University of St Andrews

SEVENTH FRAMEWORK PROGRAMME

Parallel Functional Programming

Kevin Hammond

University of St Andrews, Scotland

http://www.paraphrase-ict.eu

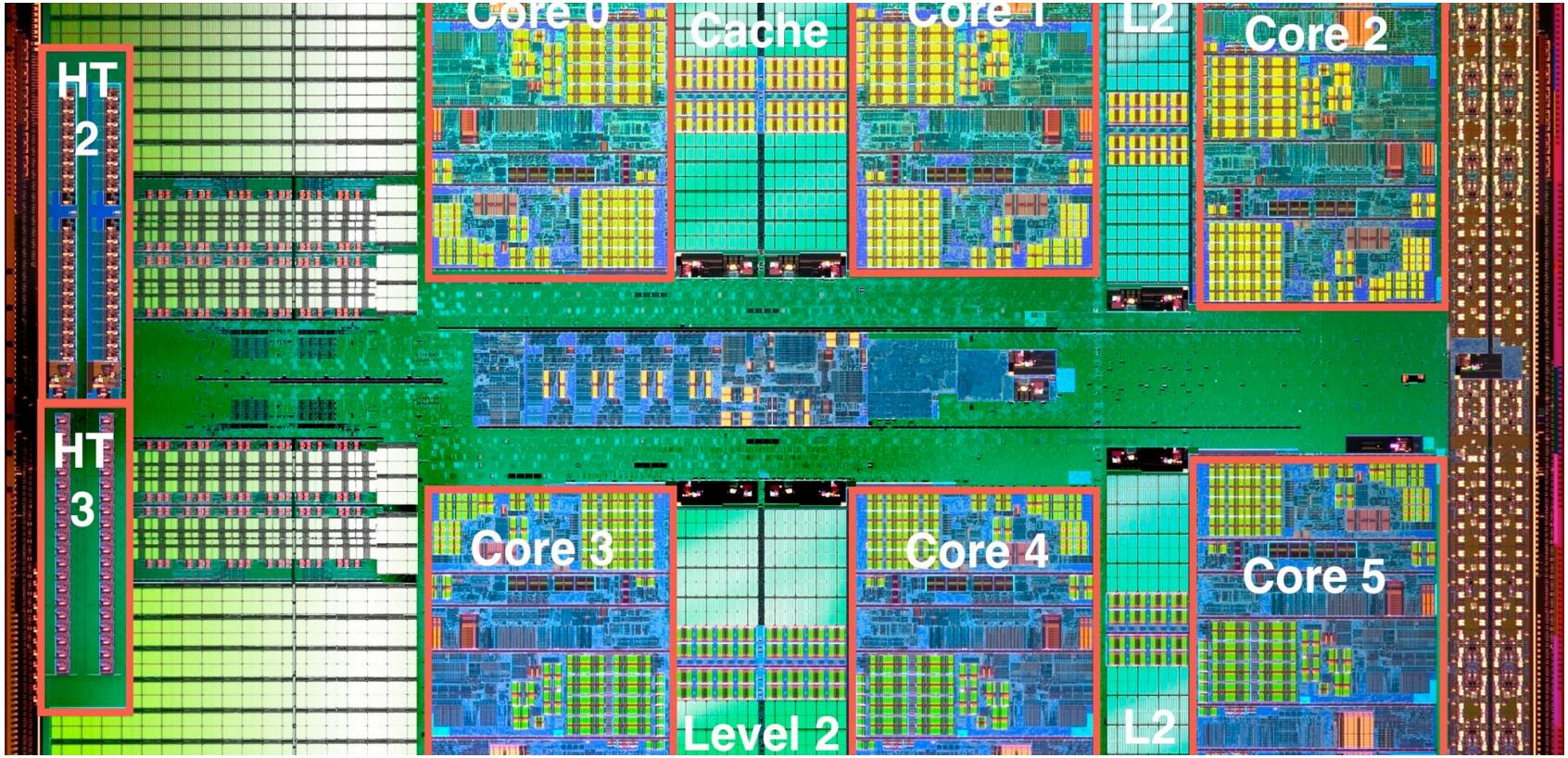
http://www.project-advance.eu

@paraphrase_fp7

sicsa*

The Dawn of the Multicore Age

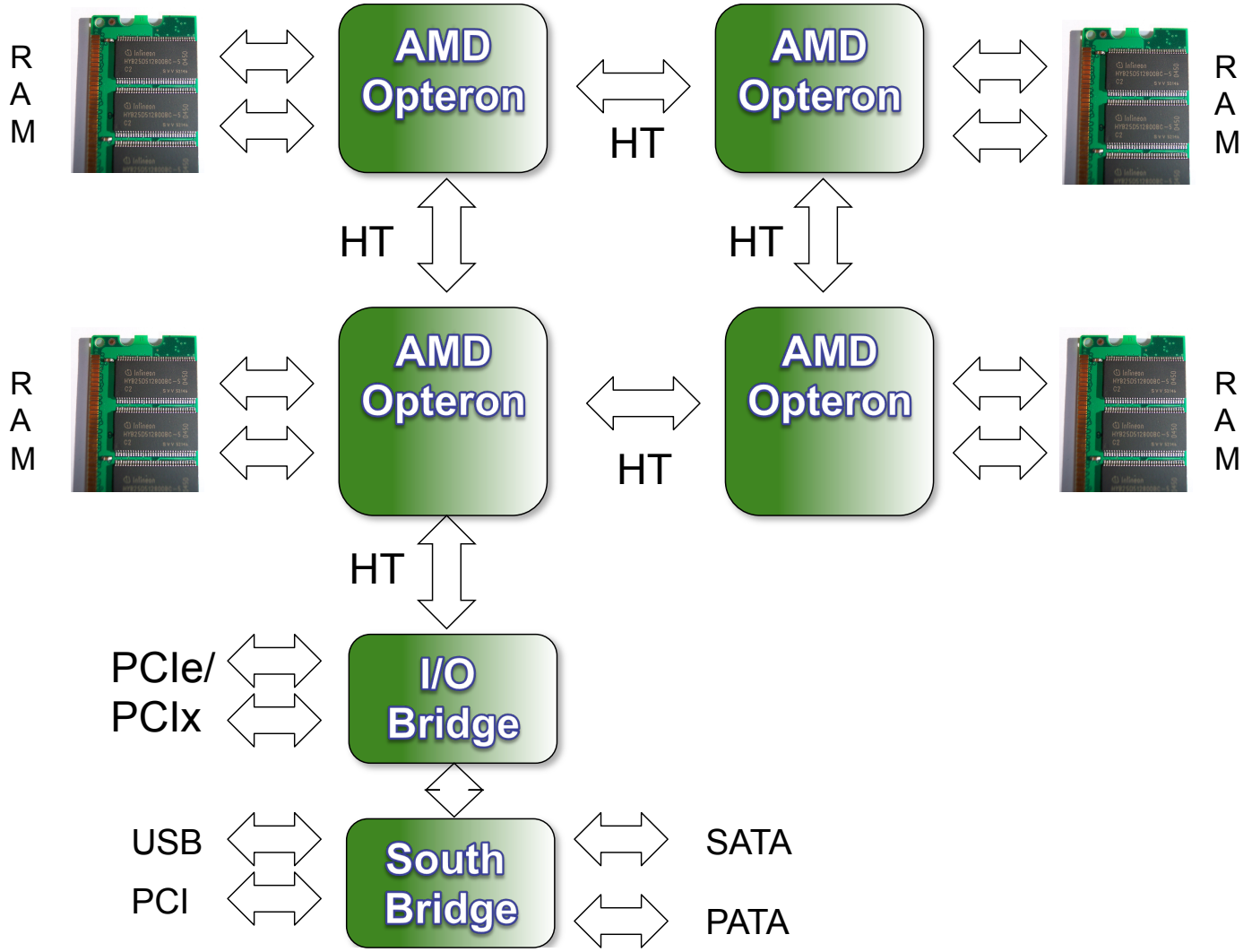
University of St Andrews



AMD Opteron Magny-Cours , 6-Core (source: wikipedia)

The Near Future: Scaling toward Manycore

University of St Andrews



The Challenge

University of St Andrews

"Ultimately, developers should start thinking about *tens, hundreds, and thousands* of cores *now* in their algorithmic development and deployment pipeline."

Anwar Ghuloum, Principal Engineer, Intel Microprocessor Technology Lab

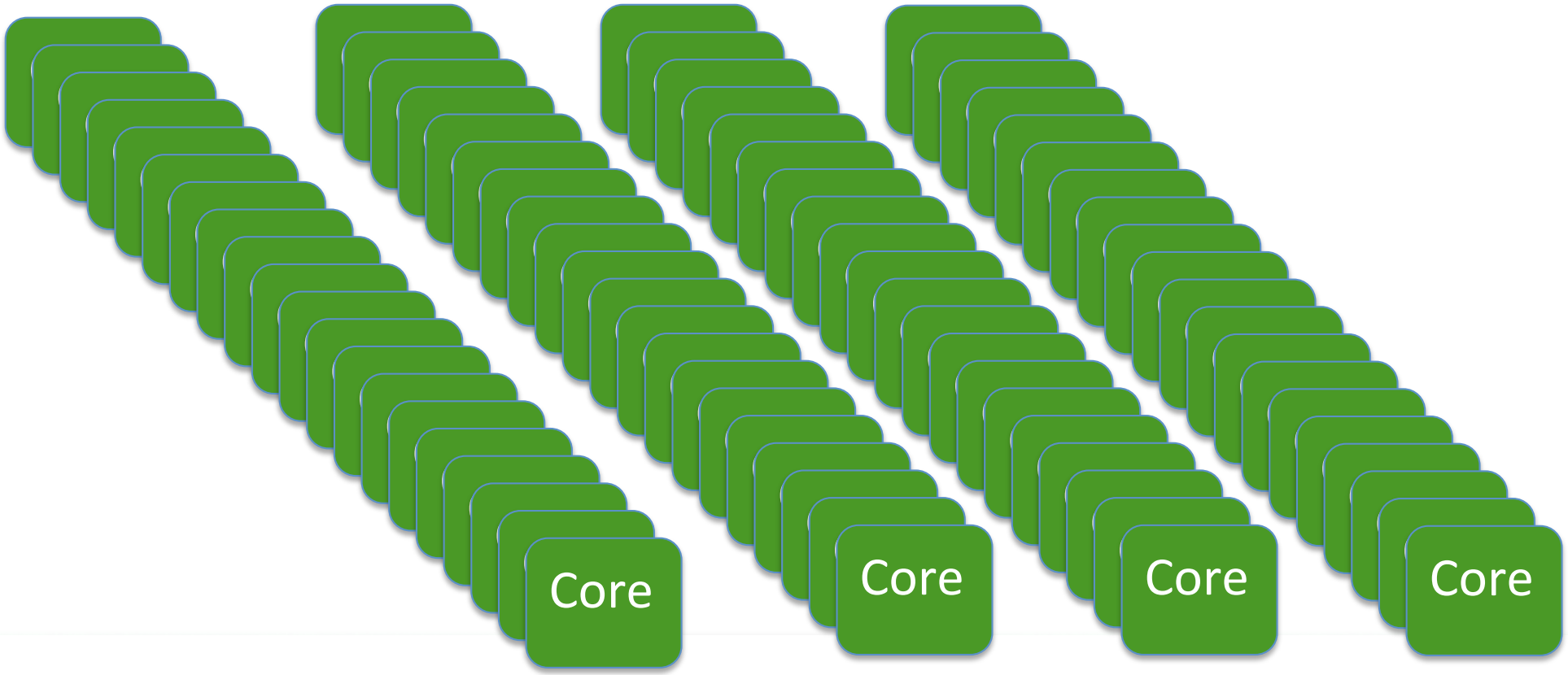
"The dilemma is that a *large percentage* of mission-critical enterprise applications will not ``automagically'' run faster on multi-core servers. *In fact, many will actually run slower.* We must make it as easy as possible for applications programmers to exploit the latest developments in multi-core/many-core architectures, while still making it easy to target future (and perhaps unanticipated) hardware developments."

Patrick Leonard, Vice President for Product Development
Rogue Wave Software

The Future: “megacore” computers?

University of St Andrews

- ***Hundreds of thousands, or millions, of cores***



What will "megacore" computers look like?

University of St Andrews

- Probably *not* just scaled versions of today's multicore
  - Perhaps hundreds of dedicated lightweight integer units
  - Hundreds of floating point units (enhanced GPU designs)
  - A *few* heavyweight general-purpose cores
  - Some specialised units for graphics, authentication, network etc
  - possibly *soft* cores (FPGAs etc)
  - *Highly heterogeneous*

- Probably *not* uniform shared memory
  - NUMA is likely, even hardware distributed shared memory
  - or even message-passing systems on a chip
  - *shared-memory will not be a good abstraction*

Laki (NEC Nehalem Cluster) and hermit (XE6)

Laki

- 700 dual socket Xeon 5560 2,8GHz (“Gainestown”)
- 12 GB DDR3 RAM / node
- Infiniband (QDR)
- 32 nodes with additional Nvidia Tesla S1070
- Scientific Linux 6.0

hermit (phase 1 step 1)

- 38 racks with 96 nodes each
- 96 service nodes and 3552 compute nodes
- Each compute node will have 2 sockets AMD Interlagos @ 2.3GHz 16 Cores each leading to 113.664 cores
- Nodes with 32GB and 64GB memory reflecting different user needs
- 2.7PB storage capacity @ 150GB/s IO bandwidth
- External Access Nodes, Pre- & Postprocessing Nodes, Remote Visualization Nodes

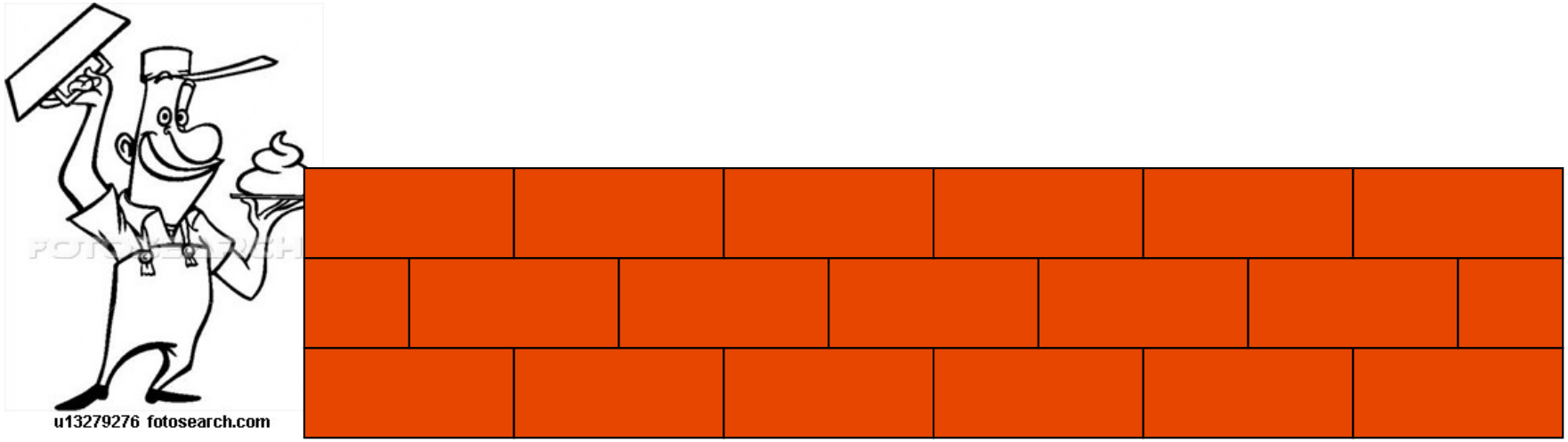
All future programming will be parallel

University of St Andrews

- No future system will be single-core
  - you must use parallel programming
- It's not just about performance
  - it's also about energy usage
- If we don't solve the multicore challenge, then all other CS advances won't matter!
  - user interfaces
  - cyber-physical systems
  - robotics
  - games
  - ...

How to build a wall

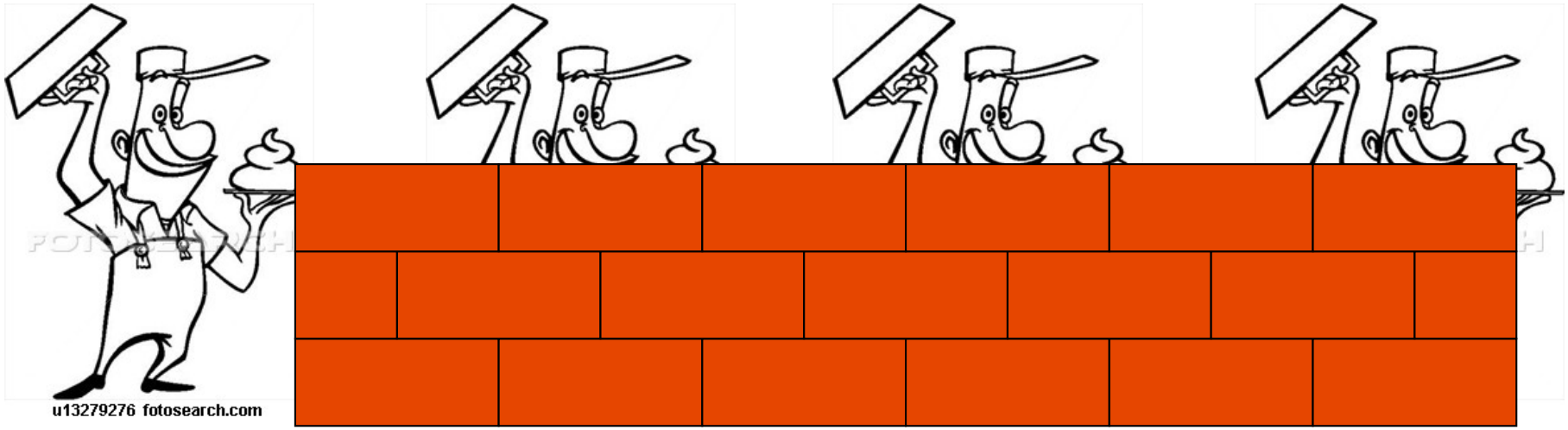
University of St Andrews



(with apologies to Ian Watson, Univ. Manchester)

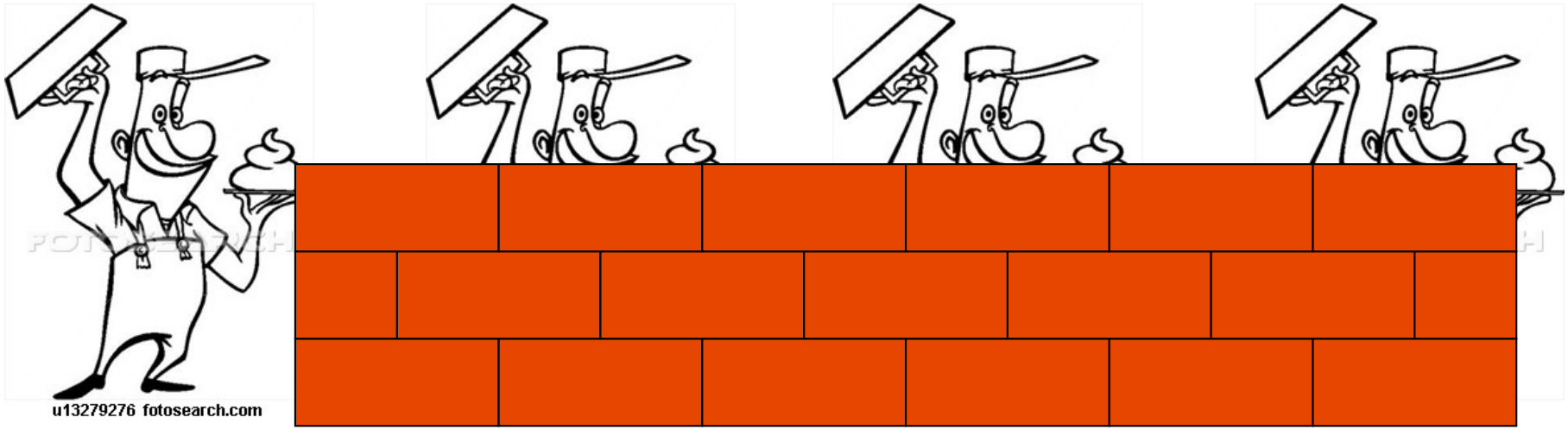
How to build a wall faster

University of St Andrews



How NOT to build a wall

University of St Andrews

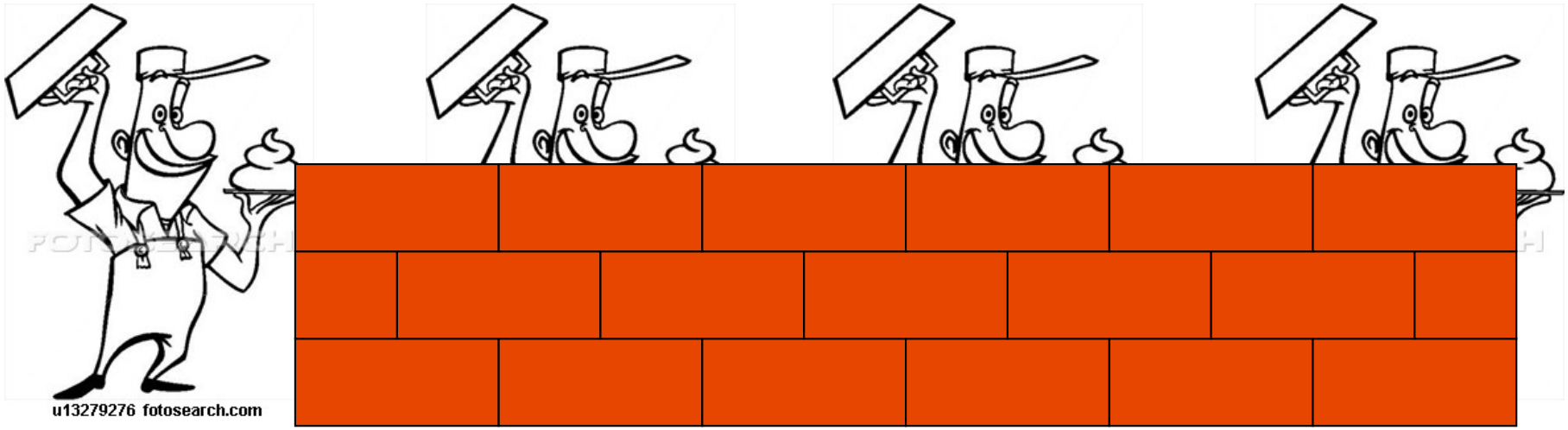


Task identification is not the only problem…

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

Scheduling

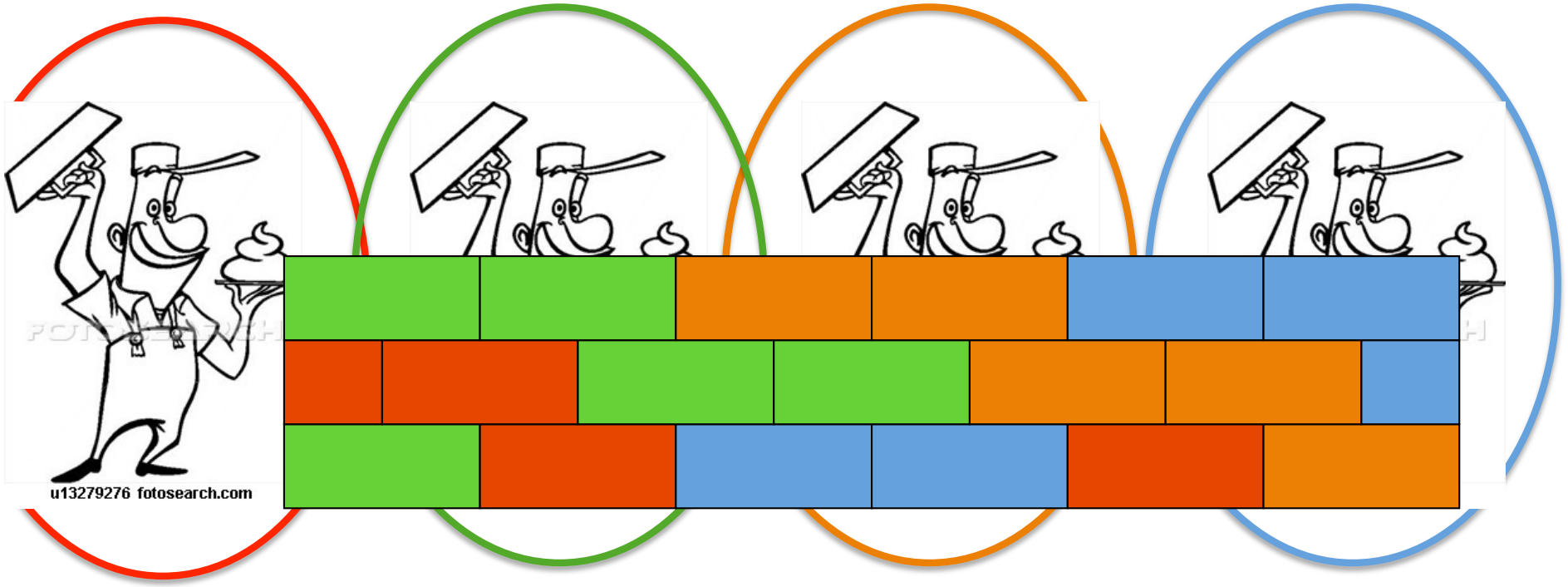
University of St Andrews



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

Placement

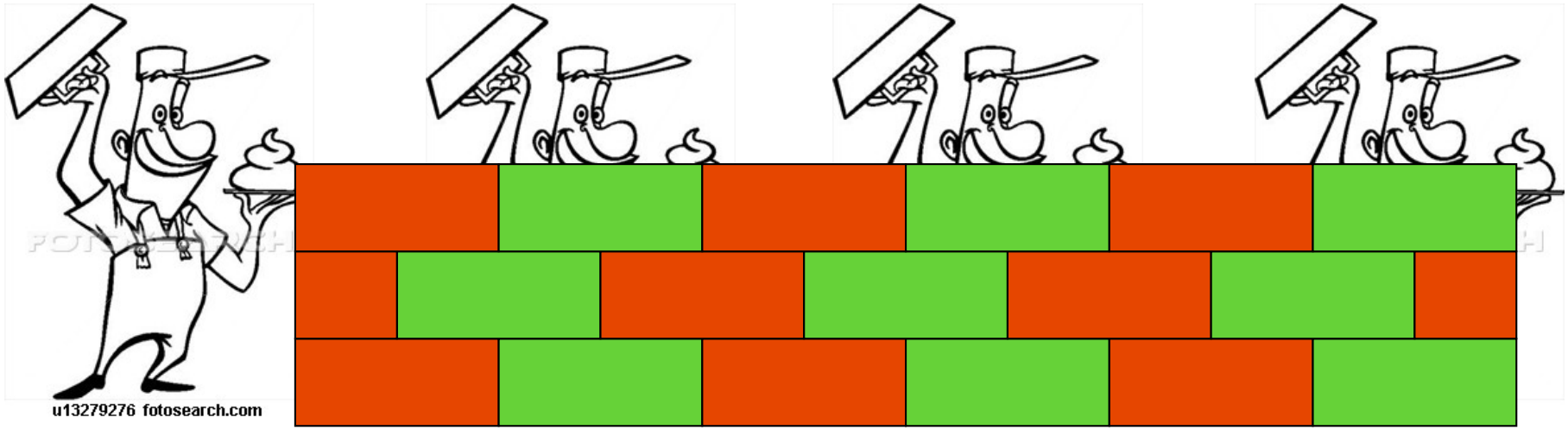
University of St Andrews



Which builder places which brick?

Coordination

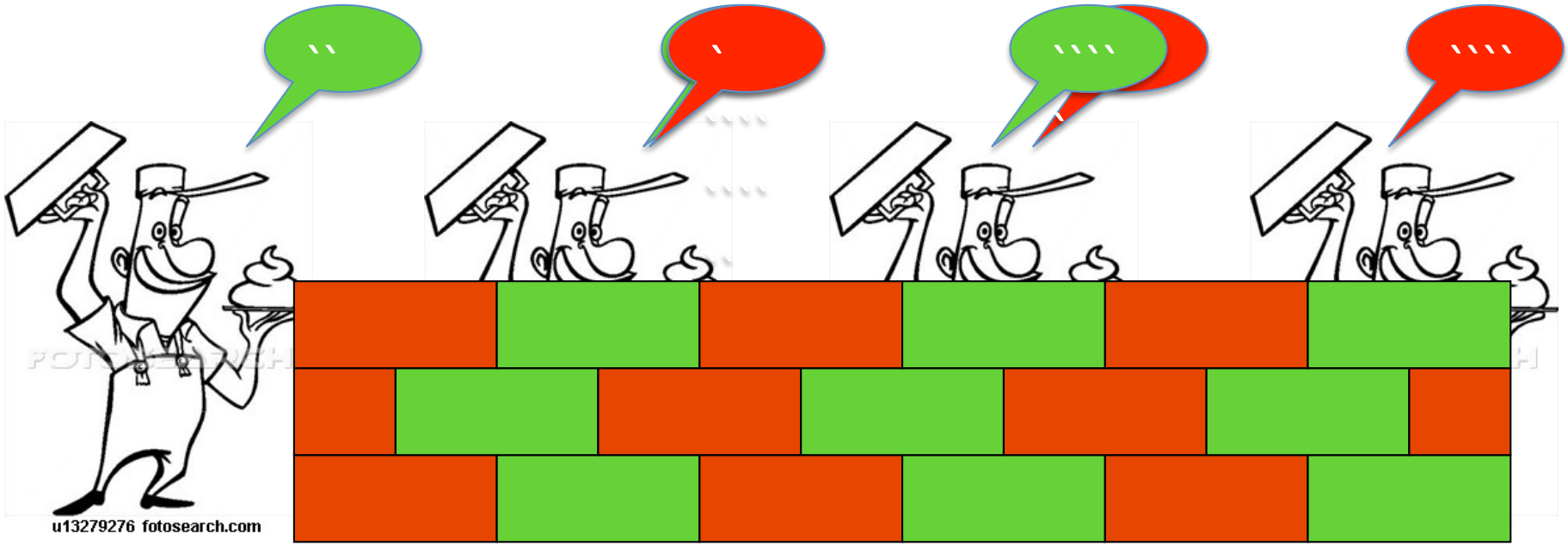
University of St Andrews



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

Thinking in Parallel

University of St Andrews

- Fundamentally, programmers must learn to "think parallel"
  - this requires new *high-level* programming constructs
  - you cannot program effectively while worrying about deadlocks etc
    - they must be eliminated from the design!
  - you cannot program effectively while fiddling with communication etc
    - this needs to be packaged/abstracted!

- Concurrency primitives are NOT the answer!

A Solution?

University of St Andrews

"The only thing that works for parallelism is functional programming"

Bob Harper, Carnegie Mellon

Parallel Functional Programming

University of St Andrews

- No explicit ordering of expressions
- Purity means no side-effects
  - Impossible for parallel processes to interfere with each other
  - Can debug sequentially but run in parallel
  - **Enormous** saving in effort
- Programmer concentrate on solving the problem
  - Not porting a sequential algorithm into a (ill-defined) parallel domain
- **No locks, deadlocks or race conditions!!**

Glorious Parallel Haskell (GpH)

GpH aims for an approach where the runtime system manages parallelism which is introduced by the parallel programmer

- Based on the functional language Haskell
- Simple parallel language extension
- Automatic control of parallelism

```
http://hackage.haskell.org/platform/
```

The *par* Annotation

University of St Andrews

The basic way to introduce parallelism in GpH. A variant of "lazy thread creation" (Peyton Jones & Salkild, 1989):

```
a `par` b
```

create a spark for a and return the value of b

For example

```
x `par` f x where x = …
```

sparks x, and returns f x

Simple Example: Parallel Fibonacci

University of St Andrews

```
import Control.Parallel — for par

pfib :: Int -> Int
pfib n
   | n <= 1     = 1
   | otherwise = n2 `par` (n1 `par` (n1+n2+1))
     where
       n1 = pfib (n-1)
       n2 = pfib (n-2)

-- sequ
fib ::
fib n
   | n <
   | otl
```

Not a real problem, but easy to understand!

This includes *all* locking, placement/mapping, synchronisation, communication, lazy thread creation, data/task serialisation, load balancing, thread migration, scheduling, parallel garbage collection, remote caching, and other memory management

Compiling and Running Parallel Haskell

- Use the `-threaded` flag to GHC to enable parallelism

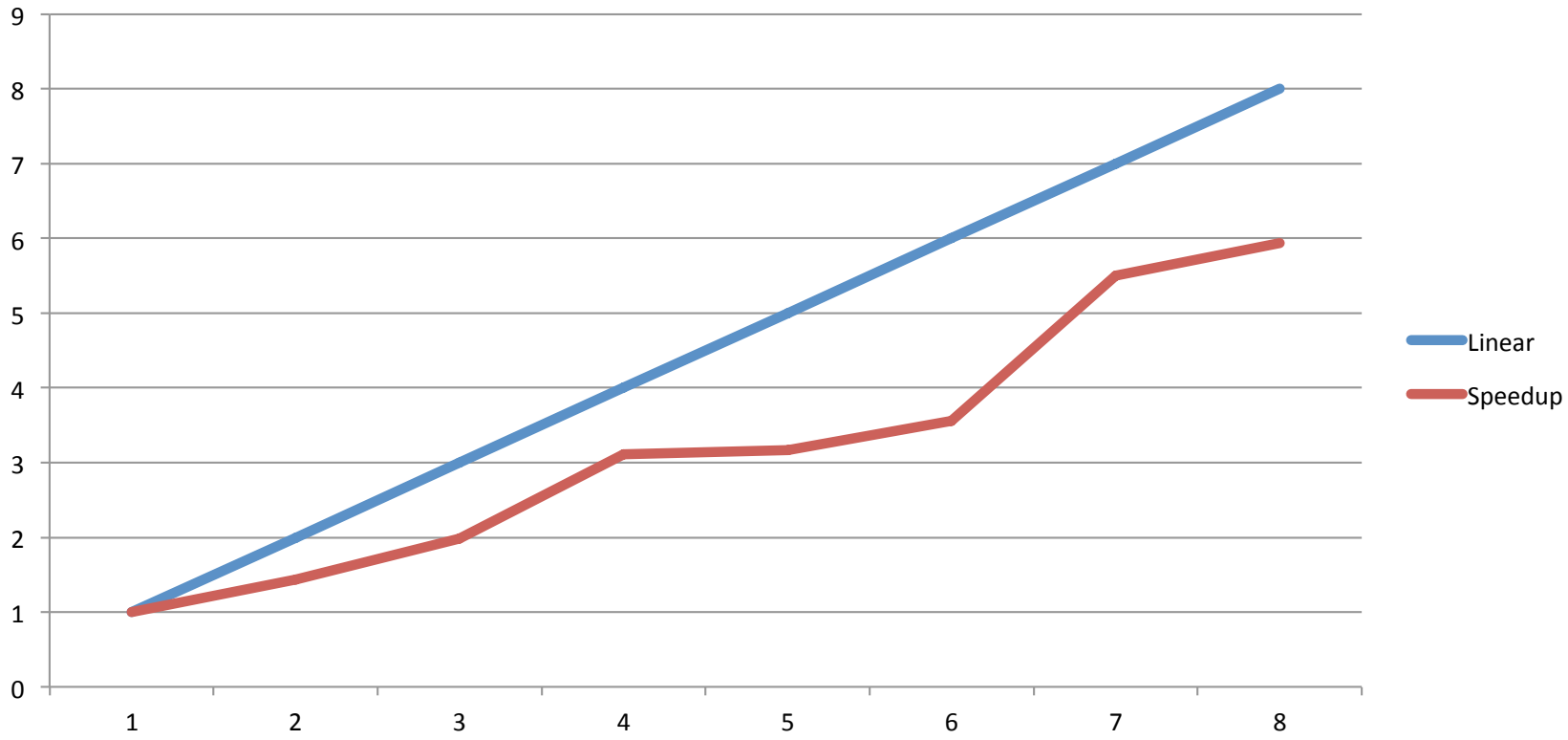
```
$ ghc -threaded -rtsopts --make MyProg.hs
```

- Run the program using the –N option to specify the number of cores

```
$ MyProg +RTS –N16
1973
$
```

Speedups for Fibonacci

University of St Andrews



Another 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]
```

Sorting Example (2)

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

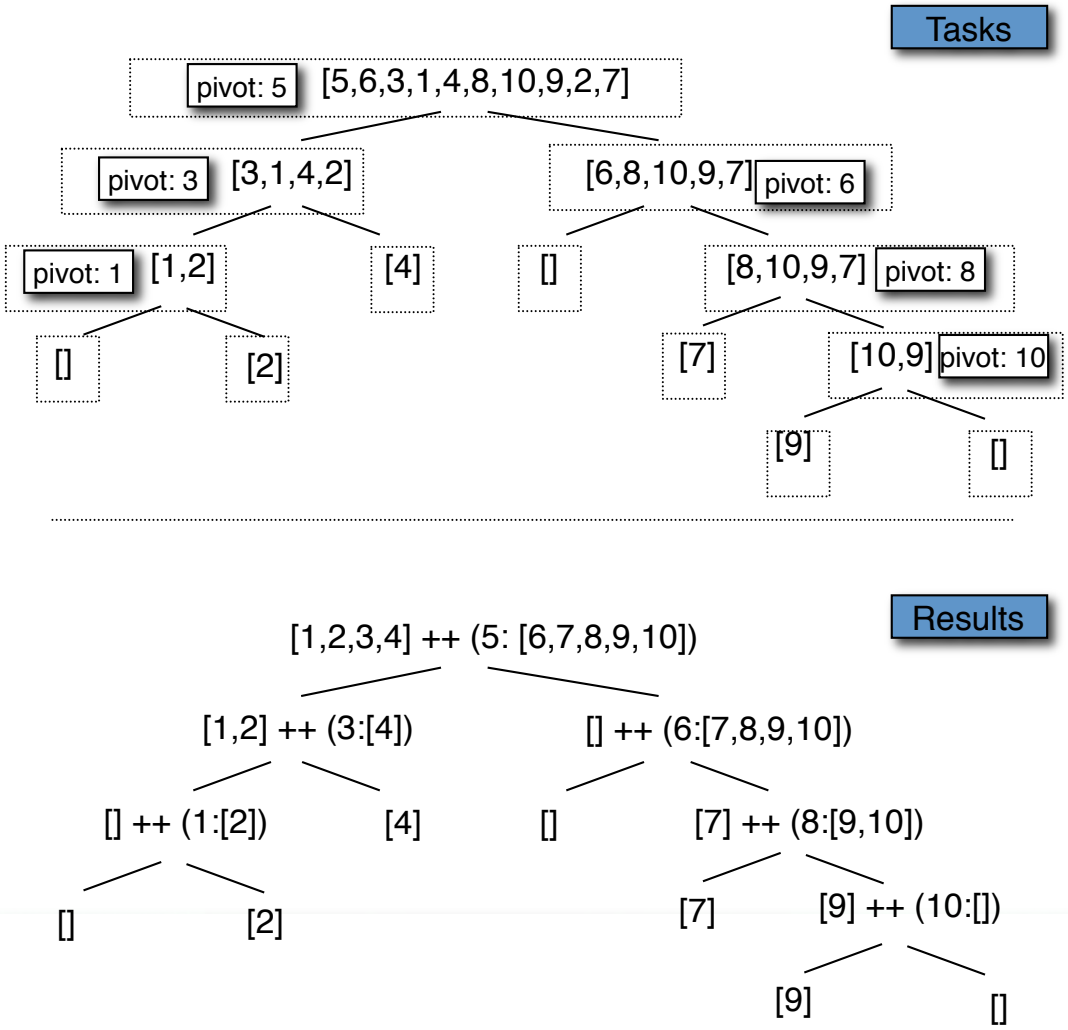
  losort `par` hisort `par` ...

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

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

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

University of St Andrews



There's a well-known pattern here

- Divide-and-Conquer (source: wikipedia)

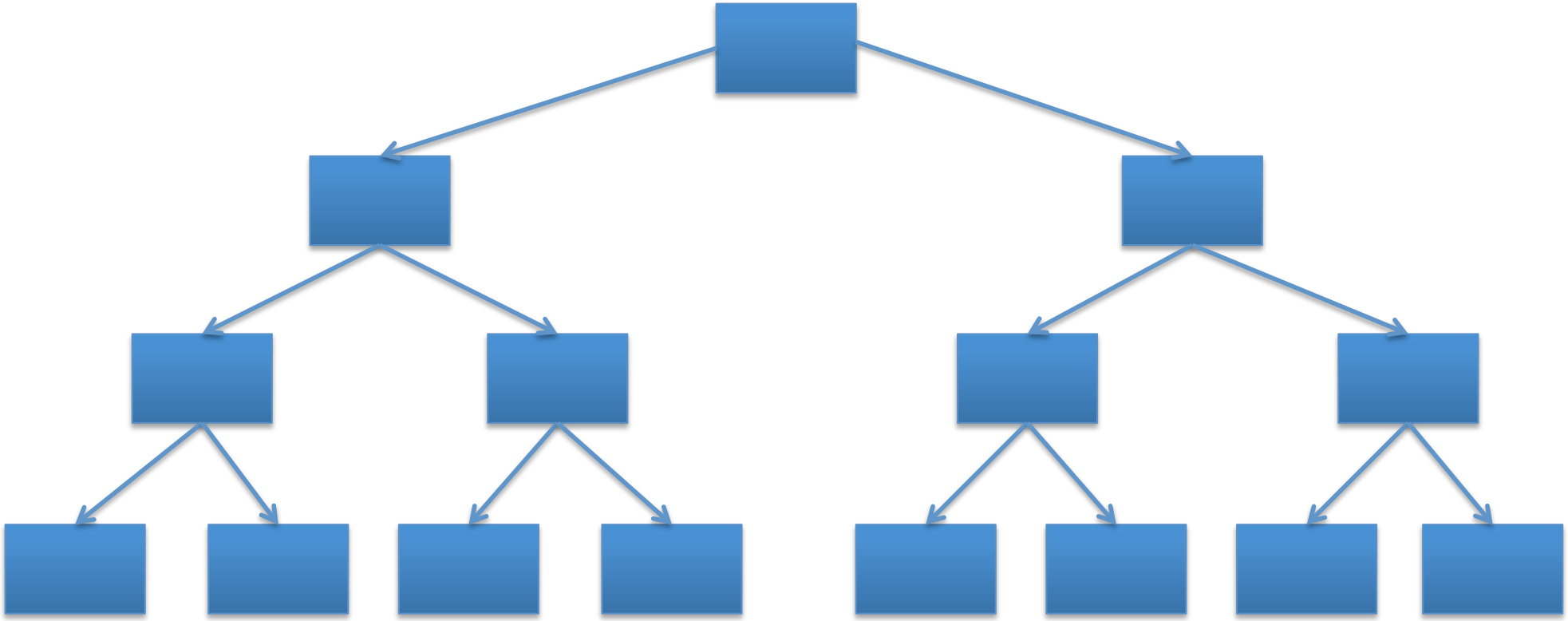
Divide-and-Conquer

University of St Andrews

- *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

Binary Divide and Conquer

University of St Andrews



Advantages of DC Parallelism

University of St Andrews

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

Parallel Quicksort Revisited

University of St Andrews

- We parallelised quicksort 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]
```

- This is an example of divide-and-conquer parallelism

A general divide-and-conquer Function

University of St Andrews

- We can write a *general* function to capture divide-and-conquer

dc split threshold combine worker input = combine results

where results =

if threshold input then

worker input

else *parmap*

(dc split threshold combine worker)

(split input)

```
dc :: (a -> [a]) ->          -- split
      (a -> Bool) ->         -- threshold
      ([b] -> b) ->          -- combine
      (a -> [b]) ->          -- worker
      a -> b
```

Skeletons

University of St Andrews

- *dc* is an example of an *algorithmic skeleton*
  - an *implementation* of a parallel pattern

- Skeletons are templates
  - pluggable *higher-order* functions
  - can be instantiated with concrete worker functions

Murray Cole,
"Algorithmic Skeletons: structured management of parallel computation" MIT Press, 1989

Horacio González-Vélez andMario Leyton:
"A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers."
Softw., Pract. Exper. 40(12): 1135-1160 (2010)

Instantiating the DC Skeleton

University of St Andrews

- dc requires four arguments
  - one *split* function – how to decompose the input
  - one *threshold* function – when is the input simple enough
  - one *combine* function – how to combine the results
  - one *worker* function – what are we actually trying to do

```
dc split threshold combine worker
```

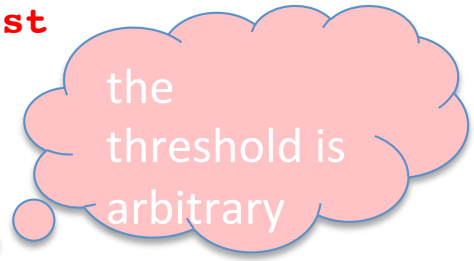
- all of these are function arguments to *dc*

A General Parallel Quicksort

University of St Andrews

- We can use this to define quicksort

```
qsortDC [] = []
qsortDC list = dc split threshold combine worker list
        where    worker = qsort

                 combine = concat

                 threshold xs = length xs < 1

                 split (x:xs) = [lows, [x], highs]
                        where    lows =  [ y | y <- list, y < x]
                                 highs = [ y | y <- list, y >= x ]
```



A General Parallel Fibonacci

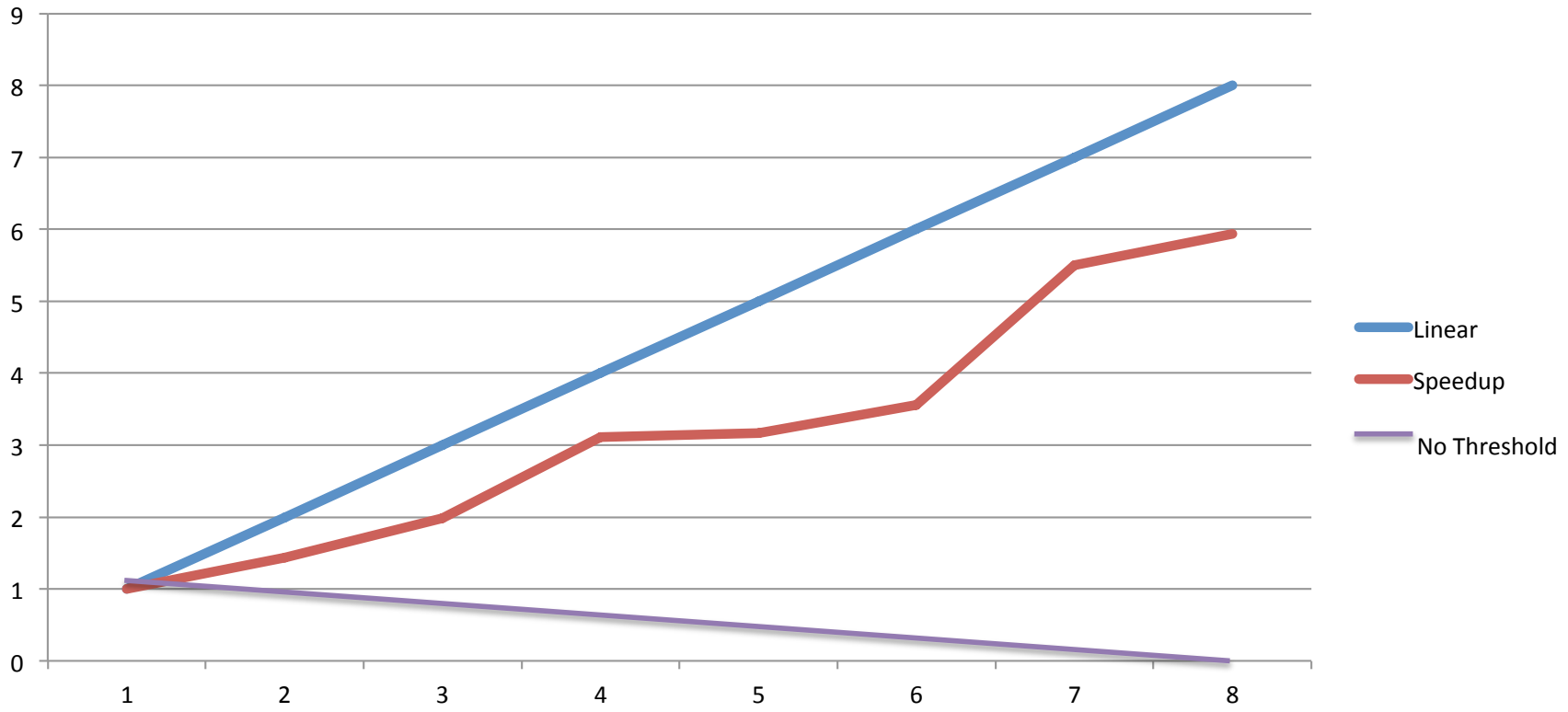
University of St Andrews

- We can also use *dc* to define Fibonacci

```
fibDC n = dc split threshold combine worker list
          where    worker = fib

                   combine = foldr (+) 1

                   threshold n = n <= 20

                   split n = [n-1,n-2]
```

Speedups for Fibonacci (again)

University of St Andrews



Parallelism is not Concurrency

University of St Andrews

- Concurrency is a programming abstraction
  - The *illusion* of independent threads of execution
- Parallelism is a hardware artefact
  - The *reality* of threads executing at the same time
- Concurrency is about breaking a program down into independent units of computation
- Parallelism is about making things happen at the same time
- Typically a parallel program has thousands or millions of threads
- A concurrent program typically has a few threads

Parallelism is not Concurrency (2)

University of St Andrews

- A concurrent thread may be broken down into many parallel threads
  - or none at all
- Parallelism can sometimes be modelled by concurrency
  - but implicit parallelism cannot!
- Concurrency is about maintaining dependencies
  - Parallelism is about breaking dependencies
- If we try to deal with parallelism using concurrency models/ primitives, we are using the wrong abstractions
  - Too low-level, Too coarse-grained,Not scalable

Conclusions

University of St Andrews

- Functional programming makes it easy to introduce parallelism
  - No side effects means any computation could be parallel
  - Matches pattern-based parallelism
  - Much detail can be abstracted
- Lots of problems can be avoided
  - e.g. Freedom from Deadlock
  - Parallel programs give the same results as sequential ones!
- *But not completely trivial!!*
  - Need to choose granularity carefully!
    - e.g. thresholding
  - May need to understand the execution model
    - e.g. pseq

Isn’t this all just wishful thinking?

University of St Andrews

NO!

University of St Andrews

- C++11 has lambda
- Java 8 will have lambda (closures)
- Apple uses closures in Grand Central Dispatch

Some Research Challenges

University of St Andrews

- How do we model parallelism formally
  - when is one program "better" than other
  - Can we prove this???
- How do we deal with the "megacore challenge"
  - scaling, heterogeneity, multiple levels
- What are the best abstractions for parallelism
  - skeletons (what skeletons?), strategies, ...
  - How do we help the programmer "think in parallel"
  - What do we do if a pattern doesn't quite fit the problem
- How do we understand performance
  - visualisation, abstraction, formal reasoning, ...
- How can we analyse resource usage in parallel systems
  - Time, energy, ...
- What about tool support (e.g. refactoring)
- Can we do it all automatically??

Further Reading

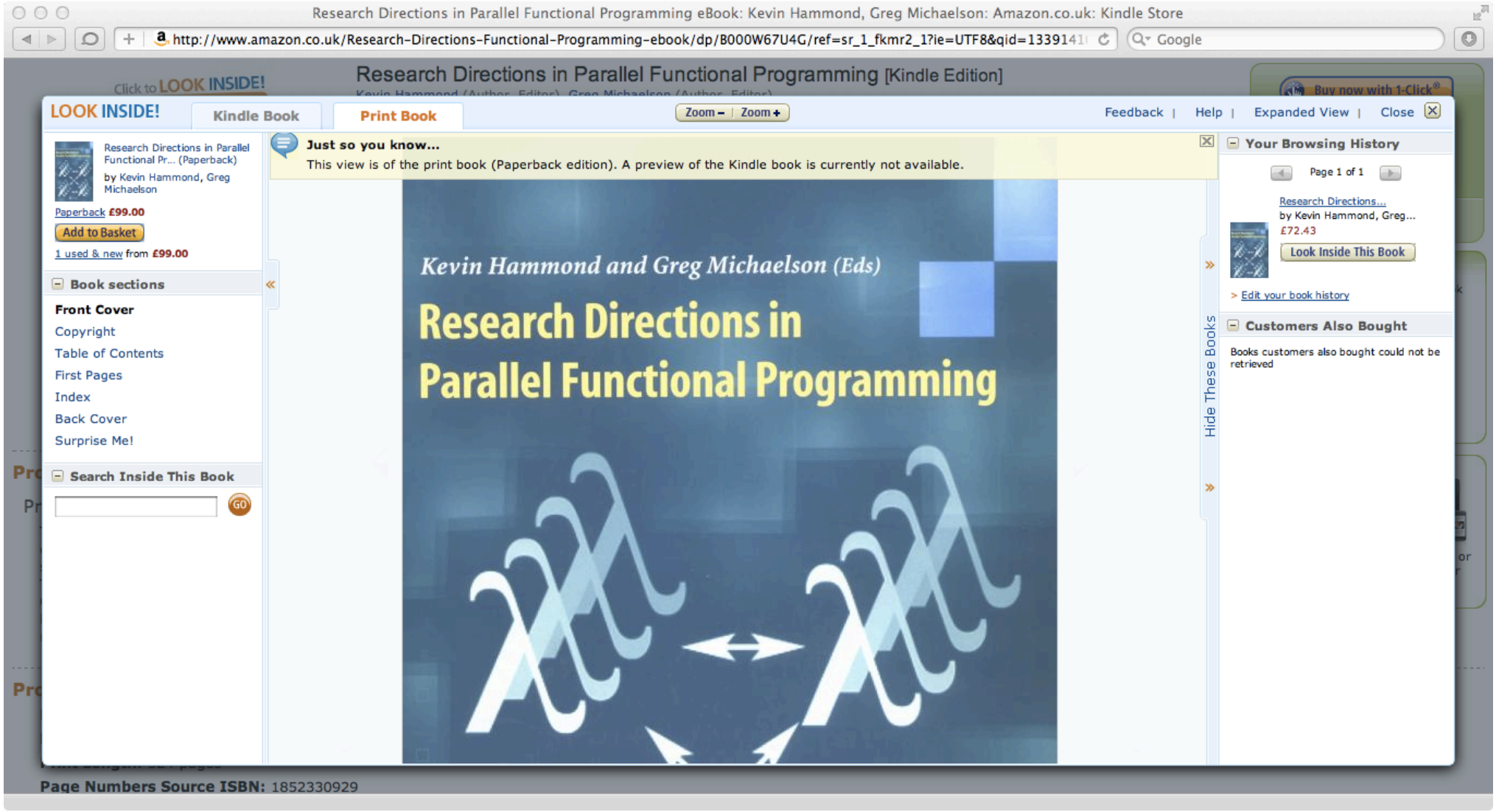
University of St Andrews

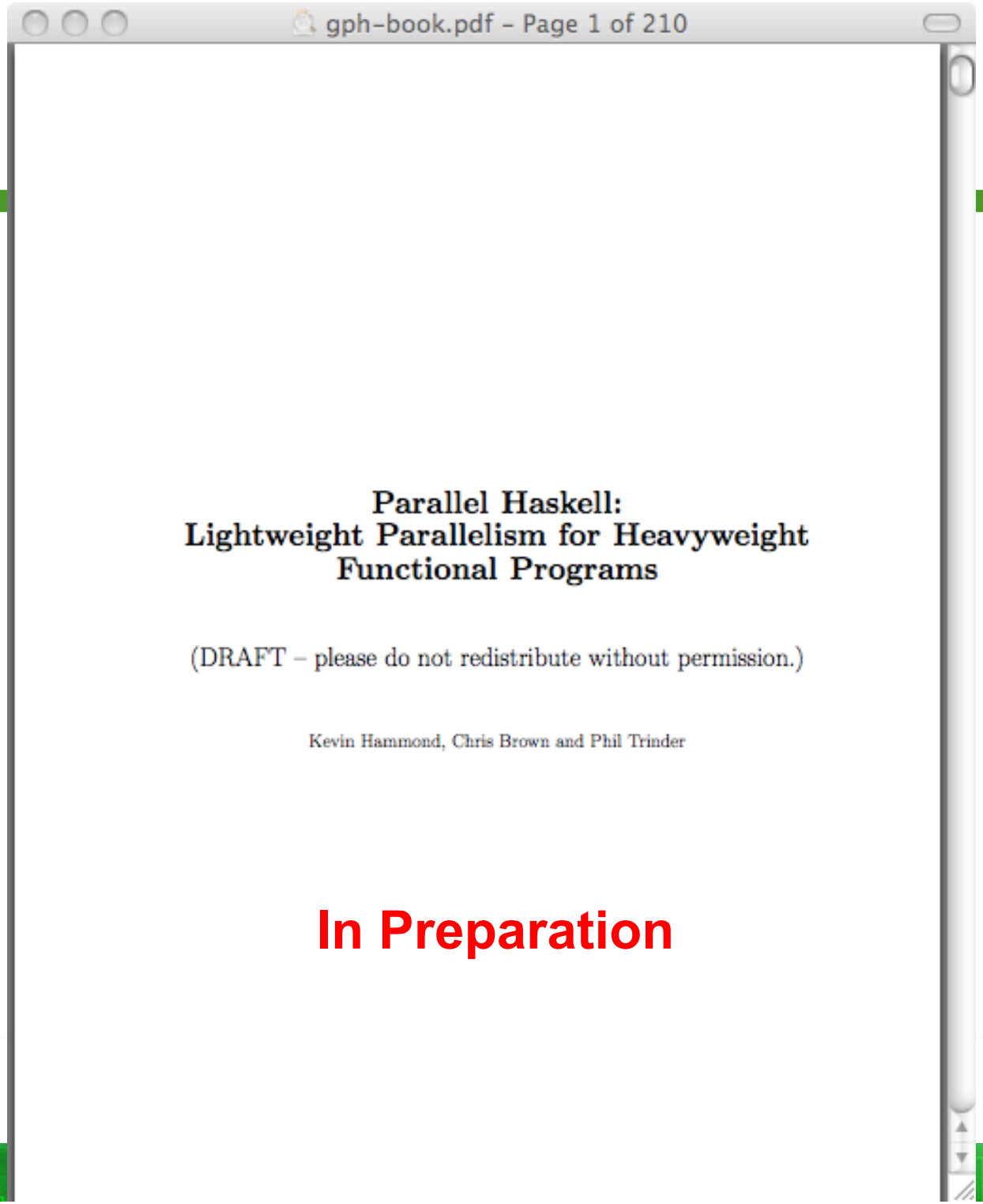
Trinder, Hammond, Loidl and Peyton Jones
"Algorithm + Strategy = Parallelism",
Journal of Functional Programming, 8(1):23–60, 1998

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

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

University of St Andrews





University of St Andrews

In Preparation

Pattern Cookbook

University of St Andrews

Pattern	Instances	Type	When to use

Divide-and-Conquer	`dC`
	`dc`
	`parThreshold`
		control-parallel	The operation can be defined recursively, subdividing its input at each recursive call; each application of the operation is independent; the results of recursive calls are combined into a final result; arguments may be created dynamically. Divide-and-conquer works well in a high latency (distributed) setting as well as a low latency (shared memory) setting, since there is no communication hotspot through a central master processor, and tasks can be created in a distributed rather than centralised way.

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

SCIENCE Symbolic Computation Infrastructure for Europe

ADVANCE StatArch

ParaPhrase

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 PhD Research Studentship

University of St Andrews - School of Computer Science

The School of Computer Science at the University of St Andrews has funding for students to undertake PhD research in any of the general research areas in the school:

http://www.cs.st-andrews.ac.uk/research

We are looking for highly motivated research students with an interest in these exciting research areas Our only requirements are that the proposed research would be good, we have staff to supervise it, and that you would be good at doing it. We have up to 5 funded studentships available for students interested in working towards a PhD. The studentships offers costs of fees and an annual tax-free maintenance stipend of about £13,590 per year for 3.5 years. Exceptionally well qualified and able students may be awarded an enhanced stipend of an additional £2,000 per year. Students should normally have or expect at least an upper-2nd class Honours degree or Masters degree in Computer Science or a related discipline.

For further information on how to apply, see our postgraduate web pages (**http://www.cs.st-andrews.ac.uk/prospective-pg**). The closing date for applications is **July 20th 2012** and we will make decisions on studentship allocation by **Sep 1st 2012**. (Applications after **July 20th** may be considered, at our discretion.) Informal enquiries can be directed to **pg-admin-cs@st-andrews.ac.uk** or to potential supervisors.

THANK YOU!

http://www.paraphrase-ict.eu

http://www.project-advance.eu

@paraphrase_fp7