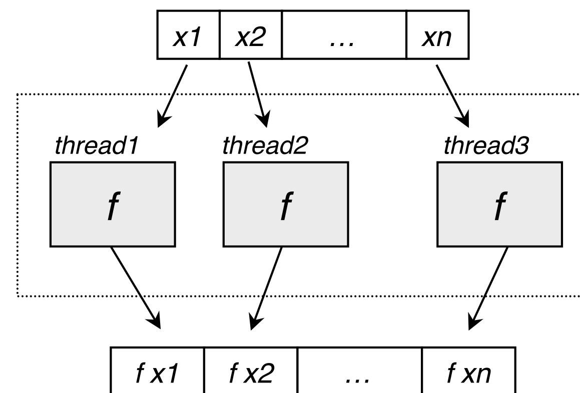
University of St Andrews

- Operations are applied *independently* to several data items
  - e.g. all elements of a list or array
- Easy to conceptualise, easy to implement
  - lots of independent computations, all the same size
  - very regular parallel structure
- May produce large amounts of very fine-grained parallelism
- A good fit to massively parallel architectures
  - e.g. GPUs, SIMD processors in general
- Examples include:
  - parallel maps
  - parallel scans
  - map-reduce

Parallel Maps

University of St Andrews

- One of the simplest forms of data parallelism
- Given an input data structure
  - map a function across each element in *parallel*



- A parallel version of a sequential *map* function

```
map f [] = []
map f (x : xs) = f x : map f xs
```

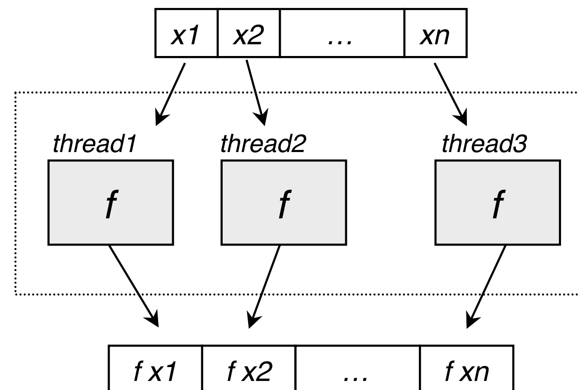
- Maps can be used on any regular data structure
  - arrays, sets etc.

Parallel Map Implementation

University of St Andrews

- A parallel map can easily be implemented using *par*

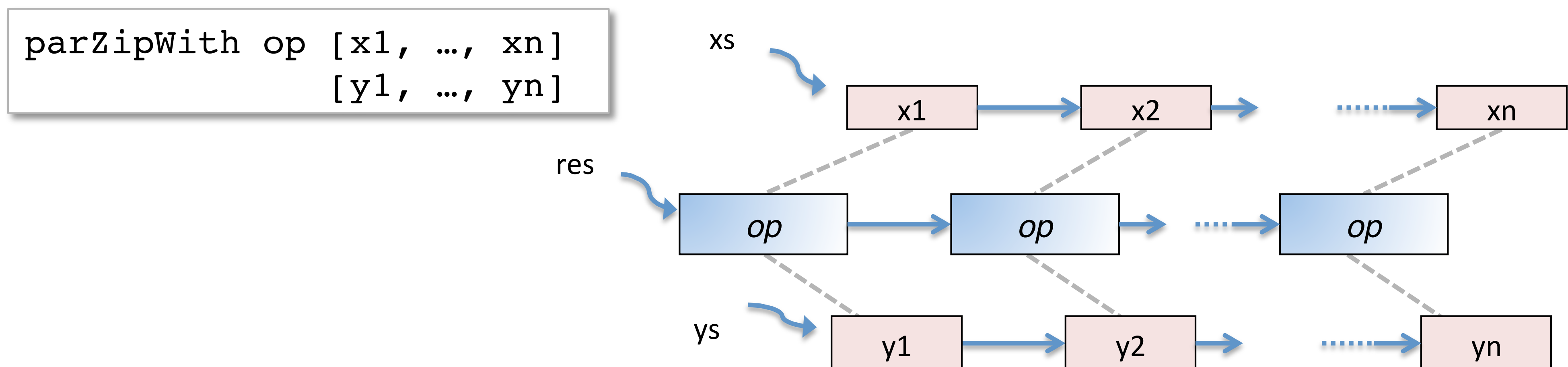
```
map f [] = []
map f (x : xs) = let fx = f x in
                   fx `par` (fx : map f xs)
```



Parallel zipWith

University of St Andrews

- A kind of *map* over two input lists
  - map a function across each pair of elements in *parallel*



- A parallel version of a sequential *zipWith* function

```
zipWith f [] [] = []
zipWith f (x : xs) (y:ys) = f x y : zipWith f xs ys
```

- Example

```
zipWith (+) [1..3] [4..6]  = [1+4,2+5,3+6] =  [5,7,9]
```

Parallel Fold

University of St Andrews

- A more complex pattern
  - puts an operator *between* each pair of elements in a list
- A parallel version of a sequential *fold* function

```
fold f z [] = z
fold f z (x : xs) = f x (fold f z xs)
```

- Examples:

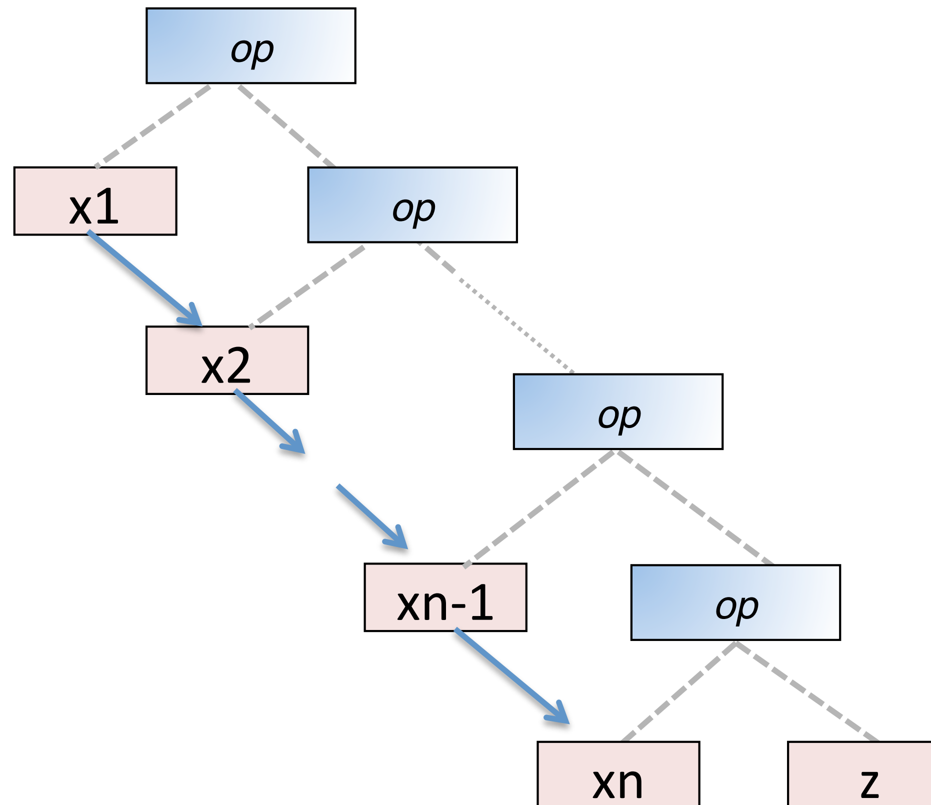
```
sum = fold (+) 0
product = fold (*) 1
```

- Also known as *reduce*

Sequential Fold

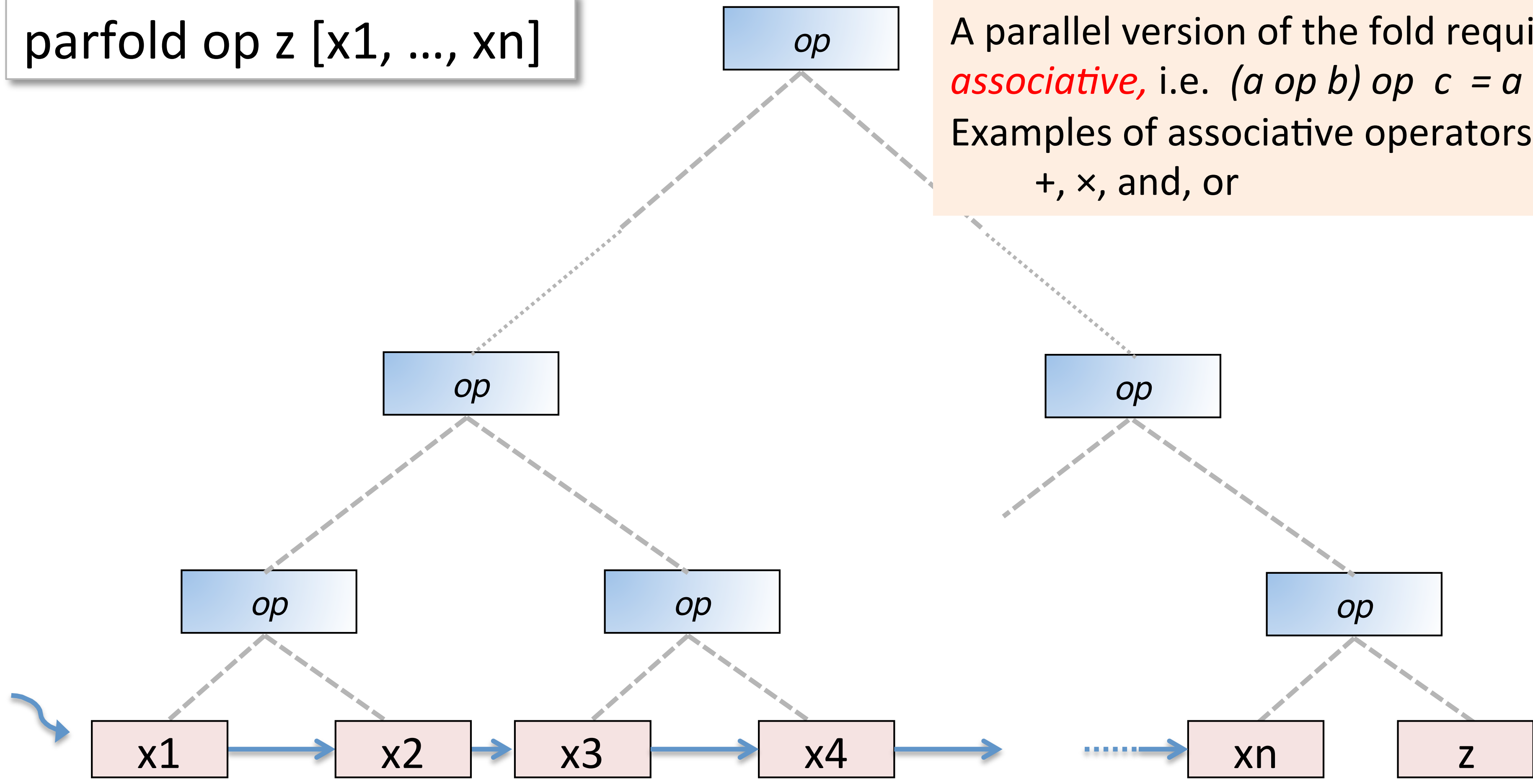
University of St Andrews

fold op z [x1, …, xn]



This is a *right* fold
Left folds also exist

Parallel Fold



A parallel version of the fold requires *op* to *associative,* i.e. *(a op b) op c = a op (b op*

Examples of associative operators include

+, ×, and, or

University of St Andrews

ParaPhrase

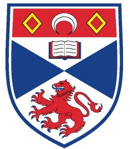
Bulk Synchronous Parallelism (BSP)

University of St Andrews

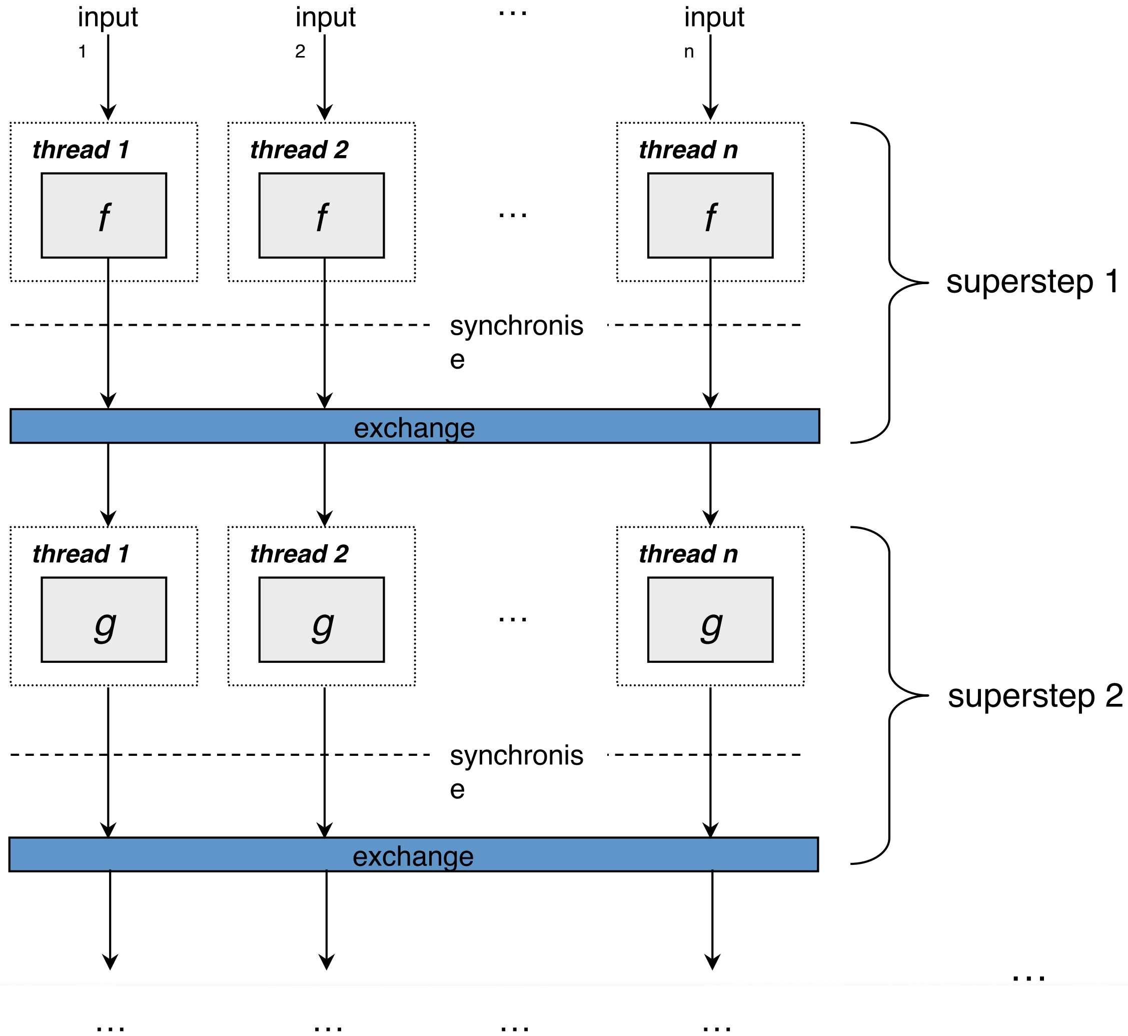
- A more sophisticated form of data parallelism
- Threads work on independent inputs, then *exchange* results
- Computation proceeds in a series of *supersteps*
  - all threads do the same thing in each superstep
  - each superstep may do different computation
- During each superstep, each thread
  - works on its own inputs using *private local data* and *shared global data*
- At the end of a superstep, a thread
  - *exchanges* its part of the global data with other threads
  - *synchonises* with the other threads to indicate that data has been exchanged

Leslie G. Valiant, “A bridging model for parallel computation”,
Communications of the ACM, Volume 33 Issue 8, Aug. 1990

The BSP Pattern



Synchronise /exchange steps may be reversed



BSP Synchronisation

University of St Andrews

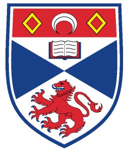
- Synchronisation is implemented by a *barrier communication*
  - all threads participate in the barrier
  - no thread can proceed until all threads have synchronised
  - this guarantees that they have all produced and exchanged information
  - barriers are nice because cannot cause deadlock/livelock
- This is sometimes actually overly strong
  - in Haskell, for example, we could use black-holing to block later computations
  - it's not always necessary for all threads to wait on all other threads
- It may not be strong enough
  - e.g. if global data is shared, it shouldn't be updated if other threads are still working on it...
- Barriers may be expensive
  - but on a multicore machine, they can usually be implemented cheaply

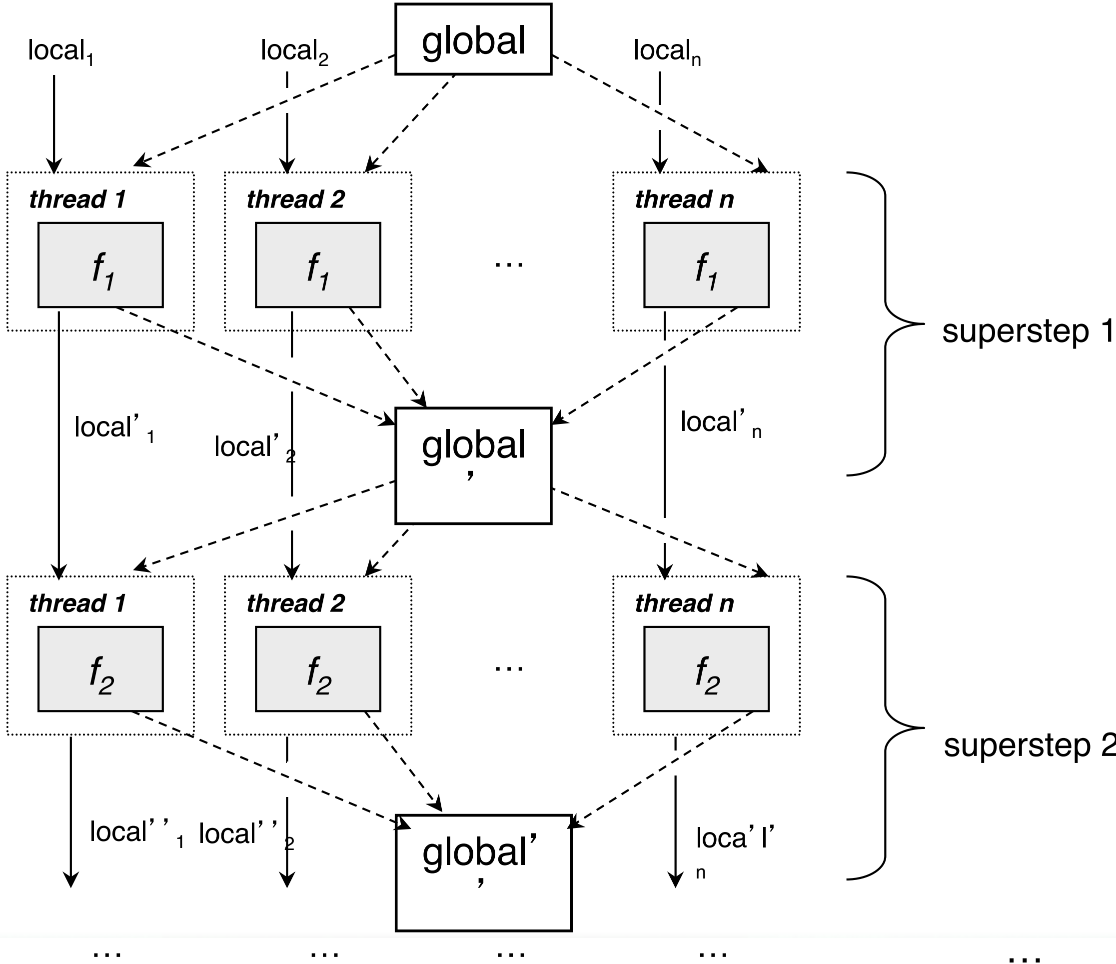
Exchanging Information

University of St Andrews

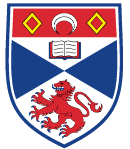
- Each thread has its own local state
  - this may be as simple as a unique thread id!
  - this may be altered as a result of the computation at each superstep
- Each thread also has access to the global state
  - as produced by combining the results of the previous superstep
- Each thread produces new local and global state
  - the global state can only be changed during the exchange
  - normally each thread owns (changes) a unique part of the global state
    - e.g. part of an array

BSP Local and Global Exchanges





BSP Timings



Total Execution Time = t1 + t2 + 2 × c

The BSP Cost Model

University of St Andrews

- BSP computations have a simple and elegant cost model
  - if f_i is the maximum cost of the function at step i
  - c_{ex} is the cost of an exchange/synchronisation
  - and there are m supersteps
  - then the cost of a BSP computation is:

$$\sum_{i=1}^{m} f_i \quad + \quad c_{ex} \times (m-1)$$

Map-Reduces

University of St Andrews

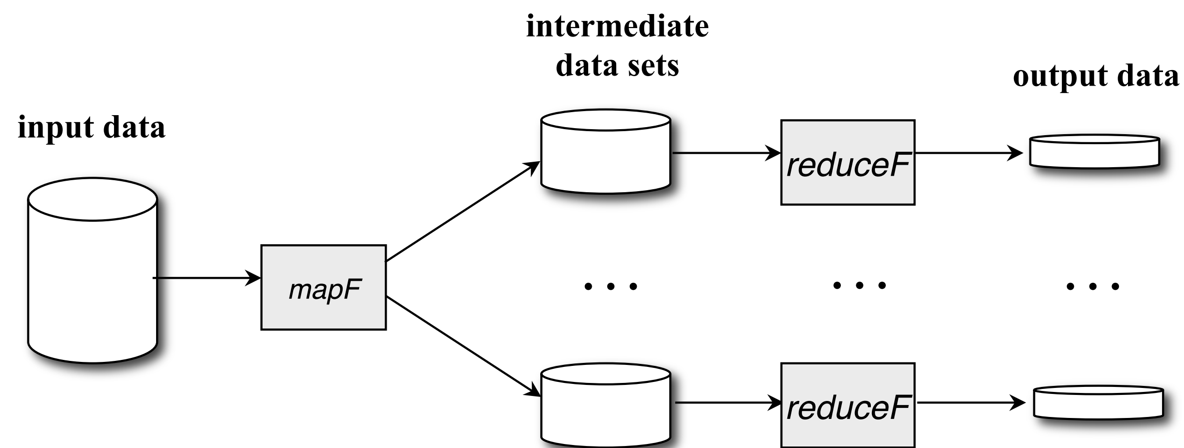
- A *map-reduce* combines two kinds of computation
  - a *map* where a function is mapped across a data structure
  - a *reduce* where map results are reduced to simpler values

```
import List

mapreduce n mapf reducef input = map reducef ls
         where ls = chunk n (map mapf input)

chunk n [] = []
chunk n inputs =
    let ( c, rest ) = splitAt n inputs in
    c : chunk n rest
```

Parallelism in Map-Reduce

University of St Andrews

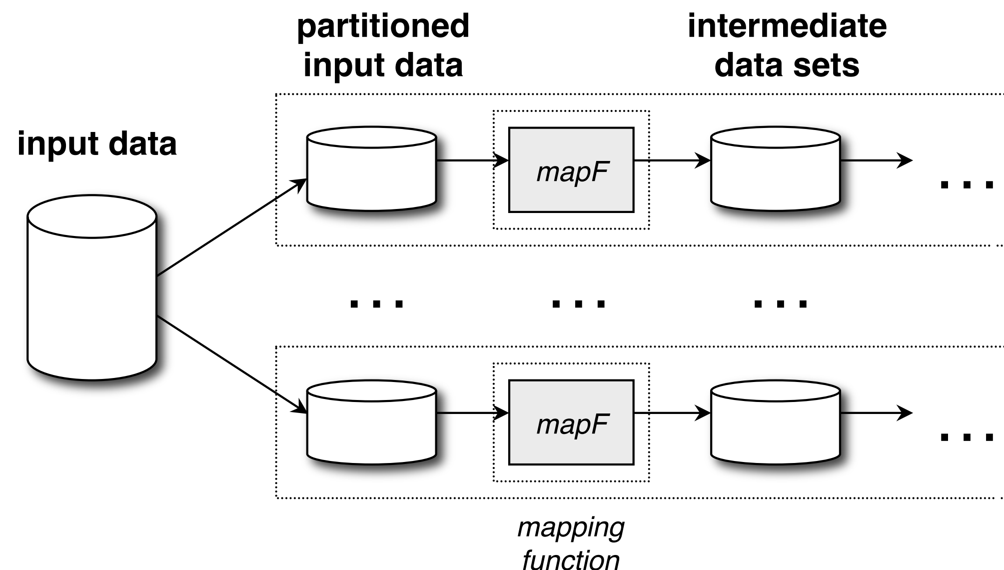
- The *reduce* operations can clearly be run in parallel
  - using data parallelism (e.g. a parallel map)



Map-Reduce with Increased Parallelism

University of St Andrews

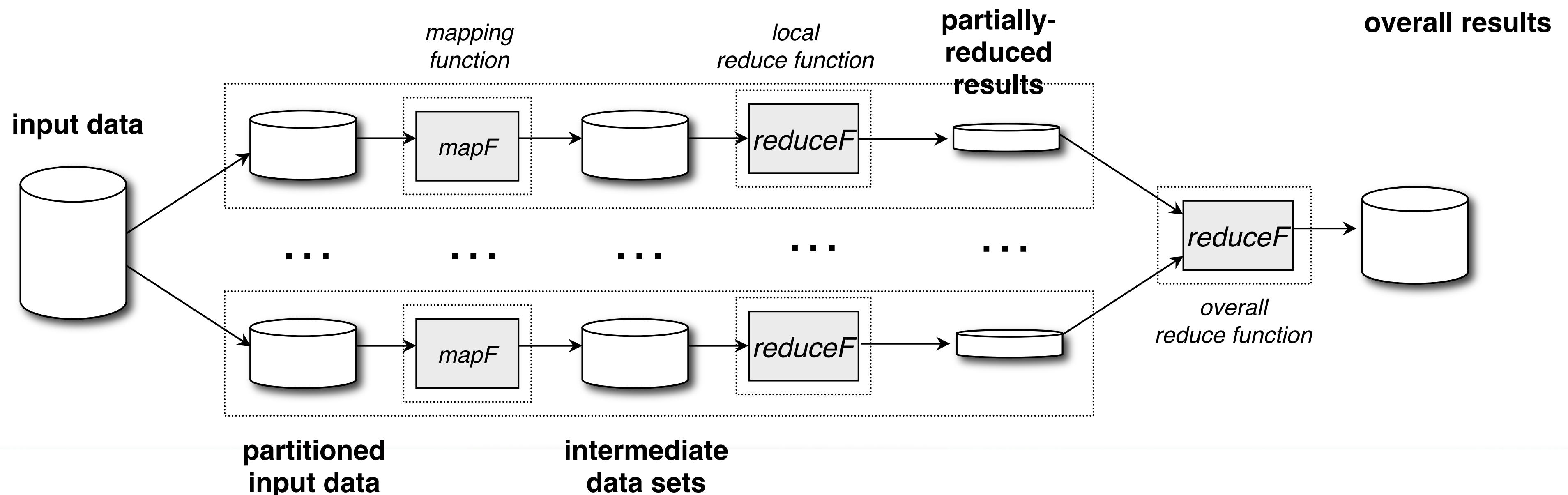
- The *map* operations can also be run in parallel
  - by splitting the data and then mapping each map in parallel
  - also using data parallelism



Map-Reduce with Increased Parallelism

University of St Andrews

- Usually, the final results also need to be reduced
  - using the same *reduce* function
- *map-reduce* is a good pattern for distributed parallelism
  - e.g. Google/Hadoop



Revised Map-Reduce Definition

University of St Andrews

- The revised definition has two uses of *reducef*

```
import List

mapreduce' n mapf reducef input = reducef (map reducef
ls)
          where ls = map mapf (chunk n input)

chunk n [] = []
chunk n inputs =
    let ( c, rest ) = splitAt n inputs in
    c : chunk n rest
```

Ralf Lämmel, "Google's MapReduce Programming Model – Revisited", *Science of Computer Programming*, **68**:3, 2007.

`http://code.google.com/edu/parallel/mapreduce-tutorial.html`

Scans

University of St Andrews

- The Haskell *scan* function is a cross between a *map* and *fold*
- Unlike map-reduce, the fold uses all the data
  - and returns the "intermediate" steps

```
scanl :: (a -> b -> a) -> a -> [b] -> [a]
scanl f q xs = q: (case xs of
              [ ] ->[ ] (x:xs) -> scanl f (f q x) xs)
```

- For example

```
scanl (+) 0 [1 .. 4]
      = [0,0+1,(0+1)+2,((0+1)+2)+3,(((0+1)+2)+3)+4]
      = [0,1, 3, 6, 10]
```

General Definition of a Scan

- Given a sequence of *n* values

  $[a_0,\ldots,a_{n-1}]$

- an initial value

  z

- and some operation

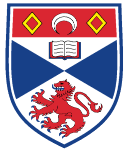
  $\oplus$

- then a (left) scan will produce

  $[z, z \oplus a_0, (z \oplus a_0 \oplus a_1), \ldots (z \oplus a_0 \ldots \oplus a_{n-1})]$

- *also known as prefix scan*

Parallel Scan



t1	t2	t3	t4	t5	t6	t7	t8
a_0	a_1	a_1	a_1	a_1	a_1	a_1	a_1
a_0	$\sum_{i=0}^{1} a_i$	$\sum_{i=1}^{2} a_i$	$\sum_{i=2}^{3} a_i$	$\sum_{i=3}^{4} a_i$	$\sum_{i=4}^{5} a_i$	$\sum_{i=5}^{6} a_i$	$\sum_{i=6}^{7} a_i$
a_0	$\sum_{i=0}^{1} a_i$	$\sum_{i=0}^{2} a_i$	$\sum_{i=0}^{3} a_i$	$\sum_{i=1}^{4} a_i$	$\sum_{i=2}^{5} a_i$	$\sum_{i=3}^{6} a_i$	$\sum_{i=4}^{7} a_i$
a_0	$\sum_{i=0}^{1} a_i$	$\sum_{i=0}^{2} a_i$	$\sum_{i=0}^{3} a_i$	$\sum_{i=0}^{4} a_i$	$\sum_{i=0}^{5} a_i$	$\sum_{i=0}^{6} a_i$	$\sum_{i=0}^{7} a_i$

Parallel Scan

University of St Andrews

- A parallel version of the scan requires $\oplus$ to be *associative*
  - i.e. $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
- A parallel scan can always be performed in $log_2(n)$ steps
  - where n is the length of the input
- Parallel scans are easily mapped to SIMD implementations
- The type of a parallel scan is less general than the sequential ones

Generalising Scan

University of St Andrews

- Scans can be generalised to arbitrary collection types
  - Sets, Vectors etc.

- *Segmented scans* are a special sub-class where the input is a nested structure
  - effectively flatten the input structure
  - allows nested data parallelism to be transformed to flat data parallelism

Sergei Gorlatch, "Systematic Efficient Parallelization of Scan and Other List Homomorphisms", *Proc. Europar 1996,* Springer LNCS 1124, pages 401-408, 1996.

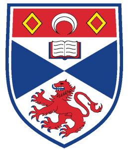
S. Sengupta, M. Harris, and M. Garland. **Efficient parallel scan algorithms for GPUs**. NVIDIA Technical Report NVR-2008-003, December 2008

Next Lectures

University of St Andrews

- More Parallel Patterns
  - Some Task-Parallel Patterns
- Skeletons
  - Implementing parallel patterns directly
  - Template-based programming
- Evaluation Strategies
  - Flexible, programmable parallelism
  - Can be used to implement patterns and/or skeletons
  - Can also program arbitrary new forms of parallelism

Task Parallelism



Some Issues with BSP

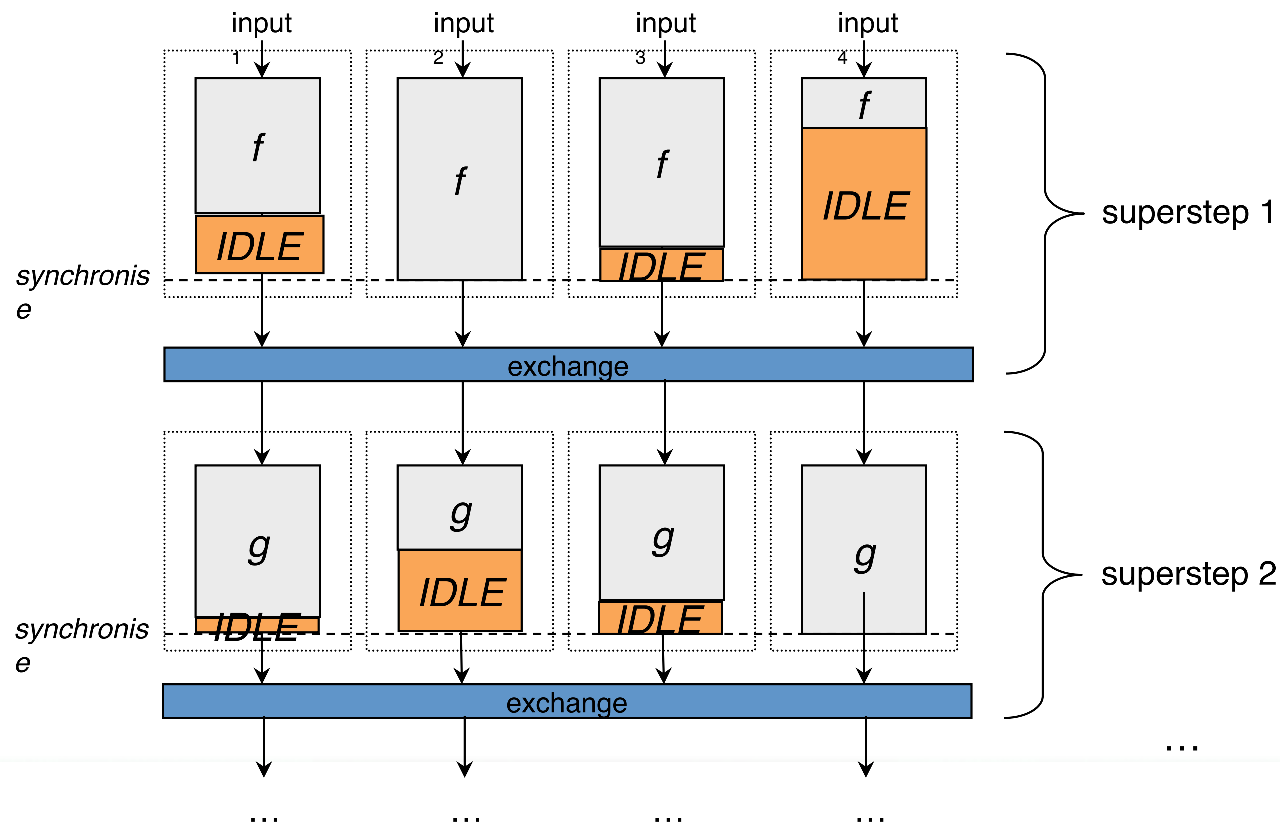
University of St Andrews

- What if not all computations are the same size?
  - Irregular thread granularities
- What if the number of threads doesn't match the number of cores?
- How expensive is the barrier?

Unbalanced BSP Computations

University of St Andrews

- If thread granularities are not *balanced*, some cores may be idle



Task Parallelism

University of St Andrews

- Where parallelism does not come purely from the structure of the data, where computations are not the same for each item and/or where there are irregular granularities, this is *task parallelism* (also called *control parallelism*)

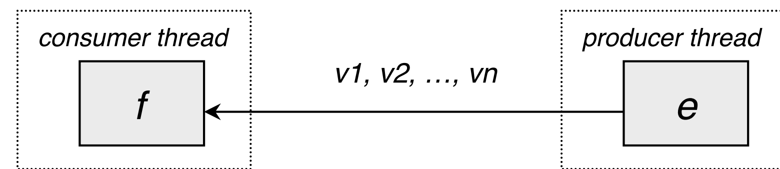
- Task parallelism is much harder to handle than data parallelism
  - less regular
  - harder to map to the available resources
  - can give much less massive parallelism
  - can be harder to identify the patterns
  - may need specialist support to implement

- However, many algorithms are not amenable to data parallelism
  - especially ones with complex or irregular control/data flows

Producer-Consumer Parallelism

University of St Andrews

- Producer-consumer is one of the simplest patterns of task parallelism
  - one thread produces values, which are consumed by another



- Producer-consumer can easily be implemented using *par*

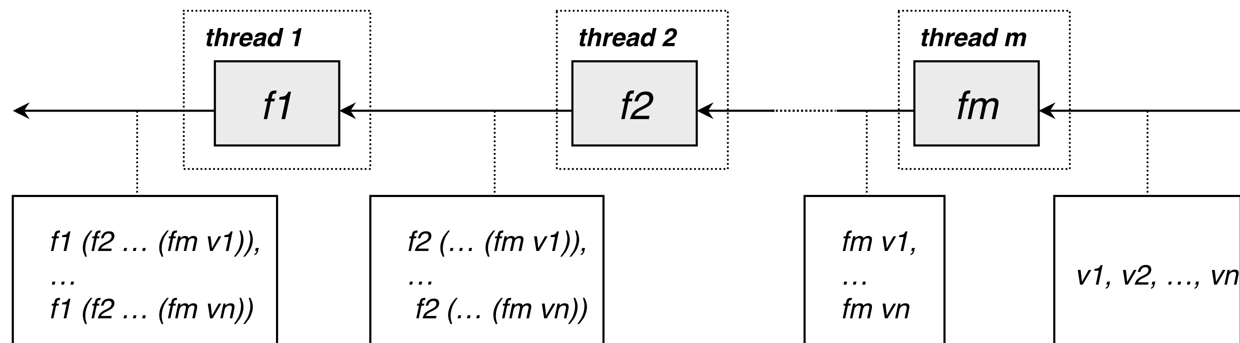
```
f $ e = f e
f $| e = e `par` f e
```

- This works in Haskell because of *lazy evaluation*
  - `provided that the argument is e.g. a list`

Parallel Pipelines

University of St Andrews

- A pipeline is a more complex form of producer-consumer parallelism
  - multiple pipeline stages
  - each stage produces values that are consumed by the next stage



- Pipelines can also be easily implemented using *par*

```
pipeline3 f1 f2 f3 x = f1 $ f2 $ f3 $ x
parpipeline3 f1 f2 f3 x = f1 $| f2 $| f3 $| x
```

Streams

University of St Andrews

- A stream is a pipeline where the operations are repeated over multiple values
  - in Haskell, e.g. when the input is a (lazy) list
- The stream produces results in the order that the inputs are given
  - there is exactly one output produced for each input value
    (this requirement might be relaxed in some stream models)

```
parstream3 f1 f2 f3 s = parpipeline3 (map f1) (map f2) (map f3) s
```

or

```
parstream3 f1 f2 f3 = parpipeline3 (map f1) (map f2) (map f3)
```

Arbitrary Length Pipelines

- It's easy to make a generic length pipeline by supplying a list of function arguments

```
parpipeline :: [a->a] -> a -> a
parpipeline fs z x = foldr (\f x -> f $| x) z fs x
```

- And this can easily be extended to a stream

```
parstream :: [a->a] -> [a] -> [a]
parstream fs s = parpipeline (map map fs) s
```

Divide-and-Conquer

- Divide a problem into subparts
- Solve each of those parts independently
  - using a divide-and-conquer approach
- When the problem is small enough to be trivially solvable
  - just solve it
- Combine the results for each solved part

Divide-and-Conquer Example

University of St Andrews

- Consider a definition of quicksort in Haskell

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

- This can be parallelised by sparking off each sub-sort

```
qsort [] = []
qsort [x] = [x]
qsort (x:xs) = losort `par` hisort `par` (losort ++ (x:hisort))
        where losort = qsort [y|y <- xs, y < x]
              hisort = qsort [y|y <- xs, y >= x]
```

Divide-and-Conquer Example

University of St Andrews

- There are two trivial (base) cases

```
qsort [] = []
qsort [x] = [x]
```

- Otherwise, the two sub-parts are *losort* and *hisort*

```
losort = qsort [y|y <- xs, y < x]
hisort = qsort [y|y <- xs, y >= x]
```

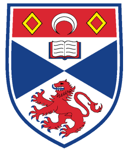
- These are sparked in parallel

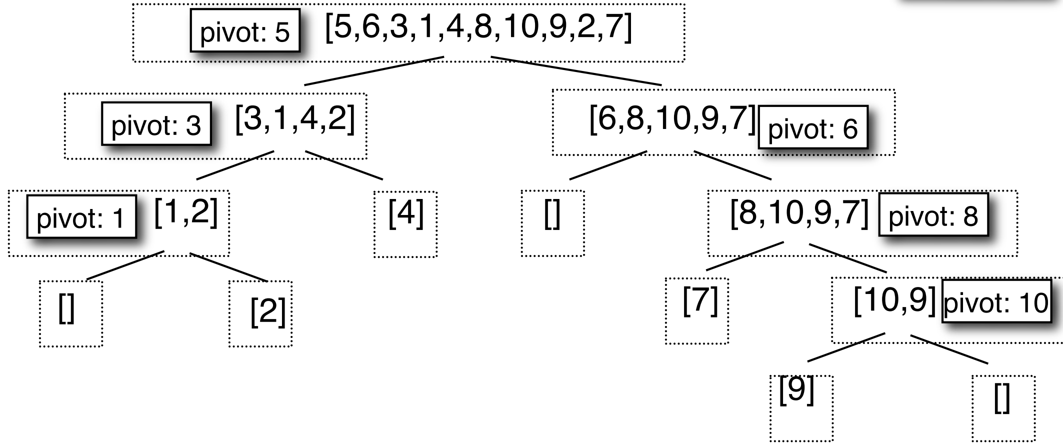
```
losort `par` hisort `par` ...
```

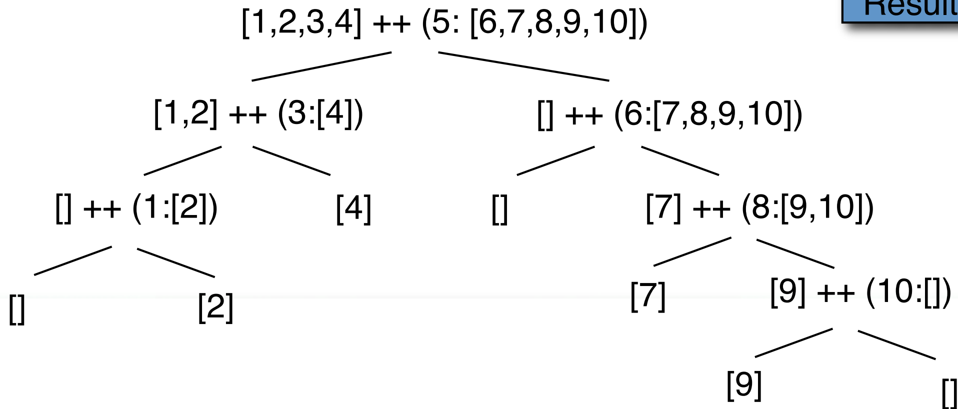
- Results from each sub-part are combined using ++ and :

```
(losort ++ (x:hisort))
```

Quicksort [5,6,3,1,8,10,9,2,7]







Advantages of DC Parallelism

1. a large number of tasks can be generated very rapidly

2. can be quickly disseminated across a parallel system
   - this is especially important for a distributed-memory system

3. the communication paths are well-defined and hierarchical
   - again, this is important in a distributed-memory system
4. no one processor becomes overloaded with the task of distributing work and marshalling results
   - the work is distributed among the available processors

Thresholding

University of St Andrews

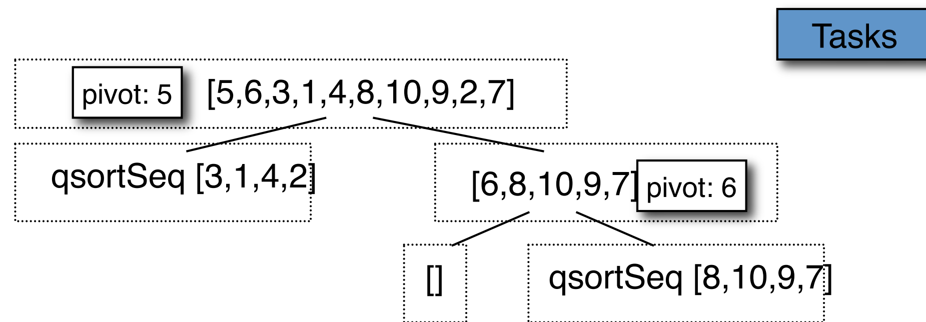
- DC parallelism can often be improved using a *threshold*

```
qsort [] = []
qsort [x] = [x]
qsort (x:xs) = losort `par` hisort `par` (losort ++ (x:hisort))
        where losort = parT qsort lows
              hisort = parT qsort highs
              lows =  [y|y <- xs, y < x]
              highs =  [y|y <- xs, y >= x]

parT f x | length arg <= threshold = fx `pseq` fx
         | otherwise = fx `par` fx
           where fx = f x
```

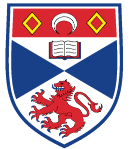
Thresholding Example

University of St Andrews

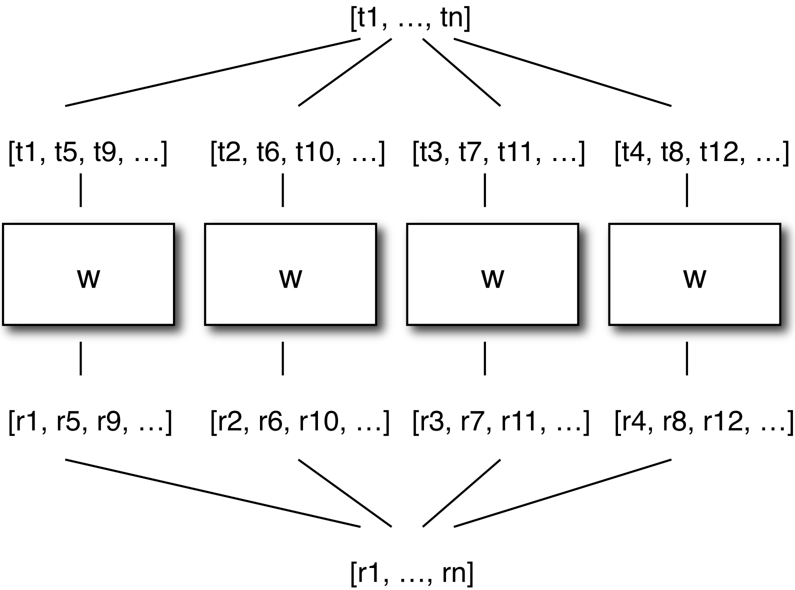
- `qsort [5, 6, 3, 1, 4, 8, 10, 9, 2, 7]`



Task Farms



- A task farm allocates tasks to workers equally
  - coarse grained workers can execute fine grained tasks
  - saves thread creation overhead

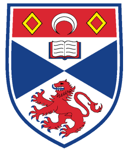


Task Farm Implementation

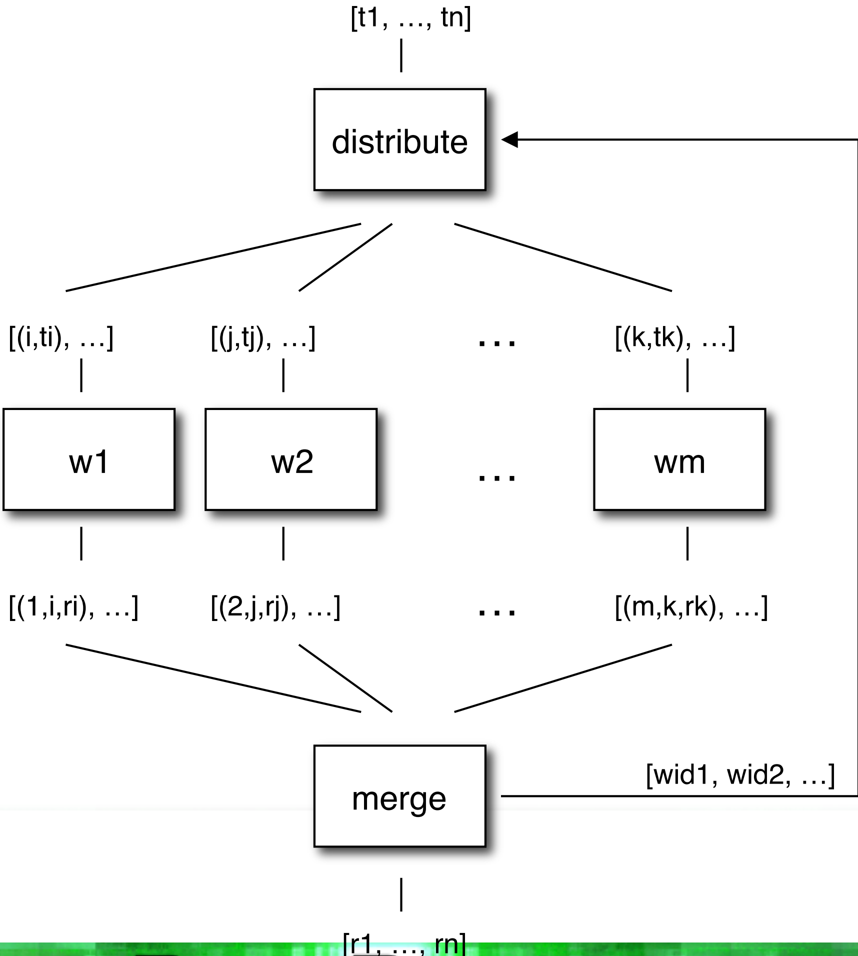
University of St Andrews

```
taskFarm :: (a -> [b]) -> Int -> [a] -> [[b]]
taskFarm f nWorkers tasks = concat results
        where results = parmap id (unshuffle nWorkers (map f tasks))

unshuffle :: Int -> [a] -> [ [a] ]
unshuffle n xs = [ takeEach n (drop i xs) | i <- [0..n-1] ]
        where takeEach :: Int -> [a] -> [a]
              takeEach n [] = []
              takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

Workpools



- A workpool is similar to a task farm
  - however tasks are allocated dynamically to workers as each task completes



Parallel Search

University of St Andrews

- Where we are looking for the existence of some value
  - each individual search can be run in parallel
  - any solution is acceptable

- A Haskell version is

```
parsearch :: [a->Bool] -> [a] -> Bool
parsearch find l =  any (== True) (parmap find l)
```

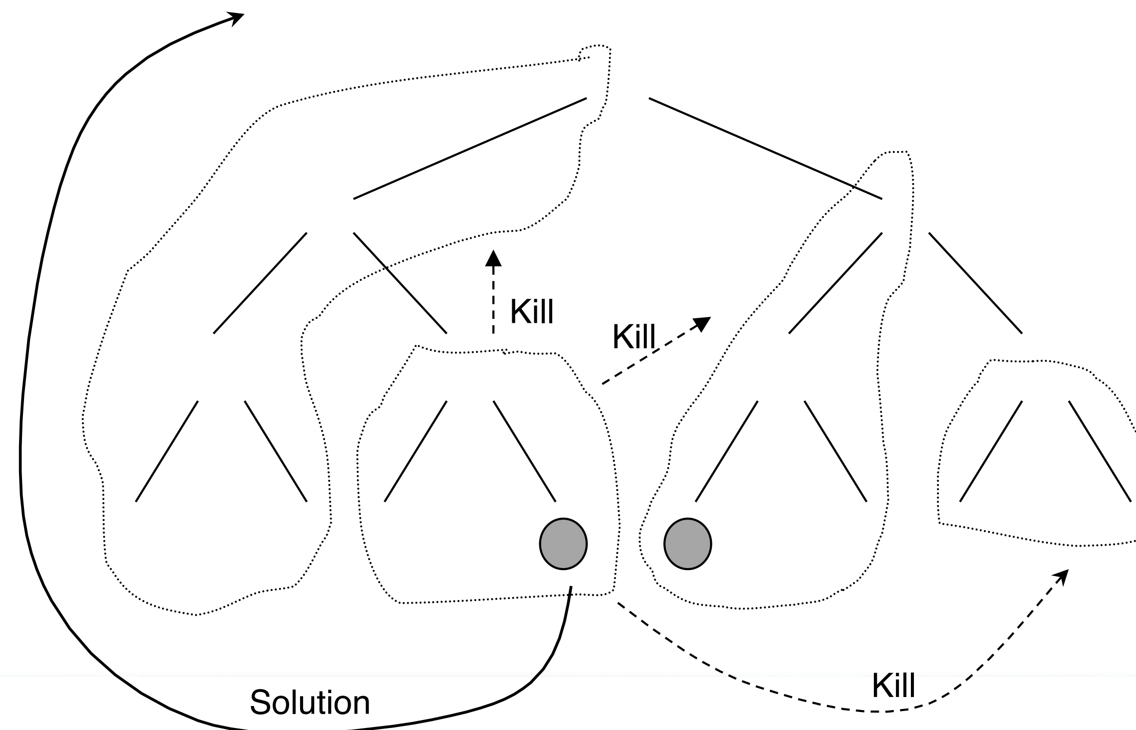
- This uses the Haskell Prelude function *any*

```
any p [] = False
any p (x:xs) = p x || any p xs
```

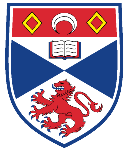
Branch-and-Bound Parallelism

University of St Andrews

- Branch-and-bound terminates search when some result is found
  - running threads must be terminated



Profiling and Performance



Profiling: Basic Runtime Statistics

University of St Andrews

```
$ ./test +RTS -N2 -s
5050
        132,752 bytes allocated in the heap
          6,264 bytes copied during GC
         64,728 bytes maximum residency (1 sample(s))
         25,384 bytes maximum slop
              2 MB total memory in use (0 MB lost due to fragmentation)


..
```

Profiling: Basic Runtime Statistics (2)

University of St Andrews

```
..

                     MUT time (elapsed)       GC time  (elapsed)
  Task  0 (worker) :    0.00s    (  0.00s)       0.00s    (  0.00s)
  Task  1 (worker) :   26.75s    ( 13.46s)       0.00s    (  0.00s)
  Task  2 (worker) :   26.75s    ( 13.46s)       0.00s    (  0.00s)
  Task  3 (bound)  :   26.64s    ( 13.36s)       0.00s    (  0.00s)
  Task  4 (worker) :   26.75s    ( 13.46s)       0.00s    (  0.00s)

  SPARKS: 202 (4 converted, 4 pruned)


...
  Total time   26.75s  ( 13.46s elapsed)
...
  Productivity 100.0% of total user, 198.8% of total elapsed
```

Profiling to Determine Execution Costs

University of St Andrews

- Enable *cost-centre profiling* for all functions using

```
$ ghc -prof -auto-all -rtsopts MyProg.hs
```

- Give the correct runtime flags to enable time profiling

`-pT`	for time
`-hc`	for heap profiling by cost centre

```
http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html
```

Example Cost Centre Profile

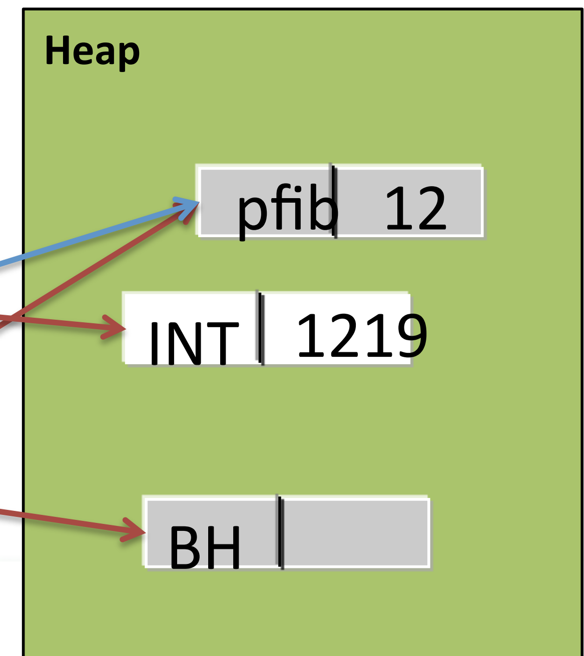
```
Tue Oct 21 09:32 2009
Time and Allocation Profiling Report  (Final)

matMult_prof +RTS -pT -hC -H16M -RTS 50

total time  = 0.24 secs   (12 ticks @ 20 ms)
total alloc = 24,745,124 bytes
                              individual   inherited
COST CENTRE   MODULE   entry %time %alloc %time %alloc

MAIN          MAIN         0    0.0    0.0  100.0 100.0
 main         Main         0   33.3   28.0  100.0  99.5
  listToList  Main       100    0.0    1.2    0.0    1.2
  multMat     Main         1    0.0    0.5   66.7   70.3
   multMatT   Main         1    0.0    0.4   66.7   69.8
    multVec   Main      2500   66.7   69.5   66.7   69.5
 CAF          PrelRead     2    0.0    0.0    0.0    0.0
 CAF          PrelHandle   2  0.0    0.0     0.0    0.0
 CAF          Main        13    0.0    0.5    0.0    0.5
  main        Main         1    0.0    0.0    0.0    0.0
 CAF          System       3    0.0    0.0    0.0    0.0
```

Spark Pruning/Throttling

University of St Andrews

- Not all sparks are turned into threads
  - they may already be normal forms
  - another thread may evaluate them
  - they may not be needed
  - they may be duplicates
  - the spark pool may be full
- Unwanted sparks can be *pruned*
  - this usually happens dur collection

Spark Pool



ThreadScope Profiling

University of St Andrews

- Enable ThreadScope profiling using the –eventlog flag

```
$ ghc -threaded -eventlog -rtsopts --make
MyProg.hs
```

- Run using the –ls flag

```
$ MyProg +RTS -ls -N2
```

- Visualise results using the ThreadScope profiling tool

```
$ threadscope MyProg.eventlog
```

ThreadScope Profiling

University of St Andrews

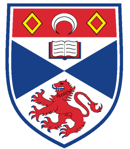
```
$ ghc -threaded -eventlog --make QuickSort.hs
[1 of 1] Compiling Main             ( QuickSort.hs,
QuickSort.o )
Linking QuickSort ...
$ ./QuickSort +RTS -N4 -ls -K50M
QuickSortD size=500000 depth=8
Sum of sort: 25024002430
$ threadscope QuickSort.eventlog ...
$
```

Viewing Event Logs

University of St Andrews

```
$ show-ghc-events Block2.eventlog
Event Types:
   0: Create thread (size 4)
   1: Run thread (size 4)
   2: Stop thread (size 6)
   3: Thread runnable (size 4)
   ...
Events:
   140000: startup: 4 capabilities
   570000: cap 3: creating thread 1
   570000: cap 3: thread 1 is runnable
   573000: cap 3: running thread 1
   680000: cap 3: stopping thread 1 (making a foreign call)
   681000: cap 3: running thread 1
   707000: cap 3: stopping thread 1 (making a foreign call)
   707000: cap 3: running thread 1
   755000: cap 3: creating thread 2
   755000: cap 3: thread 2 is runnable
   760000: cap 3: stopping thread 1 (thread finished)
   ...
```

Example ThreadScope Profile



ThreadScope - TS-QS-4-K50M.eventlog

File View Help

Traces

Timeline Events

Activity Profile

HECs

0.1s 0.2s 0.3s 0.4s 0.5s 0.6s 0.7s 0.8s 0.9s 1s 1.1s 1.2s 1.3s 1.4s 1.5s 1.6s 1.7s 1.8s 1.9s 2s 2.1s 2.2s 2.3s 2.4s

Activity

HEC 0

HEC 1

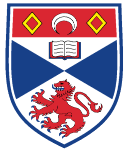
HEC 2

HEC 3

running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

TS-QS-4-K50M.eventlog (193327 events, 2.420s)

Example ThreadScope Profile (Zoomed)



Timeline Events

0.115s 0.12s 0.125s 0.13s 0.135s 0.14s 0.145s

Activity

HEC 0

HEC 1

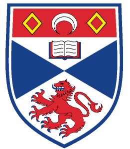
HEC 2

HEC 3

running GC create thread run spark thread runnable seq GC req

g (193327 events, 2.420s)

Example ThreadScope Profile (Zoomed)



ThreadScope - TS-QS-4-K50M.eventlog

File View Help

Traces

Timeline Events

Activity Profile

HECs

2.1s 2.15s 2.2s 2.25s 2.3s 2.35s 2.4s

Activity

HEC 0

HEC 1

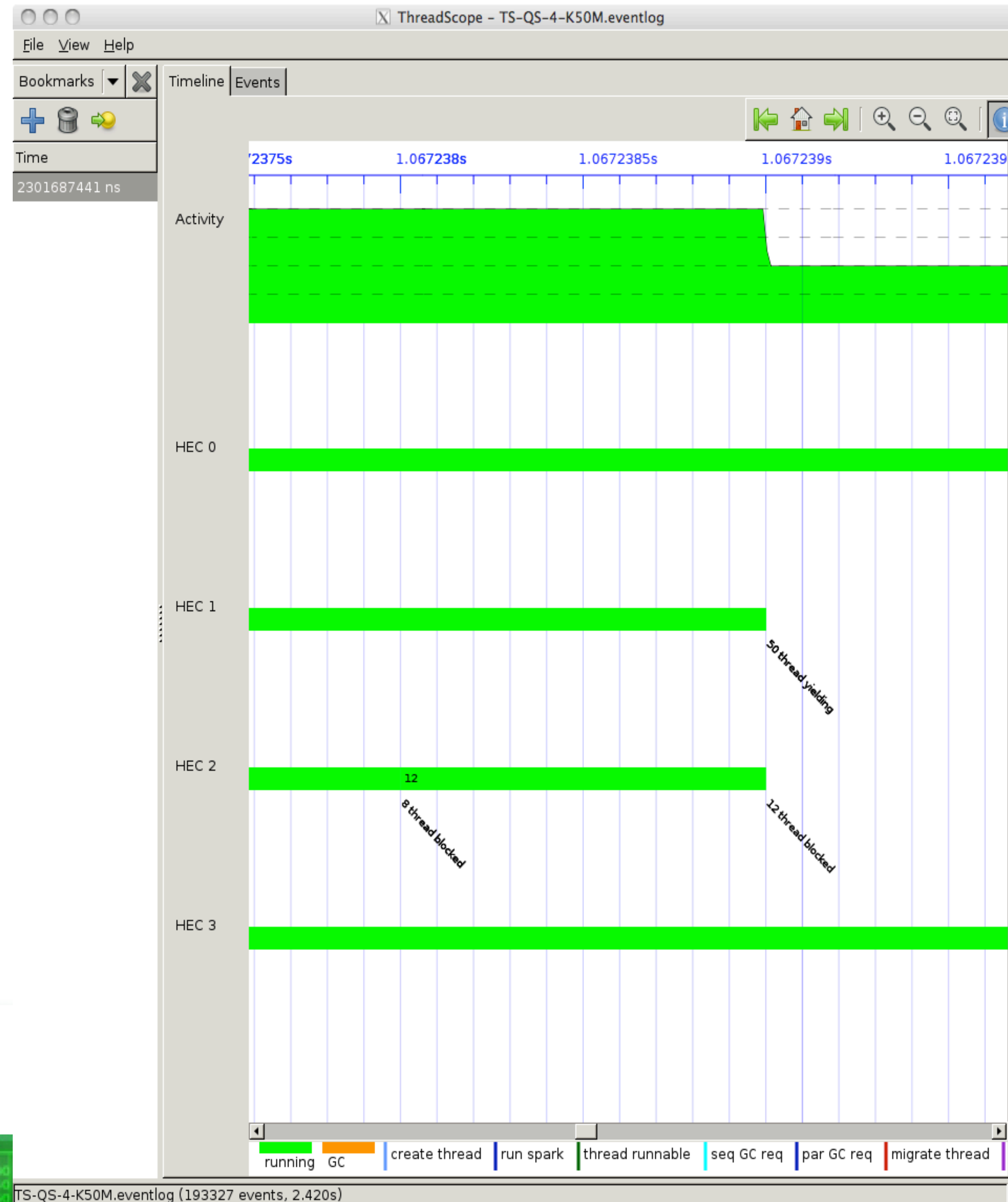
HEC 2

HEC 3

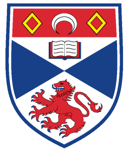
running GC create thread run spark thread runnable seq GC req par GC req migrate thread thread wakeup shutdown

TS-QS-4-K50M.eventlog (193327 events, 2.420s)

Example ThreadScope Profile (More Detail)



Example ThreadScope Profile (More Detail)



HEC 1

50 thread yielding

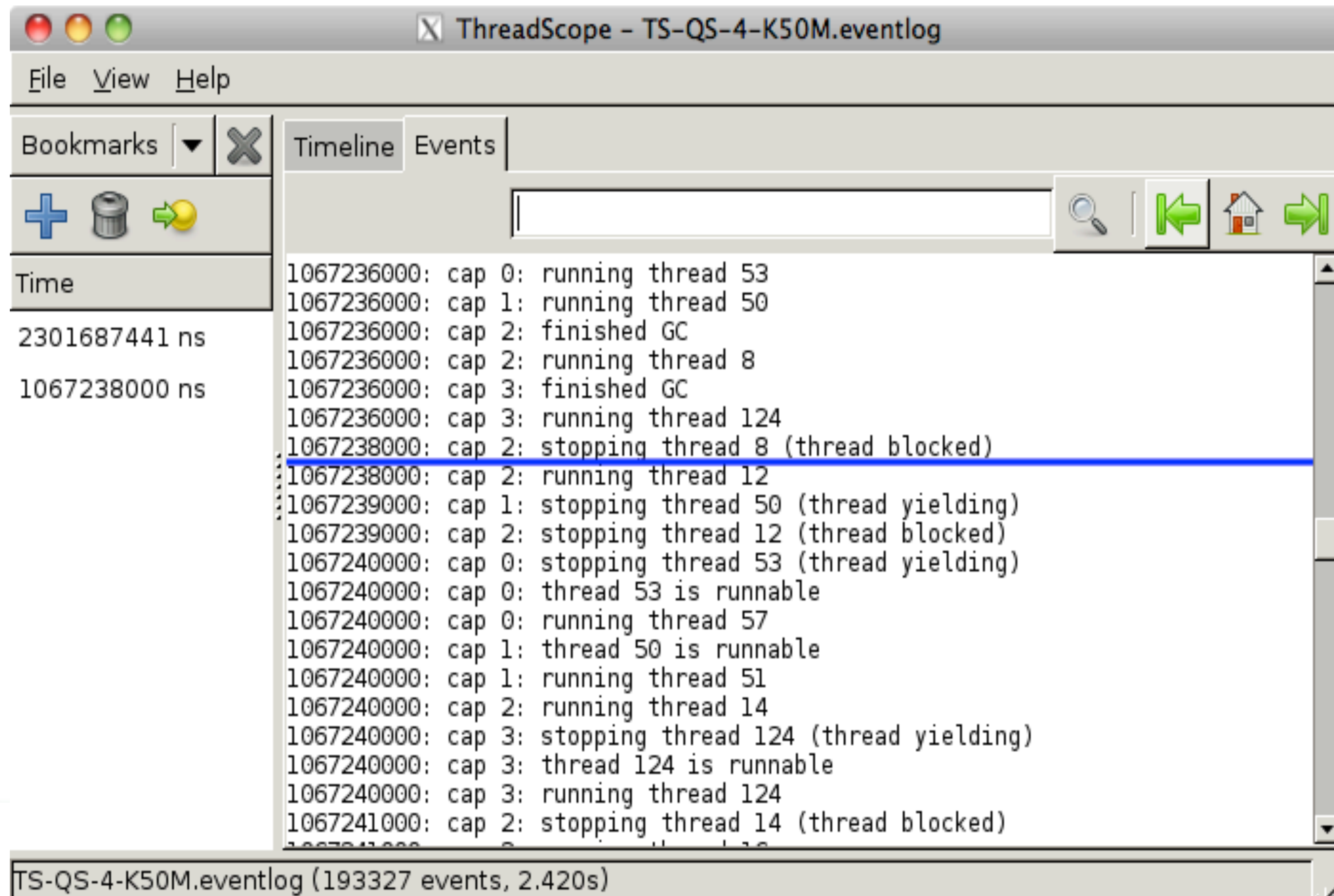
HEC 2

12

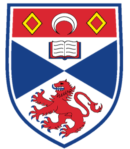
8 thread blocked

12 thread blocked

Example ThreadScope Profile (Viewing Events)



Overall Activity Profile



GrAnSim soda_mg +RTS -bP -bp32 -bM -b-G -by1 -bl400 Average Parallelism = 14.8

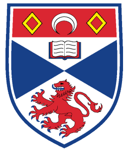
tasks

0 5 10 15 20 25 30 35 40 45

0 50.0 k 100.0 k 150.0 k 200.0 k 250.0 k 300.0 k 350.0 k 400.0 k 450.0 k

running runnable fetching blocked migrating Runtime = 493.2 k cycles

Per-PE Activity Profile

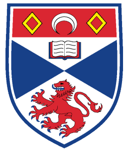


GranSim soda: 32 PEs, 400 cycles latency Average Parallelism = 14.8

PEs: 0–31

0 50.0 k 100.0 k 150.0 k 200.0 k 250.0 k 300.0 k 350.0 k 400.0 k 450.0 k 493251

Per-Thread Profile



GrAnSim soda_mg +RTS -bP -bp32 -bM -b-G -by1 -bl400 Mon Oct 21 22:59 1996

tasks

36
27
24
21
18
15
12
9
6
3
0

0 50.0 k 100.0 k 150.0 k 200.0 k 250.0 k 300.0 k 350.0 k 400.0 k

Worked Example: Parallel Sort

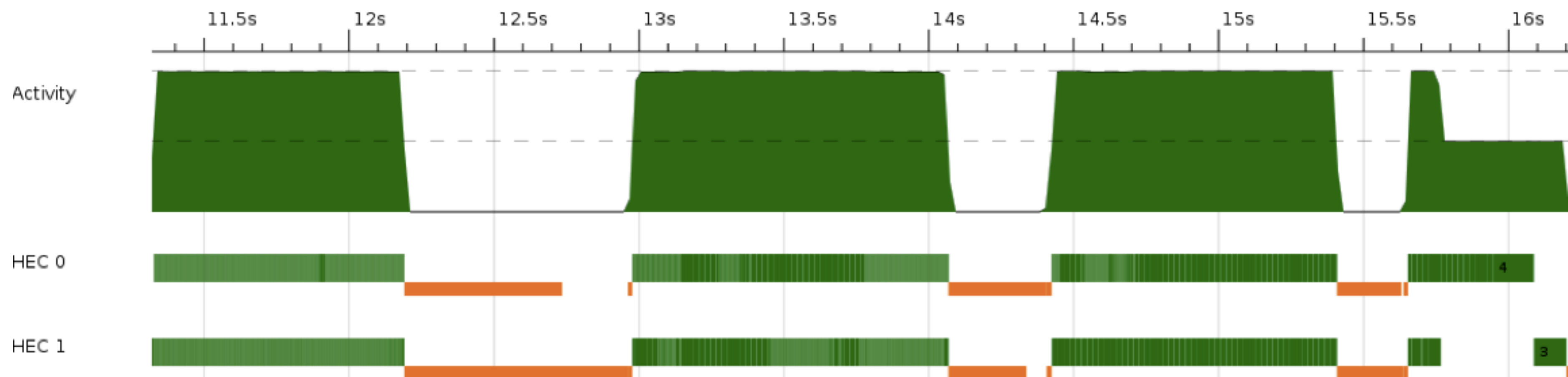
University of St Andrews

$$
\begin{array}{l}
psort :: Int \rightarrow [Int] \rightarrow [Int] \\
psort\ _ \qquad\qquad\quad [\,] = [\,] \\
psort\ parLevel\ l@(x : xs) \\
\quad \mid parLevel > 0 = hiSorted\ `par`\ loSorted\ `pseq`\ (loSorted +\!\!+\ x : hiSorted) \\
\quad \mid otherwise \qquad = seqSort\ l \\
\quad \textbf{where}\ (lo, hi) \quad = partition\ (<x)\ xs \\
\qquad\qquad\quad loSorted = psort\ (parLevel - 1)\ lo \\
\qquad\qquad\quad hiSorted = force\ (psort\ (parLevel - 1)\ hi)
\end{array}
$$

Threadscope Profile of psort

University of St Andrews

- One problem is the long pauses, but why?
- A second problem is the sequential tail



Improving the Program

1. Make *partition* strict
2. Remove *++*

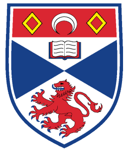
```
qsort :: [Int] → [Int]
qsort xs = seqSort xs []
   where seqSort []       zs = zs
         seqSort (x : xs) zs = seqSort lo (x : seqSort hi zs)
           where (lo, hi) = partition x xs

partition :: Int → [Int] → ([Int], [Int])
partition x xs = go xs [] []
   where go []       ts fs = (ts, fs)
         go (y : ys) ts fs
           | y < x      = go ys (y : ts) fs
           | otherwise  = go ys ts       (y : fs)
```

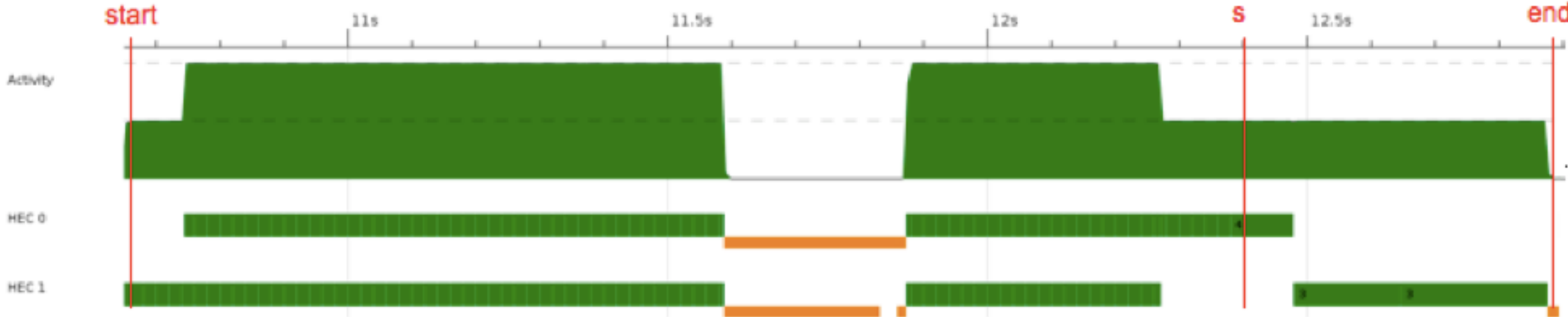
Parallel version

University of St Andrews

$$
\begin{array}{l}
\mathit{psort1}\ n\ xs = go\ n\ xs\ [\,] \\
\quad \textbf{where}\ go\ _\ [\,] \qquad zs = zs \\
\qquad\qquad go\ n\ (x : xs)\ zs \\
\qquad\qquad\quad \mid n > 0 \qquad\quad = r\ \text{`}par\text{`}\ go\ (n-1)\ lo\ (x : r) \\
\qquad\qquad\quad \mid otherwise \ = seqSort\ (x : xs)\ zs \\
\qquad\qquad\ \textbf{where}\ r = force\ (go\ (n-1)\ hi\ zs) \\
\qquad\qquad\qquad\quad (lo, hi) = partition\ x\ xs
\end{array}
$$

Performance Results





No. cores	Speedup
2	1.49
4	0.78
8	0.60

Use Execution Replay and checkpoints

University of St Andrews

```
  188.020 (   93.914) cap 0: partition [10.837.539]
  732.638 (  ...1.668) cap 1:   seqSort [11.382.157]
1.621.573 (  888.935) cap 1:   seqSort [12.271.092]
1.996.394 (  374.821) cap 1:     force [12.645.913]
2.225.849 (  229.455) cap 1:     force [12.875.368]
2.225.852 (        3) cap 1:       end [12.875.371]
```

The problem is that the *force* blocks the result production

The Solution

- Replace *force* with a lazier version
  - returns if already forced

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a) \mid Done\ [a]$$

$$\begin{aligned}
&toList :: List\ a \rightarrow [a] \\
&toList\ Nil = [\,] \\
&toList\ (Cons\ x\ xs) = \textbf{let}\ xs' = toList\ xs \\
&\qquad \textbf{in}\ x\ `seq`\ xs'\ `seq`\ x : xs' \\
&toList\ (Done\ xs) = xs
\end{aligned}$$

New Parallel version

University of St Andrews

$$
\begin{array}{l}
psort2 :: Int \to [Int] \to [Int] \\
psort2\ n\ xs = toList\ (go\ n\ xs\ Nil) \\
\quad \textbf{where}\ go\ _\ [\,]\qquad zs = zs \\
\qquad\qquad go\ n\ (x : xs)\ zs \\
\qquad\qquad\quad \mid n > 0 \qquad\quad = r\ `par`\ go\ (n-1)\ lo\ (Cons\ x\ r) \\
\qquad\qquad\quad \mid otherwise \quad = seqSort\ (x : xs)\ zs \\
\qquad\qquad \textbf{where}\ r = Done\ \$!\ toList\ (go\ (n-1)\ hi\ zs) \\
\qquad\qquad\qquad\quad (lo, hi) = partition\ x\ xs
\end{array}
$$

No. cores	Speedup
2	1.90
4	2.35
8	2.75

Conclusions

- Patterns of parallelism help structure parallel thinking
  - most patterns can be implemented directly using just par/pseq
  - some cannot...
- Choose the right parallel pattern
  - data parallelism gives a lot of parallelism
    - but requires independent tasks
    - and very regular task/data structures
  - task parallelism is more flexible
    - but gives less parallelism
    - can be hard to manage
- Related Lectures
  - Jost Berthold: Skeletons

Further Reading

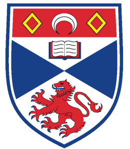
University of St Andrews

Trinder, Hammond, Loidl and Peyton Jones
"Algorithm + Strategy = Parallelism",
Journal of Functional Programming, 8(1):23–60, 1998

Brown. Loidl and Hammond
"ParaForming Forming Parallel Haskell Programs using Novel Refactoring Techniques"
Proc. 2011 Trends in Functional Programming (TFP), Madrid, Spain, 2012

Ferreiro, Castro, Janjic and Hammond
"Repeating History: Execution Replay for Parallel Haskell Programs"
Proc. 2012 Trends in Functional Programming (TFP), St Andrews, June 2012

Marlow
"Parallel and Concurrent Programming in Haskell"
http://community.haskell.org/~simonmar/par-tutorial.pdf, ***May 2012***



Research Directions in Parallel Functional Programming eBook: Kevin Hammond, Greg Michaelson: Amazon.co.uk: Kindle Store

http://www.amazon.co.uk/Research-Directions-Functional-Programming-ebook/dp/B000W67U4G/ref=sr_1_fkmr2_1?ie=UTF8&qid=133914

Click to LOOK INSIDE!

Research Directions in Parallel Functional Programming [Kindle Edition]

LOOK INSIDE! | Kindle Book | Print Book | Zoom – | Zoom + | Feedback | Help | Expanded View | Close

Research Directions in Parallel Functional Pr... (Paperback)
by Kevin Hammond, Greg Michaelson

Paperback **£99.00**

Add to Basket

1 used & new from **£99.00**

Book sections

Front Cover
Copyright
Table of Contents
First Pages
Index
Back Cover
Surprise Me!

Search Inside This Book

GO

Just so you know...
This view is of the print book (Paperback edition). A preview of the Kindle book is currently not available.

Kevin Hammond and Greg Michaelson (Eds)

Research Directions in Parallel Functional Programming

Hide These Books

Your Browsing History

Page 1 of 1

Research Directions...
by Kevin Hammond, Greg...
£72.43

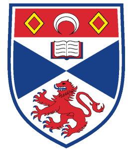
Look Inside This Book

> Edit your book history

Customers Also Bought

Books customers also bought could not be retrieved

Page Numbers Source ISBN: 1852330929



gph-book.pdf – Page 1 of 210

Parallel Haskell: Lightweight Parallelism for Heavyweight Functional Programs

(DRAFT – please do not redistribute without permission.)

Kevin Hammond, Chris Brown and Phil Trinder

In Preparation

Funded by

University of St Andrews

- SCIEnce (EU FP6), Grid/Cloud/Multicore coordination
  - €3.2M, 2005-2012
- Advance (EU FP7), Multicore streaming
  - €2.7M, 2010-2013
- HPC-GAP (EPSRC), Legacy system on thousands of cores
  - £1.6M, 2010-2014
- Islay (EPSRC), Real-time FPGA streaming implementation
  - £1.4M, 2008-2011
- ParaPhrase (EU FP7), Patterns for heterogeneous multicore
  - €2.6M, 2011-2014

EPSRC

SEVENTH FRAMEWORK PROGRAMME

Industrial Connections

University of St Andrews

SAP GmbH, Karlsrühe

BAe Systems

Selex Galileo

BioId GmbH, Stuttgart

Philips Healthcare

Software Competence Centre, Hagenberg

Mellanox Inc.

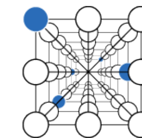
Erlang Solutions Ltd

Microsoft Research

Well-Typed

Funded by

- **ParaPhrase (EU FP7), Patterns for heterogeneous multicore,** €2.6M, 2011-2014
- **SCIEnce (EU FP6), Grid/Cloud/Multicore coordination**
  - €3.2M, 2005-2012
- **Advance (EU FP7), Multicore streaming**
  - €2.7M, 2010-2013
- **HPC-GAP (EPSRC), Legacy system on thousands of cores**
  - £1.6M, 2010-2014
- **Islay (EPSRC), Real-time FPGA streaming implementation**
  - £1.4M, 2008-2011
- **TACLE: European Cost Action on Timing Analysis**
  - €300K, 2012-2015


ADVANCE
StatArch

SEAS DTC

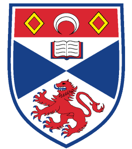
Industrial Connections

University of St Andrews

- Mellanox Inc.
- Erlang Solutions Ltd
- SAP GmbH, Karlsrühe
- BAe Systems
- Selex Galileo
- BioId GmbH, Stuttgart
- Philips Healthcare
- Software Competence Centre, Hagenberg
- Microsoft Research
- Well-Typed LLC

Microsoft Research

BAE SYSTEMS



THANK YOU!

http://www.paraphrase-ict.eu

http://www.project-advance.eu

@paraphrase_fp7