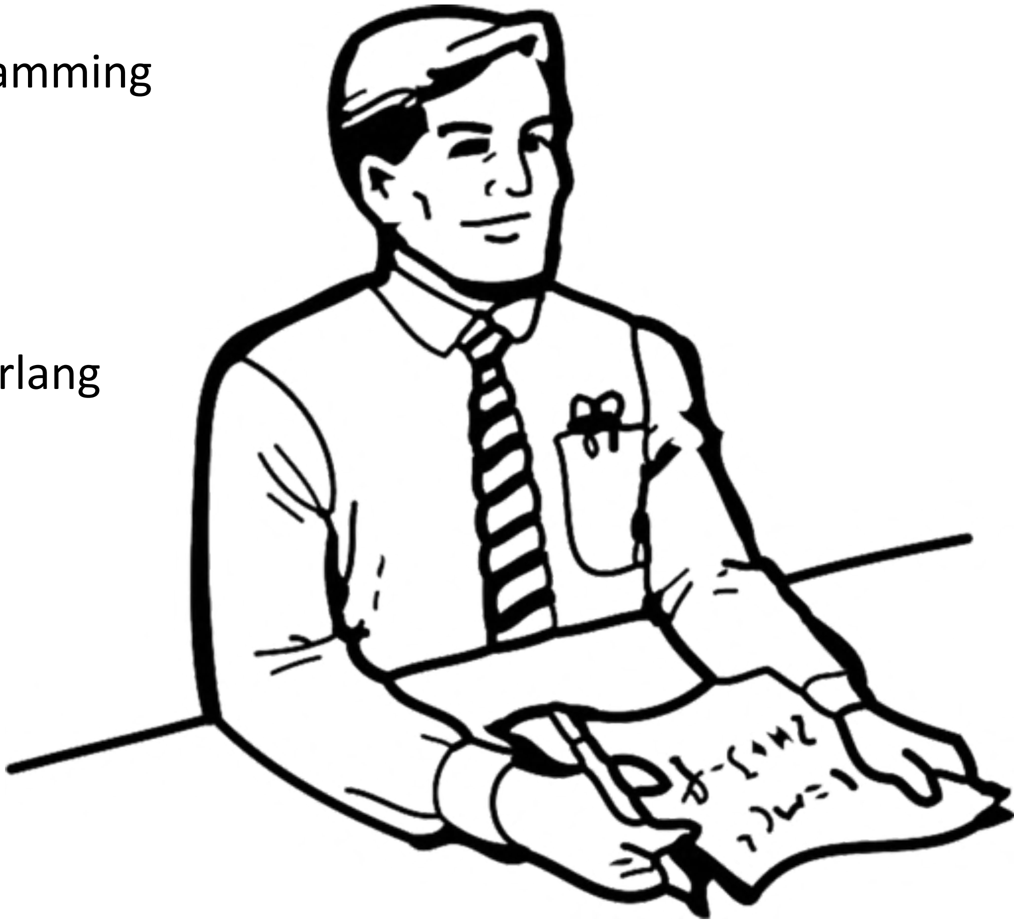
# *Introduction to Parallel Programming*

Chris Brown
University of St Andrews

cmb21@st-andrews.ac.uk

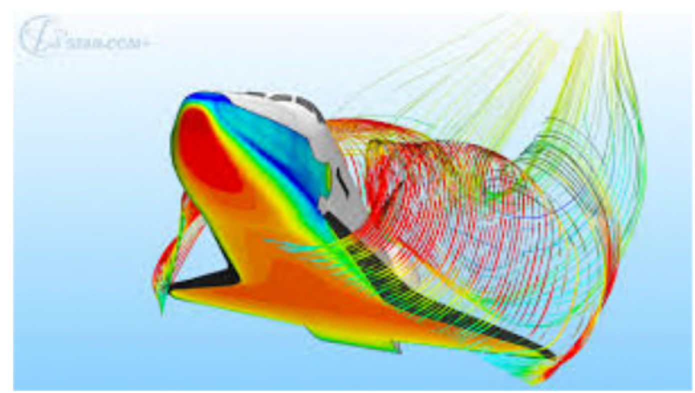
# Lecture Structure

- Course on Parallel Programming using Erlang
- Two 90 minute lectures
- Lecture 1 will cover
  - Introduction to parallel programming
  - parallel patterns
  - Erlang
- Lecture 2 will cover
  - The Erlang “skel” library
  - Writing parallel programs in Erlang
- 2 lab sessions
- Take notes…
- Ask me questions…
- Chat in the breaks…



# Trends in Parallel Computing

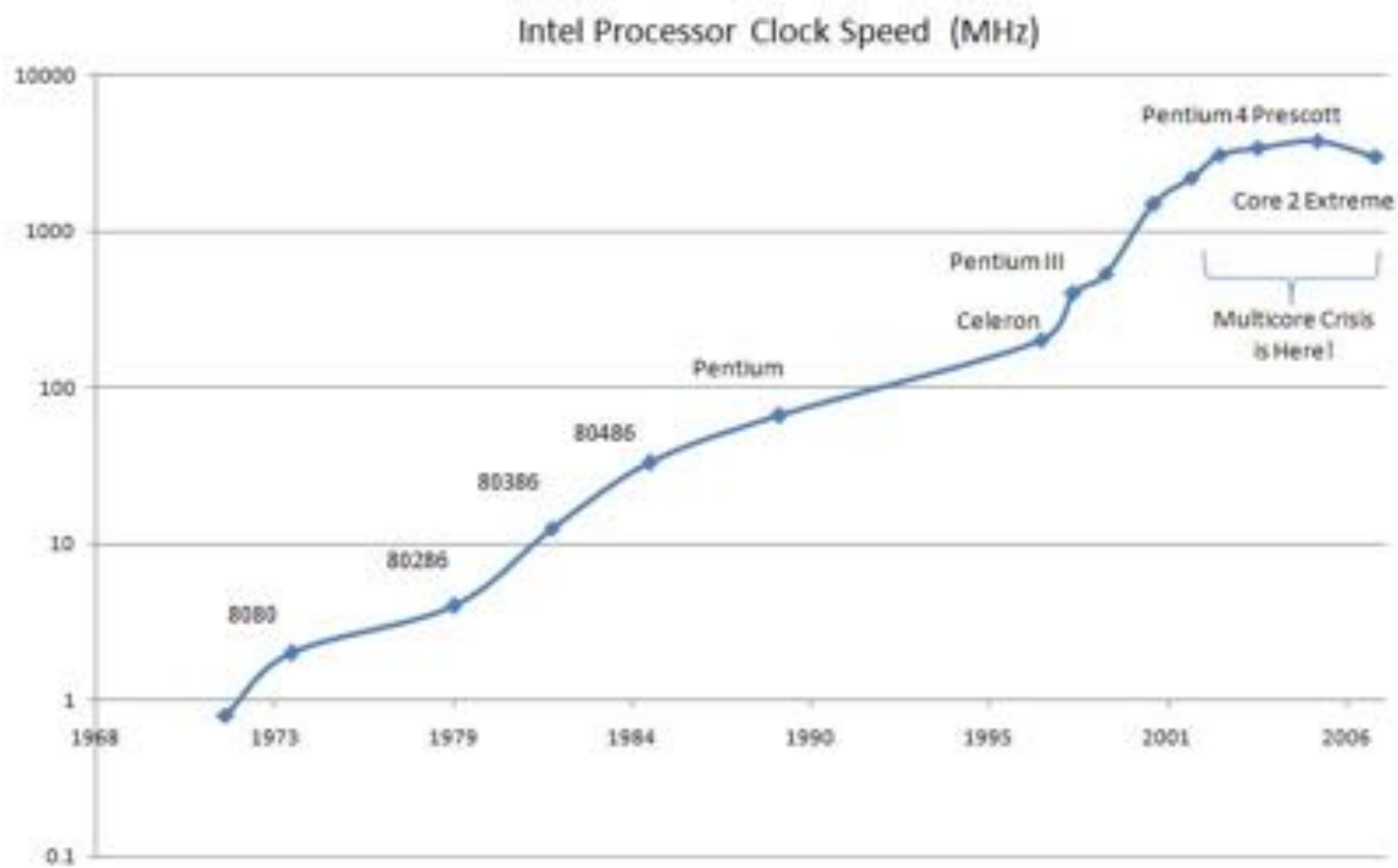
# The Internet of Everything?

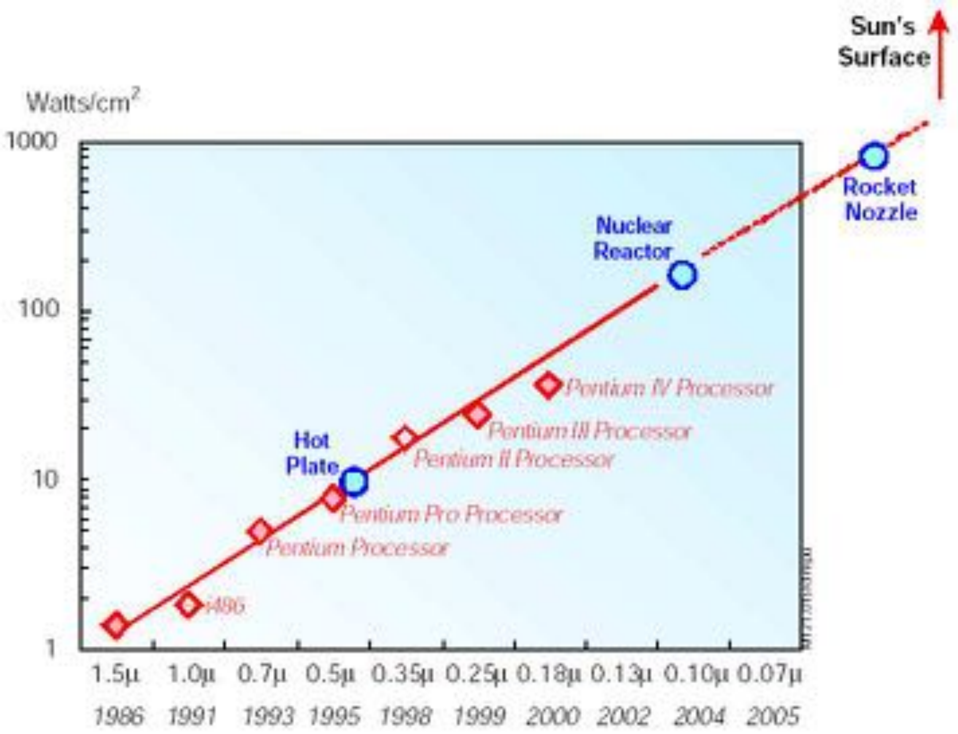
# Single core processors

- CPU only contains ONE core to process operations
- This was commonplace in computers from the 1970s up to about 2006.
- It is still used in many devices, such as some cheaper types of smartphones

# How do we make things go faster?



# Energy vs. Performance

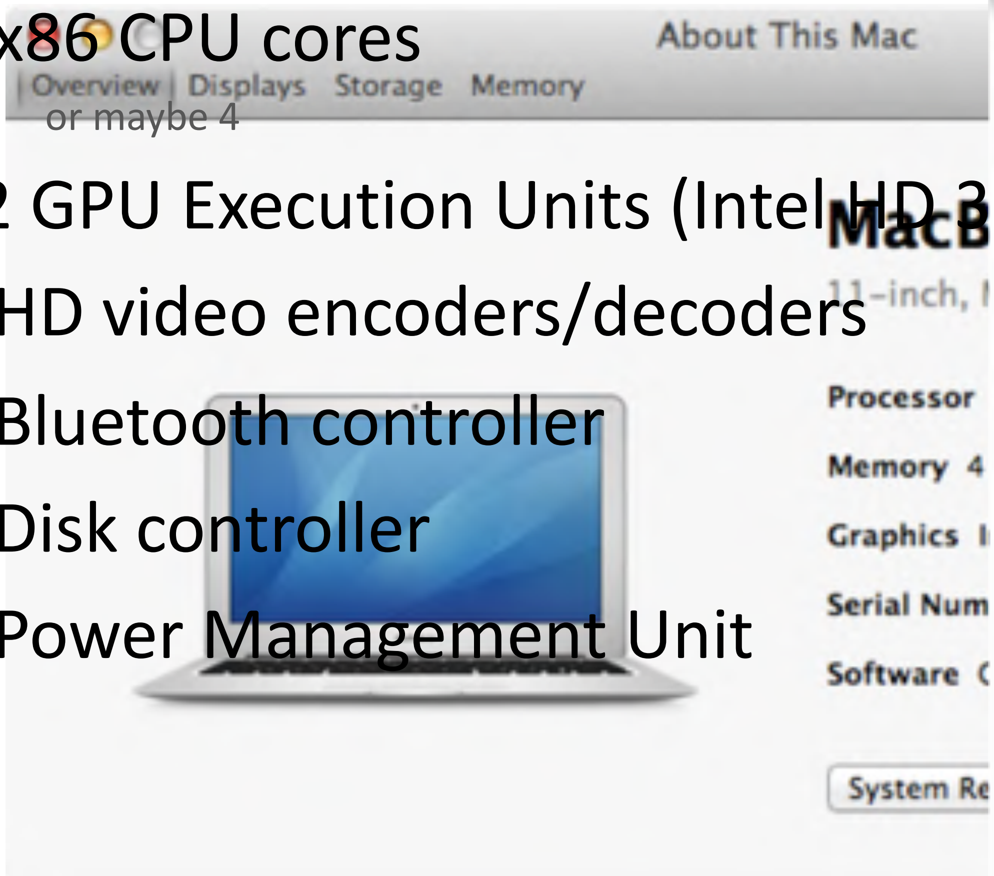


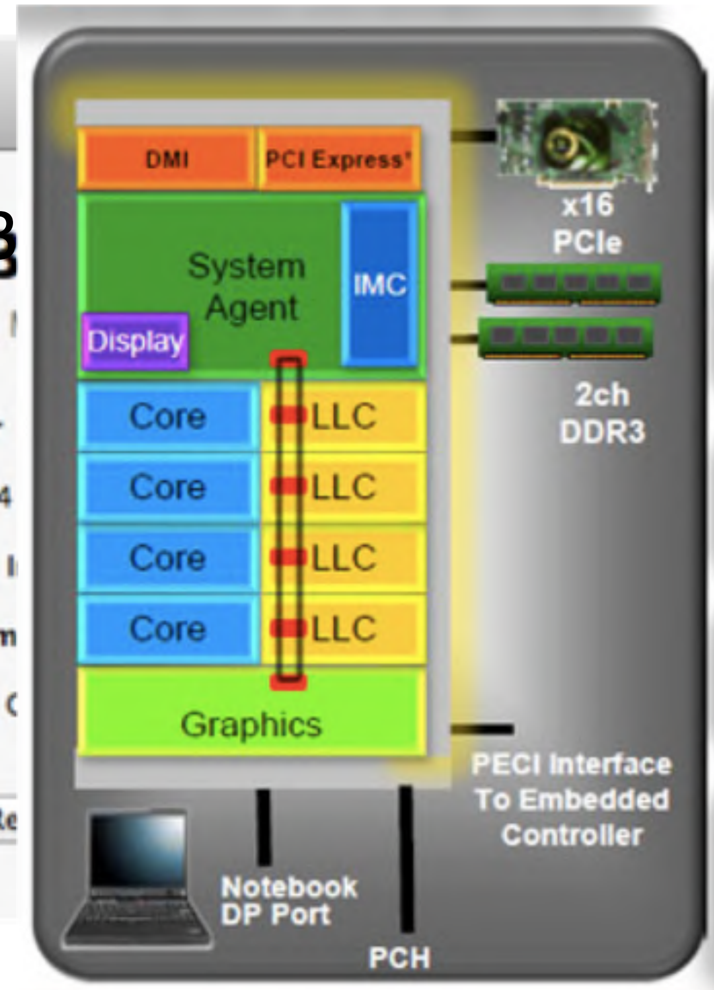

- Power is roughly cubic to clock frequency
- This means that we can't just increase the processor's speed…

# Even my laptop is multicore

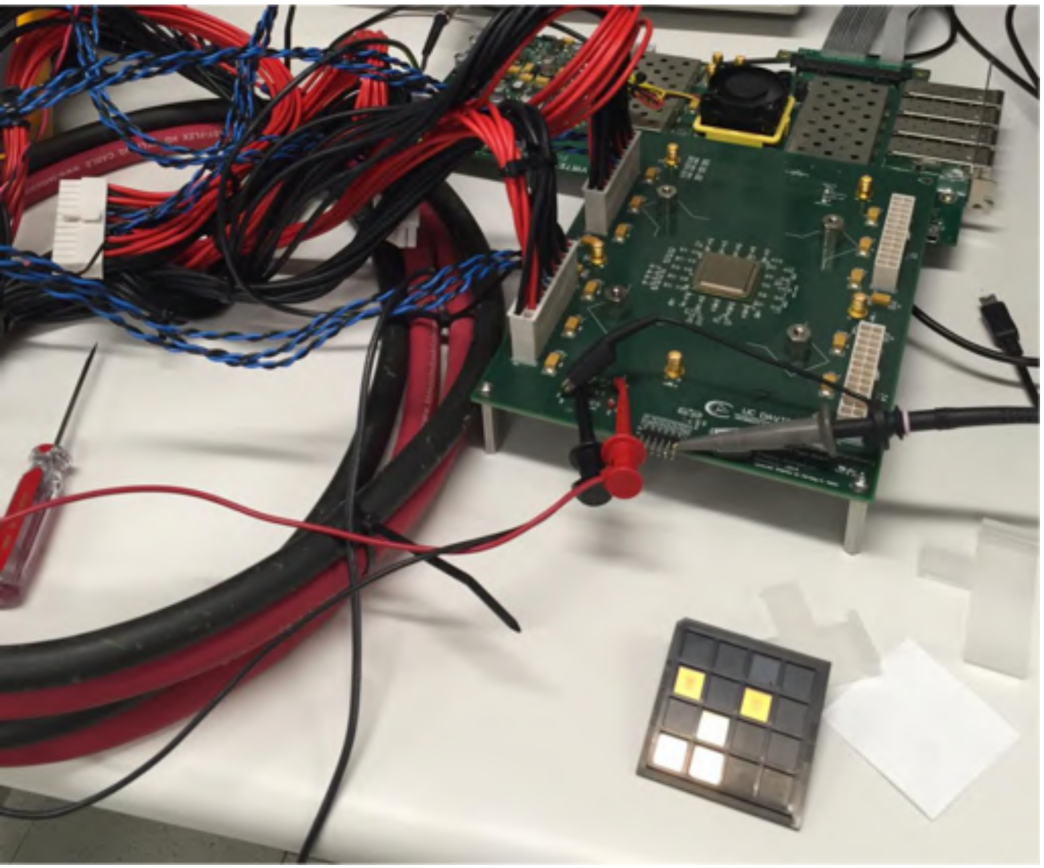
# How Many Cores does my laptop have?

- 2 x86 CPU cores
  - or maybe 4
- 12 GPU Execution Units (Intel HD 3
- 2 HD video encoders/decoders
- 1 Bluetooth controller
- 1 Disk controller
- 1 Power Management Unit
- …



# The world's first 1000 core processor



- 2018
- "Kilo-Core"
- 1000 independent programmable processors
- Designed by a team at the University of California, Davis
- 1.78 trillion instructions per second and contains 621 million transistors
- Each processor is independently clocked, it can shut itself down to further save energy
- 1,000 processors execute 115 billion instructions per second using 0.7 Watts
- Powered by a single AA battery

# The Fastest Computer in the World


**Sunway TaihuLight, National Supercomputer Centre, Wuxi**
93 petaflops/s (June 17, 2016)
40,960 Nodes; each with 256 custom Sunway cores
**10,649,600 cores in total!!!**

# It’s not just about large systems

- Even mobile phones are multicore
  - Samsung Exynos 5 Octa has 8 cores, 4 of which are “dark”
- Performance/energy tradeoffs mean systems will be increasingly parallel
- If we don’t solve the multicore challenge, then no other advances will matter!


ALL Future Programming will be Parallel!

Everyone in this room is already an expert in parallel programming.

ALDI

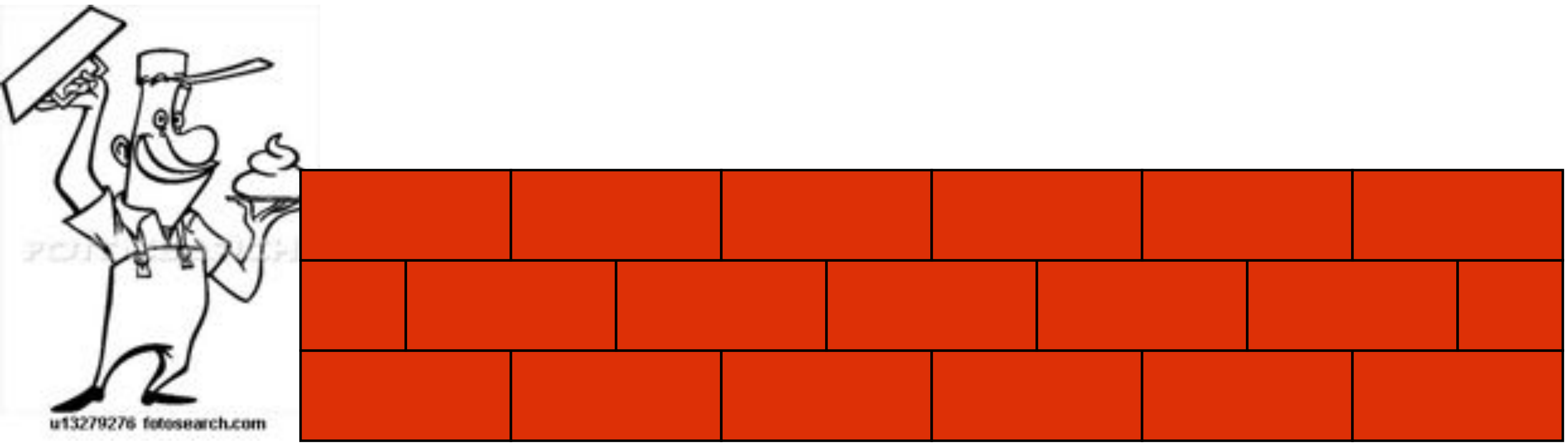
LIDL


USN
alamy stock photo
www.alamy.com

You really need multiple checkouts and queues….

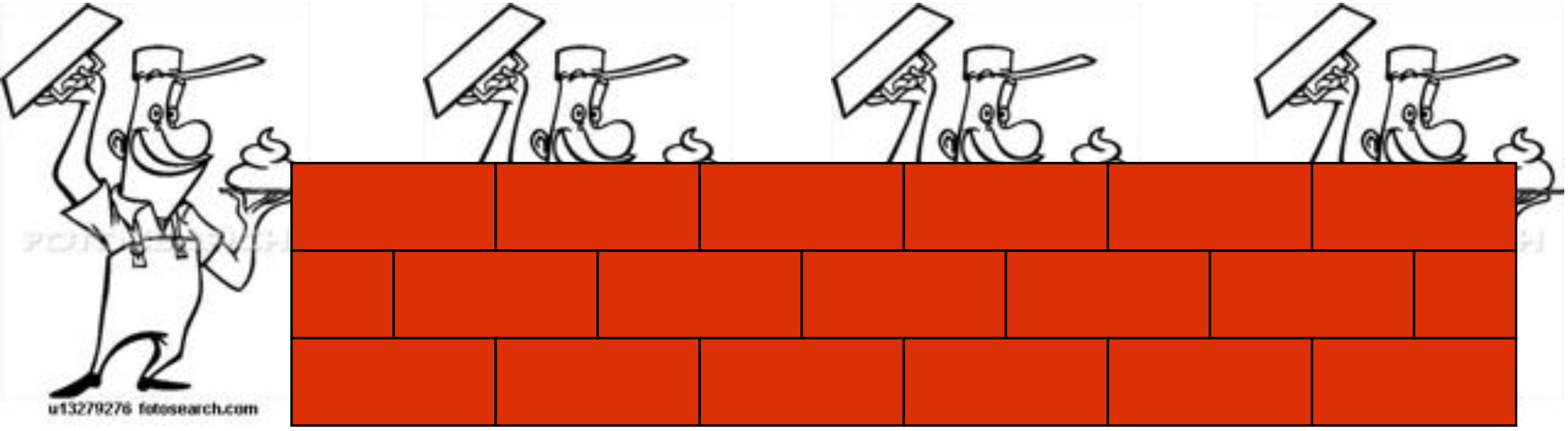
# Coffee, anyone?

# How to build a wall

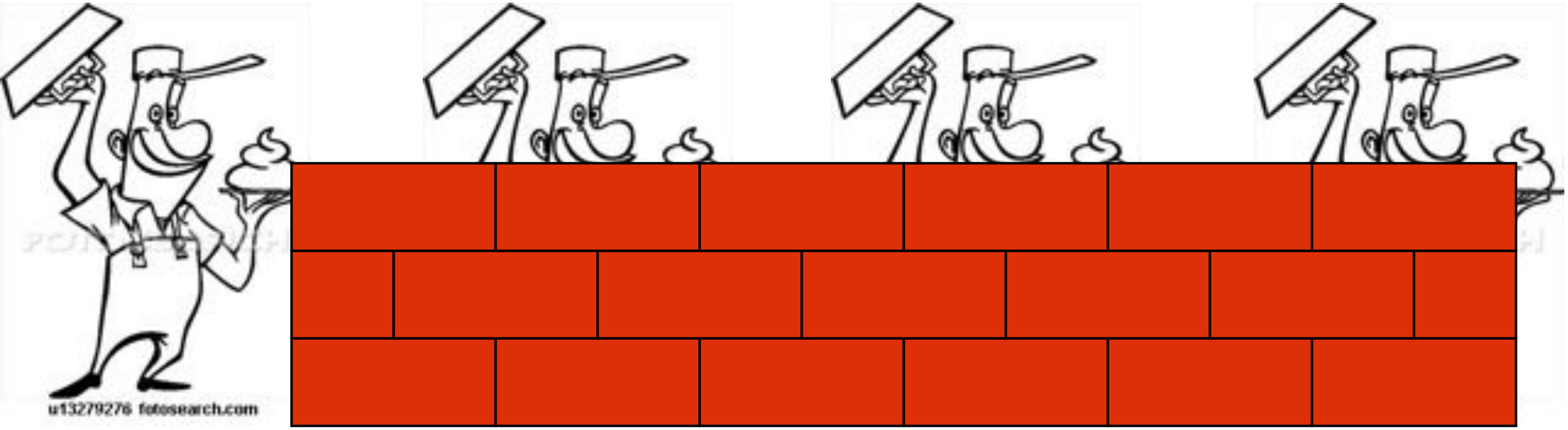



(with apologies to Ian Watson, Univ. Manchester)

# How to build a wall *faster*



# How NOT to build a wall



# Current Programming Models

# pThreads/OpenMP

- Designed for Shared-memory systems
  - communication via shared variables
- Explicit thread creation
- Synchronisation requires explicit locks
  - mutexes
- VERY easy to deadlock

```
#include <pthread.h>

void *fn(void *arg);
main(){
   pthread_t mainthread;
   ptr_t arg = ...;
   void *result;

   pthread_create(&mainthread,NULL
                  fn,(void*) arg);

   pthread_join(mainthread,
                 (void*) &result);
   ...
}
```

# Cilk/Cilk Plus

- spawn/sync constructs
- uses global shared memory
- avoids explicit locking
  - but explicit synchronisation

```
cilk int fib (int n)
{
    if (n < 2) return n;
    else
    {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
    }
}
```

# PVM/MPI

- Designed for shared-nothing systems
  - but some implementations work (well) on shared-memory systems

- One process per node

- Communication via explicit message-passing
  - synchronous or asynchronous
  - possibly broadcast/multicast

- No structure to messages, easy to break protocols

```
#include <mpi.h>

int main (int argc, char *argv[])
{
 …
  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &id);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
   ....
  MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Recv(&rq, 1, MPI_INT, MPI_ANY_SOURCE, REQUEST,
                world, &status);
  …
  MPI_Send(res, CHUNKSIZE, MPI_INT,
                status.MPI_SOURCE, REPLY, world);
  …
  MPI_Finalize();
  return 0;
}
```

# C++1X

- Thread support
  - std::thread class
- Atomic locking
  - faster than mutex
  - but still a lock protocol
- futures and promises
  - std::future class
- Still a shared-memory model

```
#include <iostream>
#include <future>
#include <thread>

int main()
{
   // future from a packaged_task
   std::packaged_task<int()> task([](){ return 42; });
   std::future<int> fut = task.get_future();
   std::thread(std::move(task)).detach();
   fut.wait();
  std::cout << "Done!\nResult is: "
            << fut.get() << '\n';
}
```

# “Lock-Free” Programming

- Rather than protecting a critical region with a *mutex*, use a single hardware instruction
  - e.g double compare-and-swap
- A single “commit” releases all changes (barrier)
- Only for shared-memory
- VERY easy to get wrong; VERY hard to debug

# How does parallelism usually work?

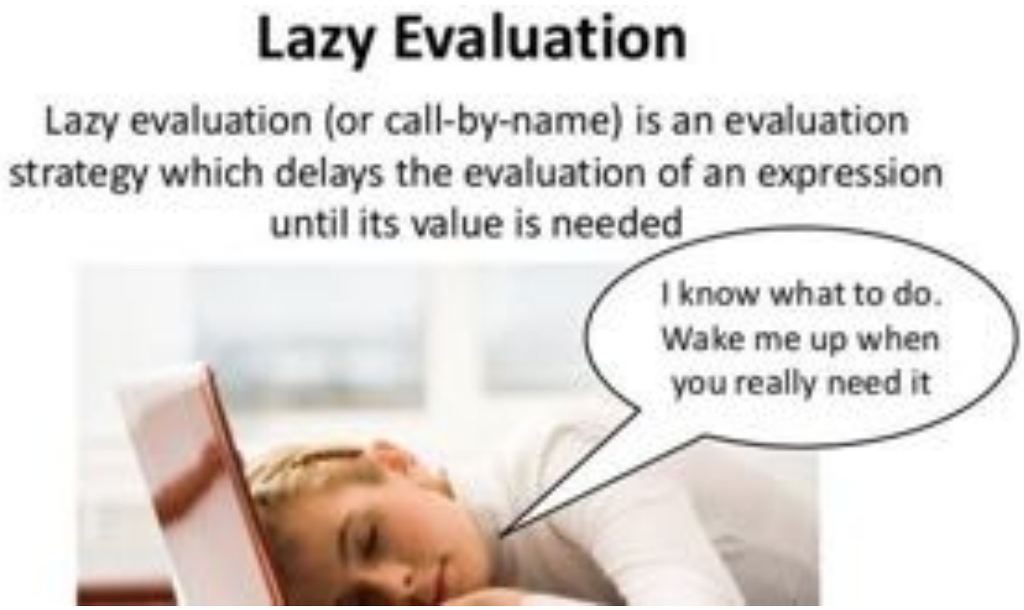
- Most programs are heavily procedural by nature
  - Do this, do that, …
- Parallelism always a bolted-on afterthought
  - Lots of threads
  - Message passing
  - Mutexes
  - Shared memory
- Almost impossible to correctly implement…
  - Deadlocks
  - Race conditions
  - Synchronization
  - Non-determinism
  - Etc.
  - Etc.

# What about functional programming?

- In theory, perfect model
- Purity is perfect parallelism model
  - No side effects!
- Implicit parallelism models
- Small programmer overhead
- Minimal language effort
  - E.g. Haskell only has two parallel primitives
- No locks, deadlocks or race conditions

# I said "in theory"

- Haskell is beautiful, but …
- Lazy semantics are the opposite of what you need for parallelism
- Spend more time understanding laziness….



# Parallelism in Haskell

In Haskell, there are only two operators you need to do parallelism.

```
par :: a -> b -> b
```

```
a `par` b
```

This creates a spark for a and returns b

# A Spark?

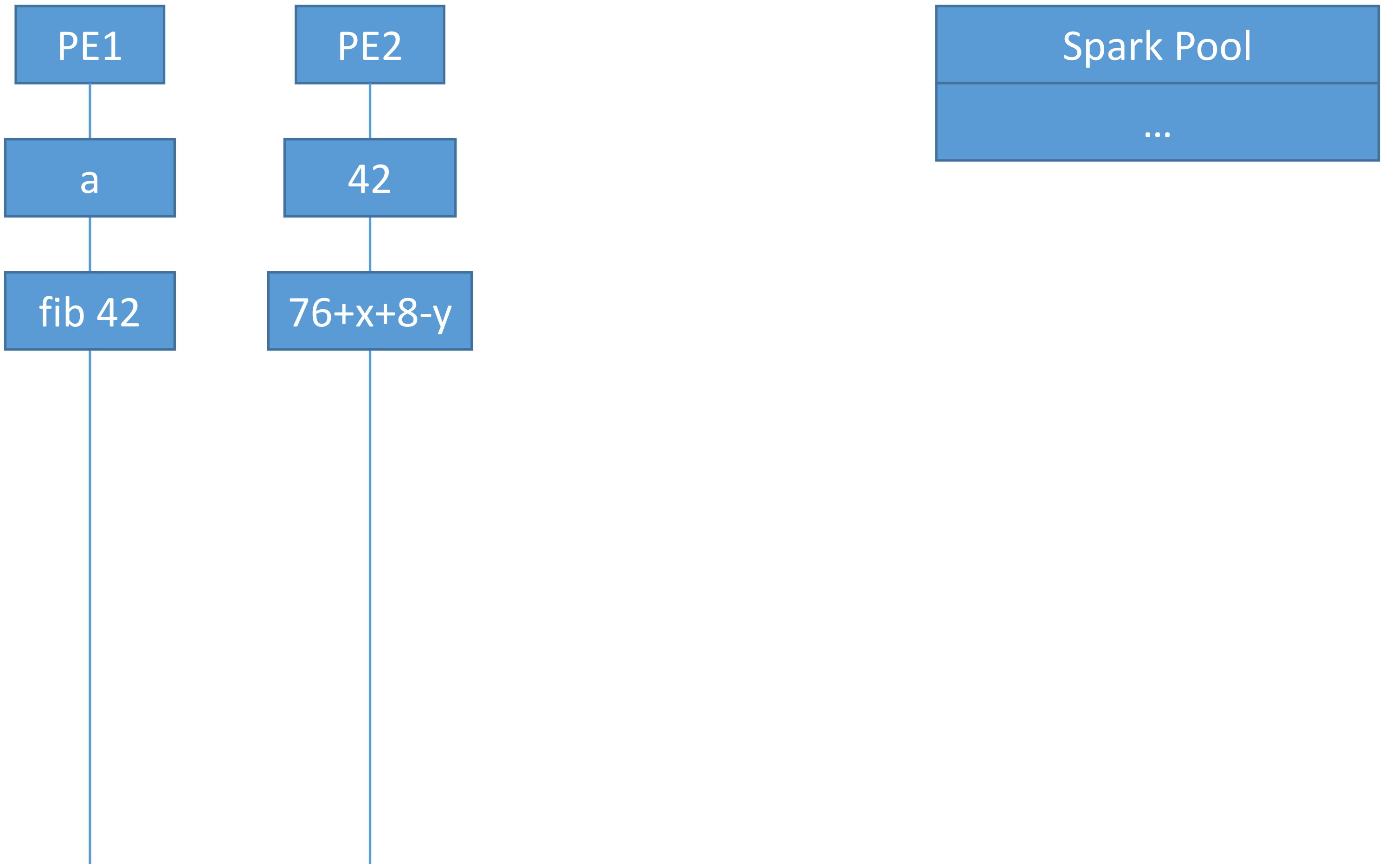
A kind of "promise"

```
a `par` b

x `par` f x where f x = …
```

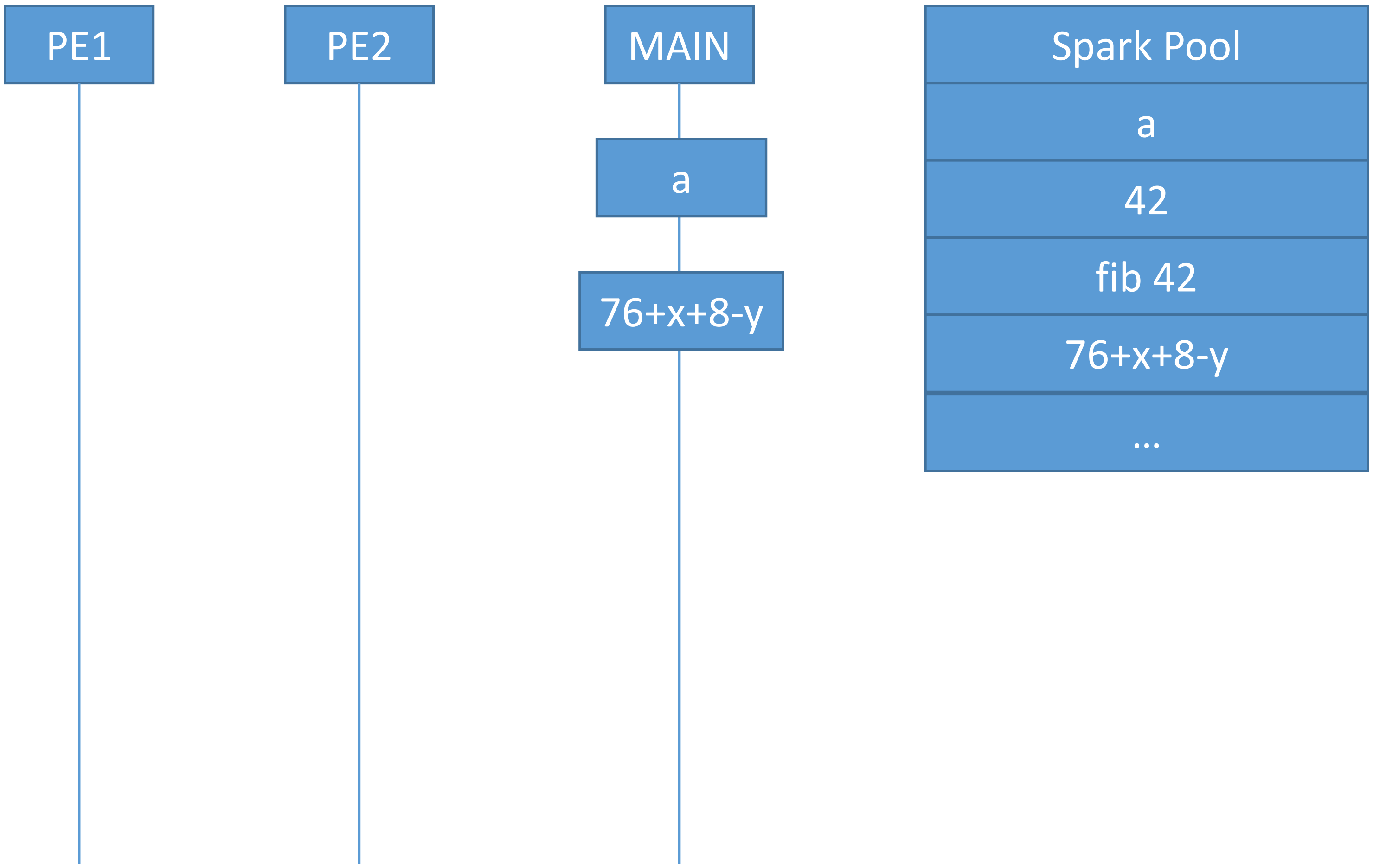
"I will try my best to evaluate a in parallel to b, unless you go ahead and evaluate a before I get around to doing that.

… in which case I won't bother."

# The Spark Pool



# The Spark Pool

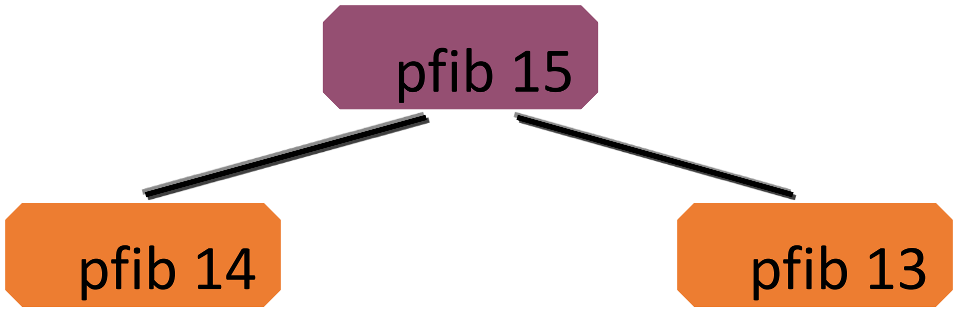


# Divide and Conquer Evaluation

pfib 15

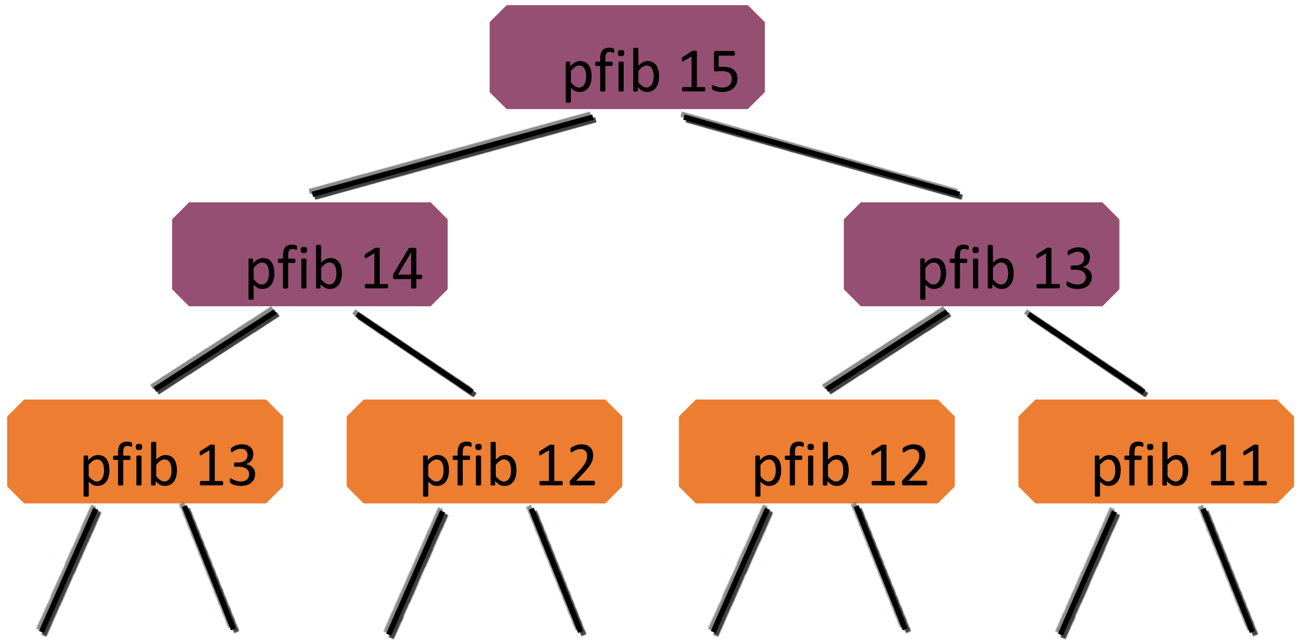
```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `par` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Divide and Conquer Evaluation



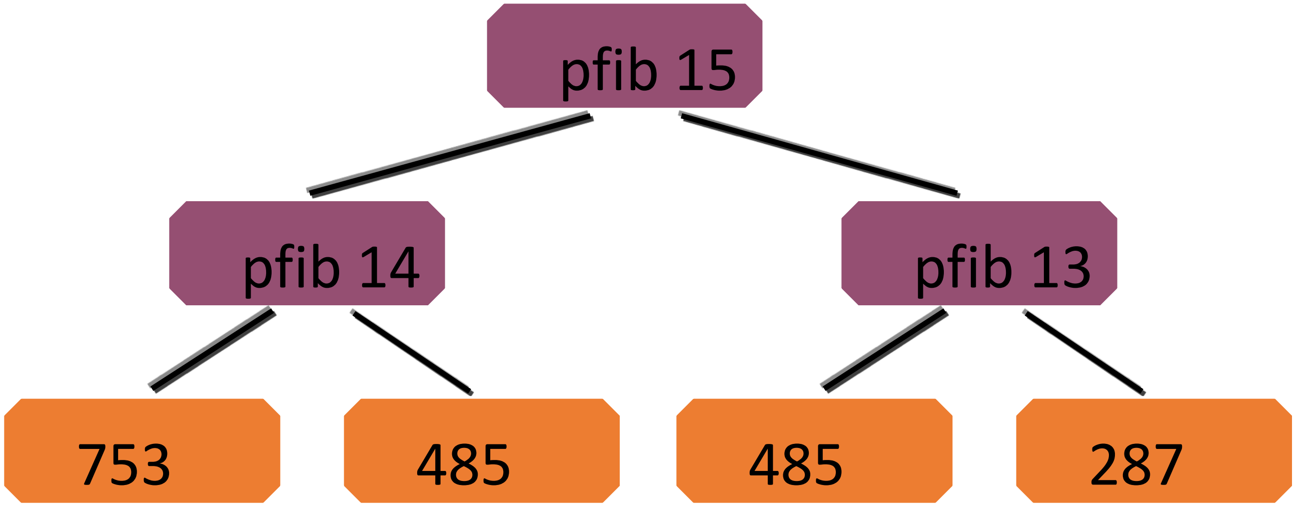

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `par` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Divide and Conquer Evaluation



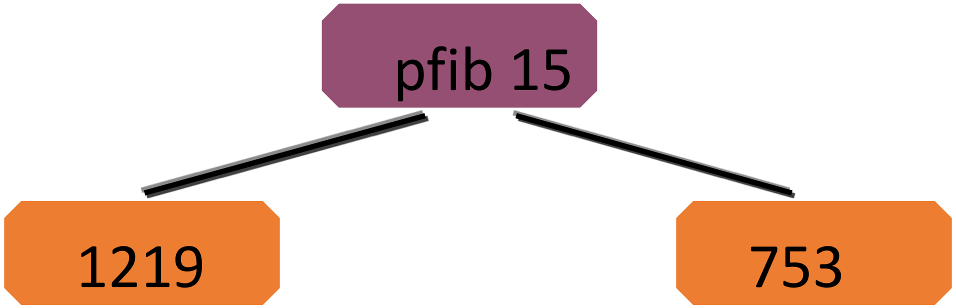

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `par` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Divide and Conquer Evaluation




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

# Divide and Conquer Evaluation




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

# Divide and Conquer Evaluation

1973

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

# Seq

*Seq is the most basic method of introducing strictness to Haskell*

```
seq :: a -> b -> b

_|_ `seq` b = _|_
a `seq` b   = b
```

***Seq doesn't sequence and doesn't evaluate anything!***

***Only puts a dependency on both its arguments***

***When b is demanded, a must (sort of) be evaluated, too***

# The *pseq* Construct

```
a `pseq` b
```

**evaluate a and return the value of b**

**For example**

```
x `pseq` f x where x = …
```

**first evaluates x, and then returns f x**

# Evaluate-and-Die

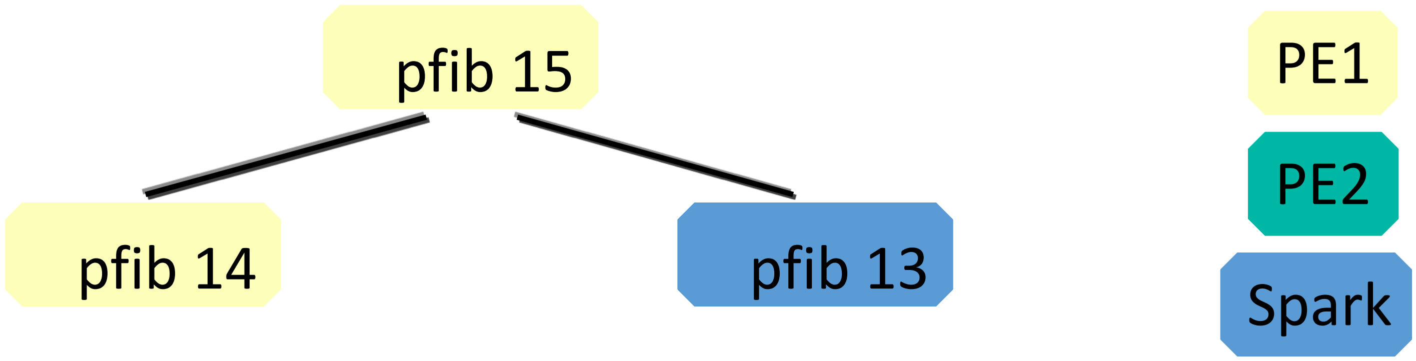
pfib 15

PE1

PE2

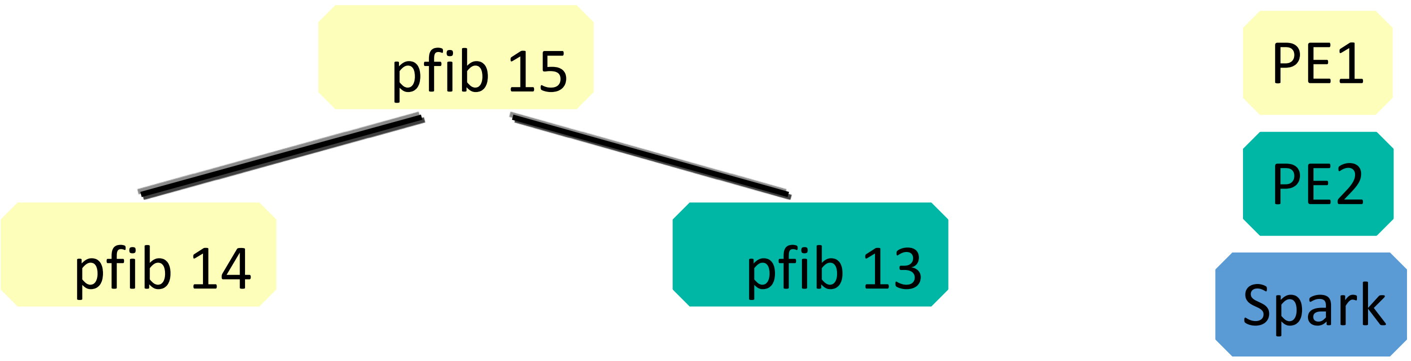
```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2+1)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



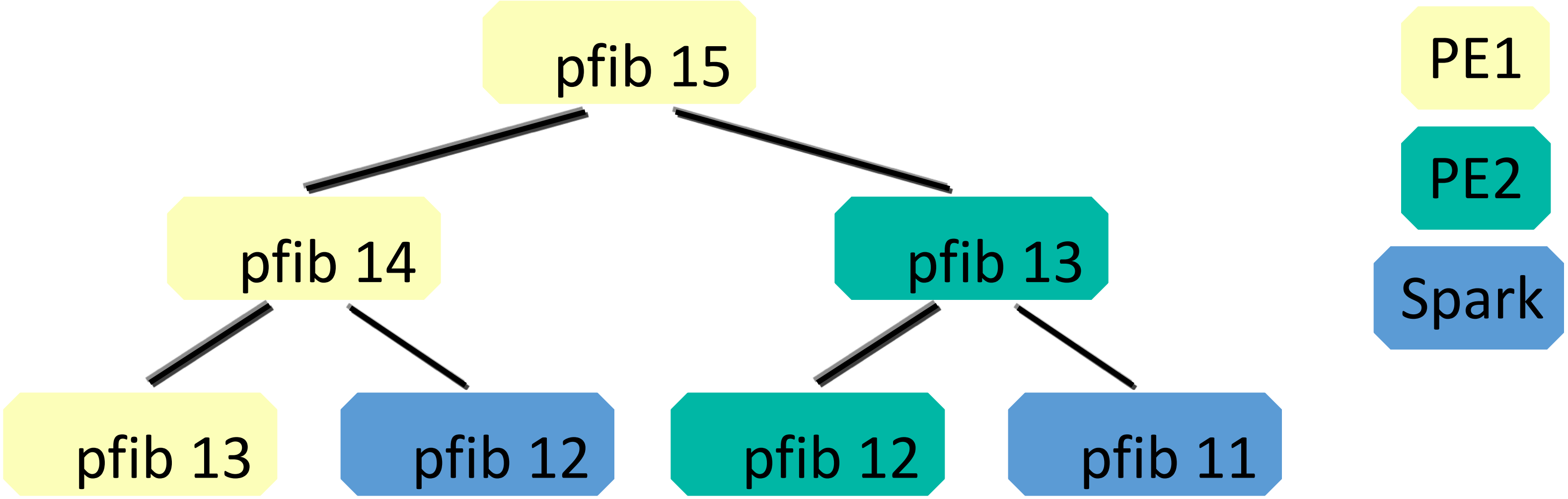

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



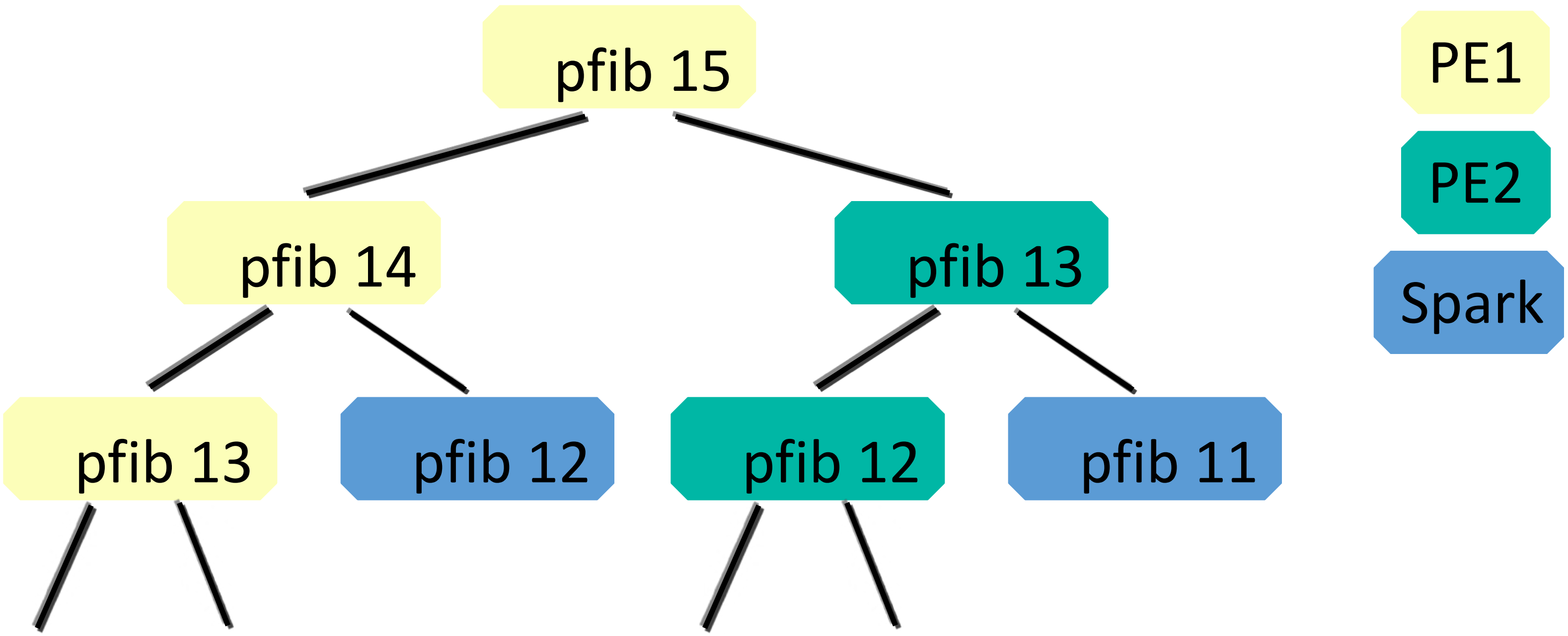

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



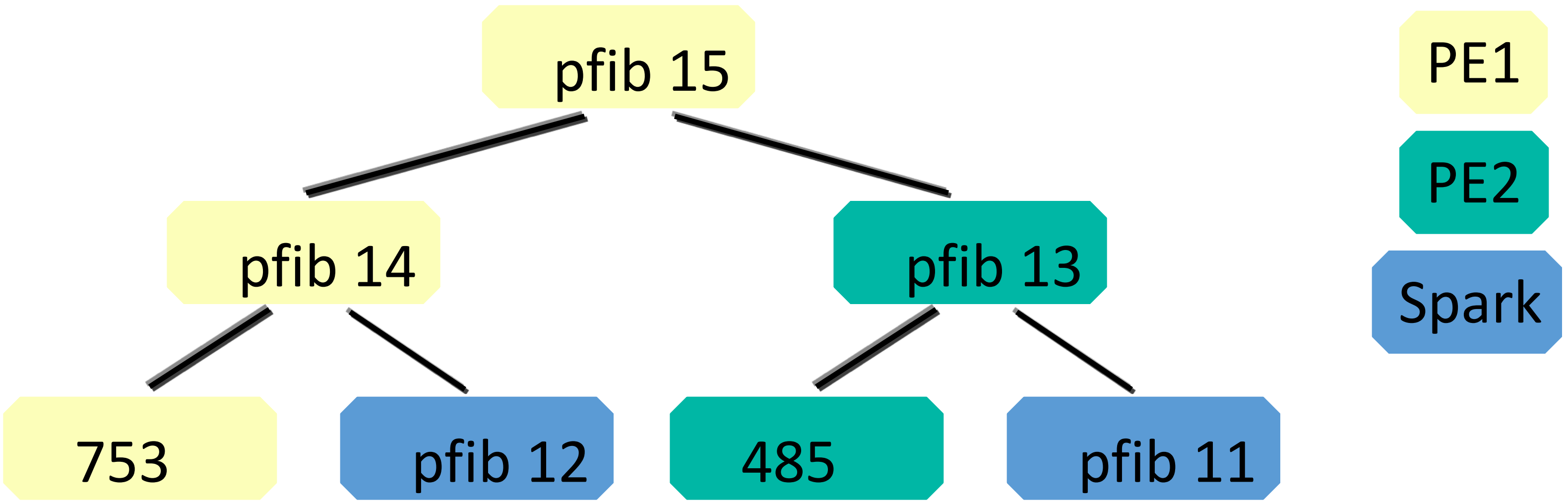

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



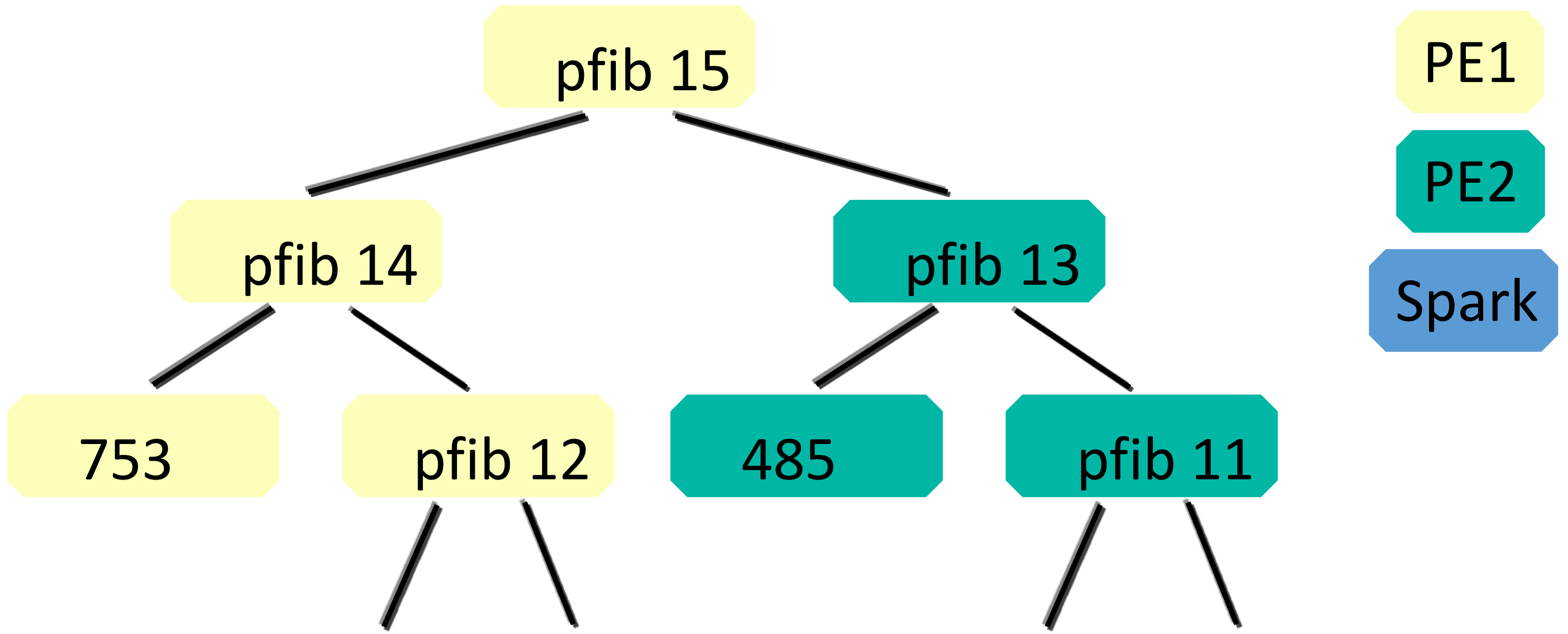

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



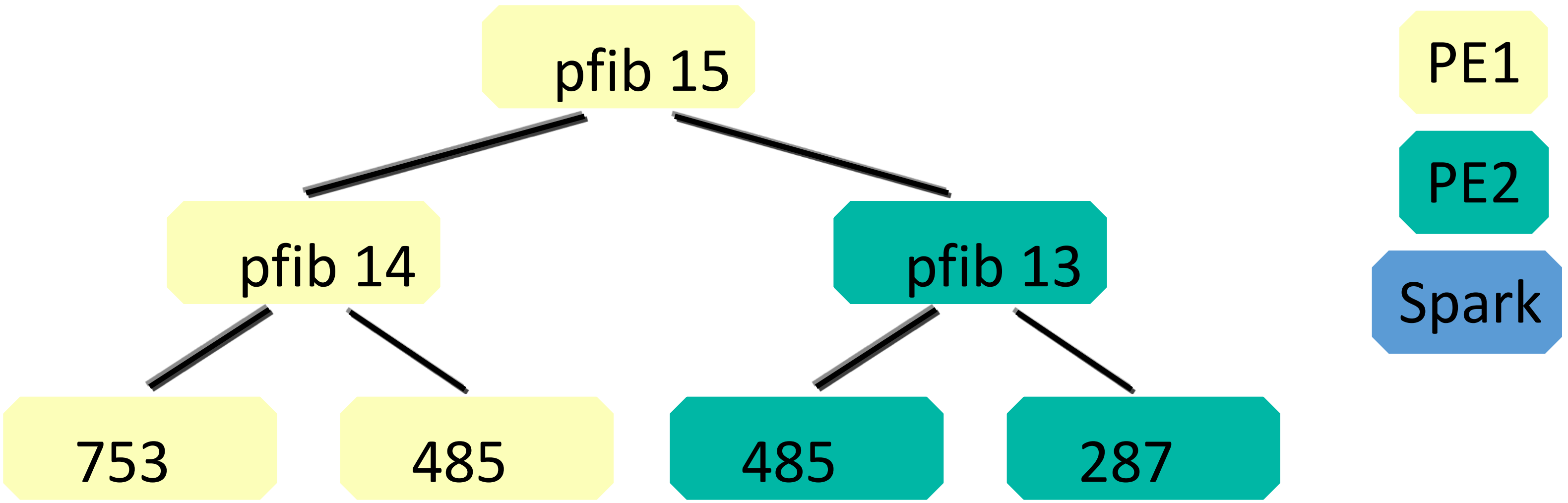

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die



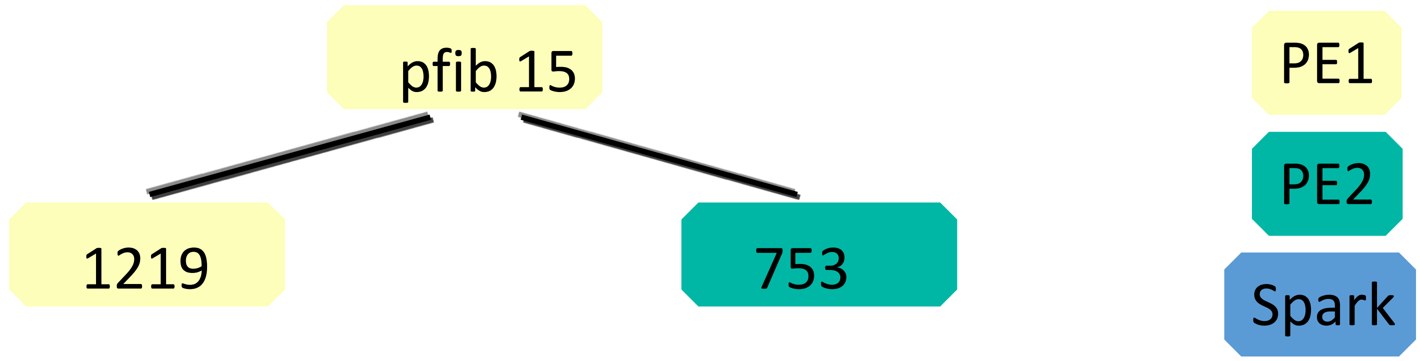

```
pfib n | n <= 1    = 1
       | otherwise = n2 `par` (n1 `pseq` n1+n2)
    where n1 = pfib (n-1)
          n2 = pfib (n-2)
```

# Evaluate-and-Die




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

# Evaluate-and-Die




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

# Evaluate-and-Die

1973

PE1

PE2

Spark

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

# What about Erlang?

- Functional language
- No Laziness
- Built in concurrency
- Process model
- Message passing
- “lightweight” threads
  - In Erlang, you can just spawn everything, right??

# Fib, in Erlang

```erlang
fib0(0) -> 0;
fib0(1) -> 1;
fib0(N) -> fib0(N-1) + fib0(N-2).
```

http://trigonakis.com/blog/2011/02/27/parallelizing-simple-algorithms-fibonacci/

# Fib, in Erlang

```
fib1(0) -> 0;
fib1(1) -> 1;
fib1(N) -> Self = self(),
           spawn(fun() ->
                     Self ! fib1(N-1)
                 end),
           spawn(fun() ->
                     Self ! fib1(N-2)
                 end),
           receive
               F1 ->
                   receive
                       F2 ->
                           F1 + F2
                   end
           end.
```

http://trigonakis.com/blog/2011/02/27/parallelizing-simple-algorithms-fibonacci/

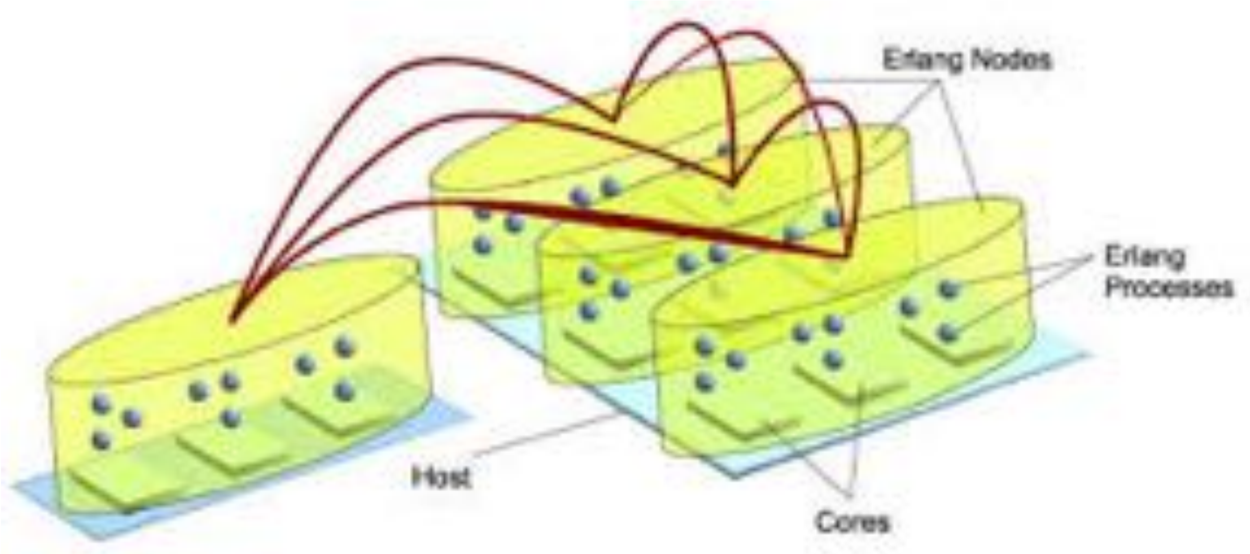
# Does it actually go faster?

- Fib0: Average 44.7 microseconds
- Fib1: average 2202.0 microseconds

- But I thought in Erlang you just spawn everything and get amazing concurrency and parallelism for free, right? I mean, “lightweight” threads!!

http://trigonakis.com/blog/2011/02/27/parallelizing-simple-algorithms-fibonacci/

# The Erlang Model




Thanks to Natalia Chechina, University of Bournemouth

# Erlang, heavyweight concurrency

- Turns out these lightweight threads are not really lightweight at all
  - millisecond magnitude to set up
  - Comparable to a pthread!
  - Micro/milli second for message to pass between threads
  - (depends on the message being sent)

  - Fib 15 (sequential) = 44.7 microseconds
  - Fib 15 (concurrent) = **2202 microseconds**

  - Aprox., 140 microseconds to spawn each process

# Thinking Parallel

- **Fundamentally, programmers must learn to "think parallel"**
  - this requires new *high-level* programming constructs
    - perhaps dealing with hundreds of *millions* of threads
- **You cannot program effectively while worrying about processes.**
  - Arguably, too heavy and low-level!
- **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!
- **You cannot program effectively without performance information**
  - this needs to be included as part of the design!

# Parallelism is not Concurrency

- Concurrency is a programming abstraction
  - The ***illusion*** of independent threads of execution
  - Scheduling
- Parallelism is a hardware artifact
  - The ***reality*** of threads executing at the same time
  - PERFORMANCE!
- Concurrency is about breaking a program down into independent units of computation
- Parallelism is about making things happen at the same time

# Parallelism is not Concurrency (2)

- A concurrent thread may be broken down into many parallel threads
  - or none at all
- Parallelism can sometimes be modeled 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

# How NOT to Program Multicore

- **Use concurrency techniques!**
  - Transactional memory, spin-locks, monitors, mutexes
- **Program at a low abstraction level**
  - Without first understanding the parallelism
- **Program with a fixed architecture in mind**
  - Specific numbers of cores
  - Specific bus structure
  - Specific instruction set
  - Specific type of GPU
- **Think shared memory**
  - Big arrays, shared variables….

# Parallel Patterns

# 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

# Multithreaded programming

# Multi-core Software is Difficult!

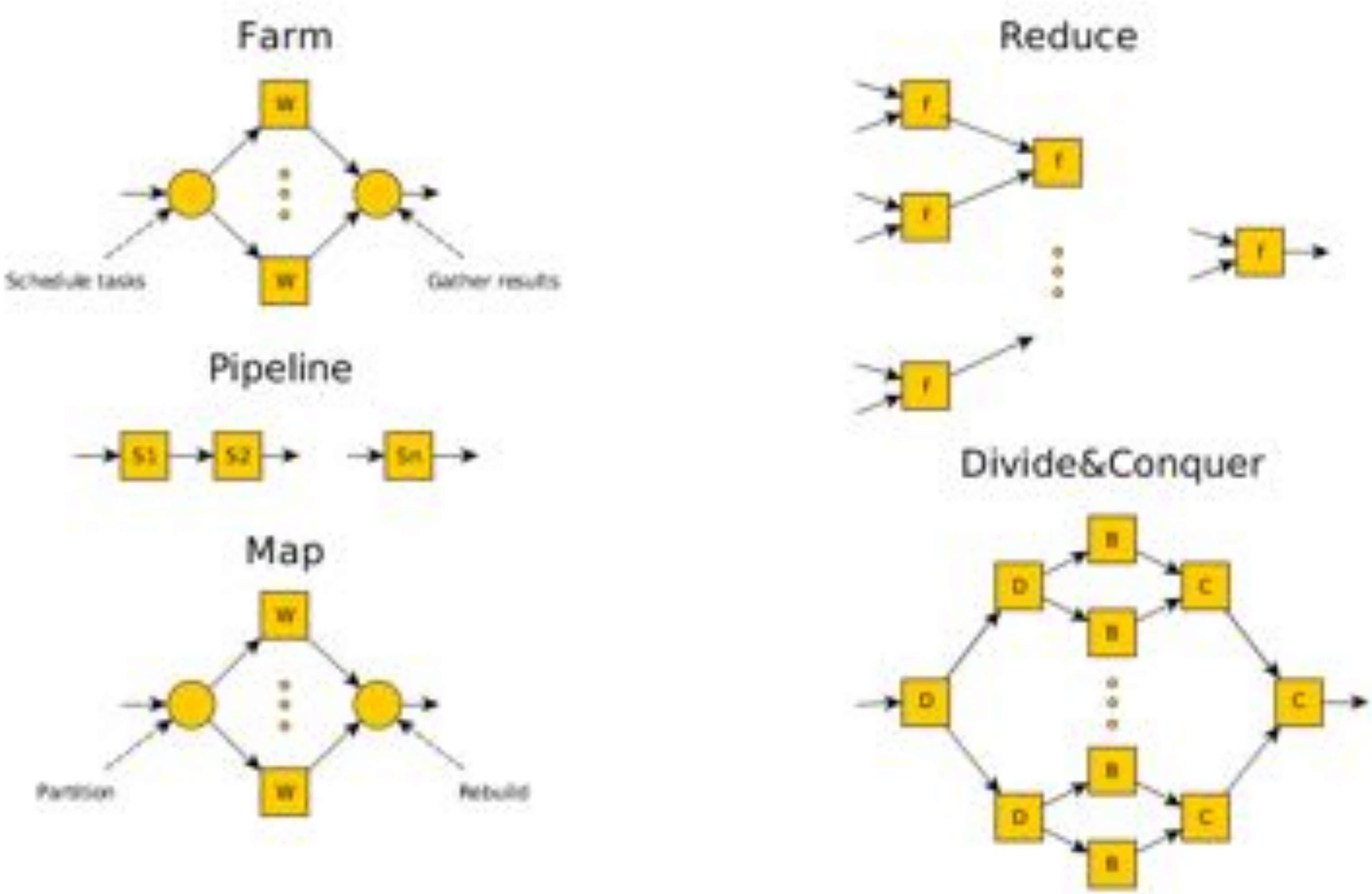
# Multi-Threaded Programming

**Chuck Norris can write multi-threaded Java applications with a single thread**

Image source: http://4.bp.blogspot.com/-RhqCAgEyhxs/T7CqtsC0RUI/AAAAAAAAJuw/LnkKl4xBcFA/s1600/ChuckNorris.png

# Patterns are Everywhere...

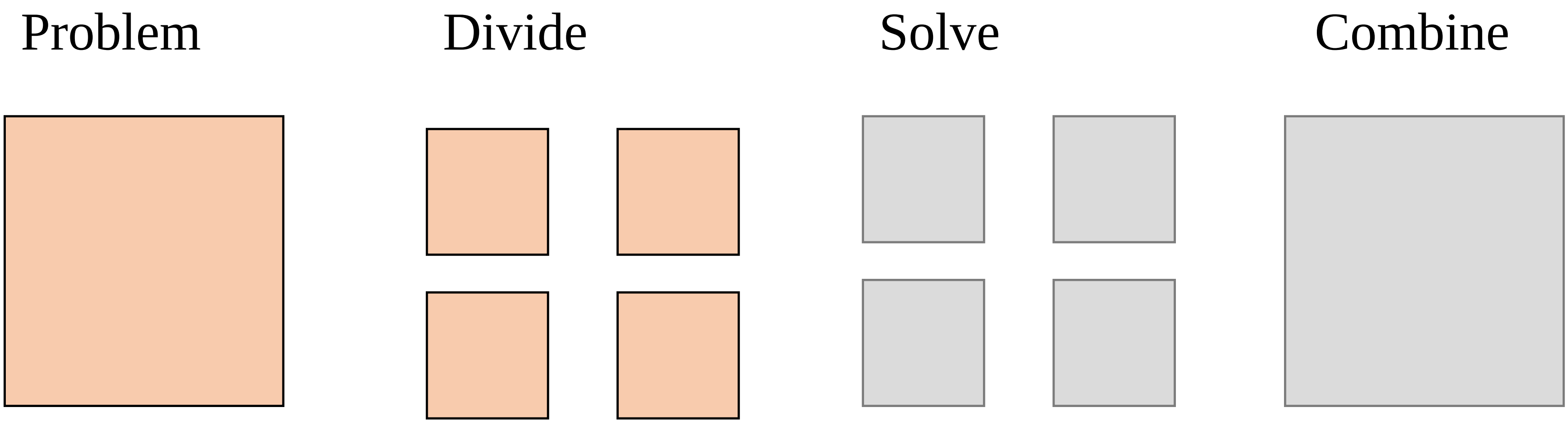
# …Including Parallel Software




# Car manufacturing

# Divide-and-Conquer

- If the problem is trivial
  - *solve it!*
- Otherwise, *divide* the problem into two (or more) parts
  - *Solve* each part independently
  - *Combine* all the sub-results to give the result



# Divide-and-Conquer (wikipedia)

# D&C Example

```
procedure D&C (x: input data ) is
   begin
       if BaseCondition(x) then
           return baseSolve(x);
        else
           Split x into sub-tasks;
           Use D&C to solve each sub-task;
           Merge the subtasks results through the Conquer
              (cont.)function;
        end if;
   end D&C;
```

# Parallel Divide-and-Conquer in C

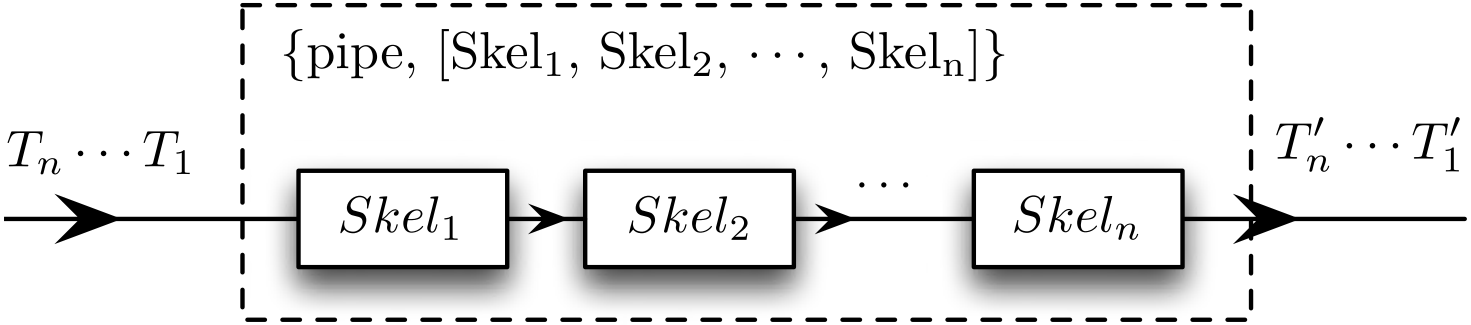
```
void *dc(void *valToFind){
…
pthread_t leftThread;
pthread_t rightThread;

if(finished(valToFind))
  return (valToFind);

else{
  long newValToFind1 = leftval (valToFind);
  long newValToFind2 = rightval (valToFind);
```

# Parallel Divide-and-Conquer in C

```
 …

  pthread_create(&leftThread,NULL,dc,(void*)
newValToFind1);
  pthread_create(&rightThread,NULL,dc,(void*)
newValToFind2);


pthread_join(leftThread,(void*)&returnLeft);

pthread_join(rightThread,(void*)&returnRight)
;

  return (combine(returnLeft, returnRight));
}
```

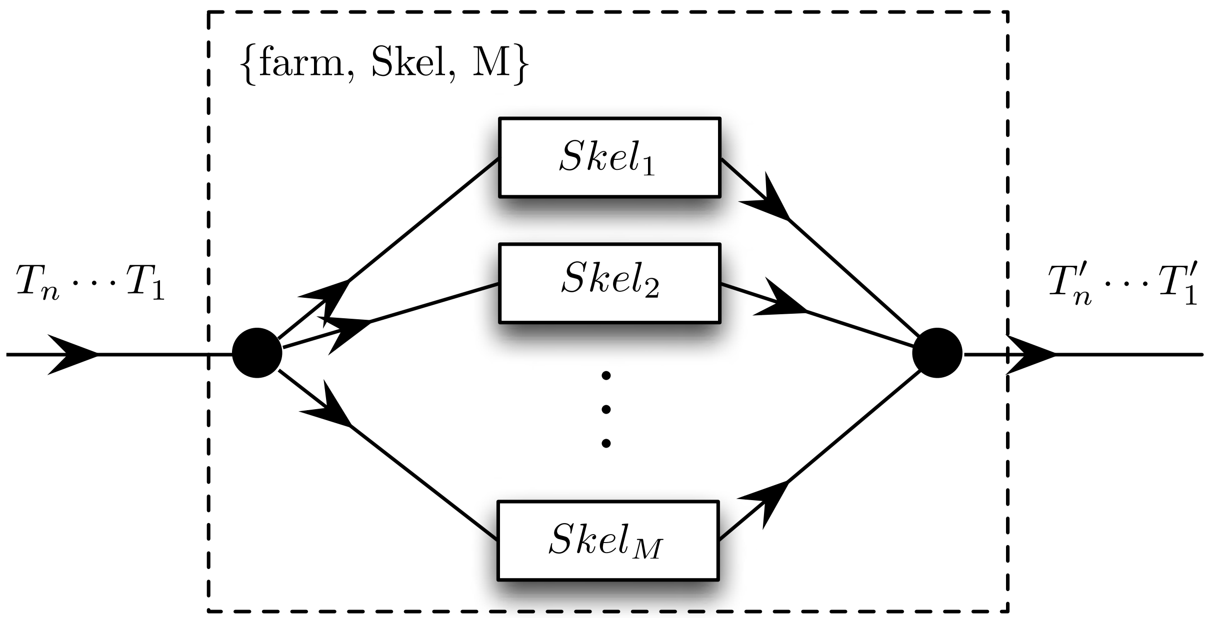
# Parallel Pipeline

- Each stage of the pipeline is executed in parallel
- The computation at each stage can be as complex as you wan
- The input and output are streams

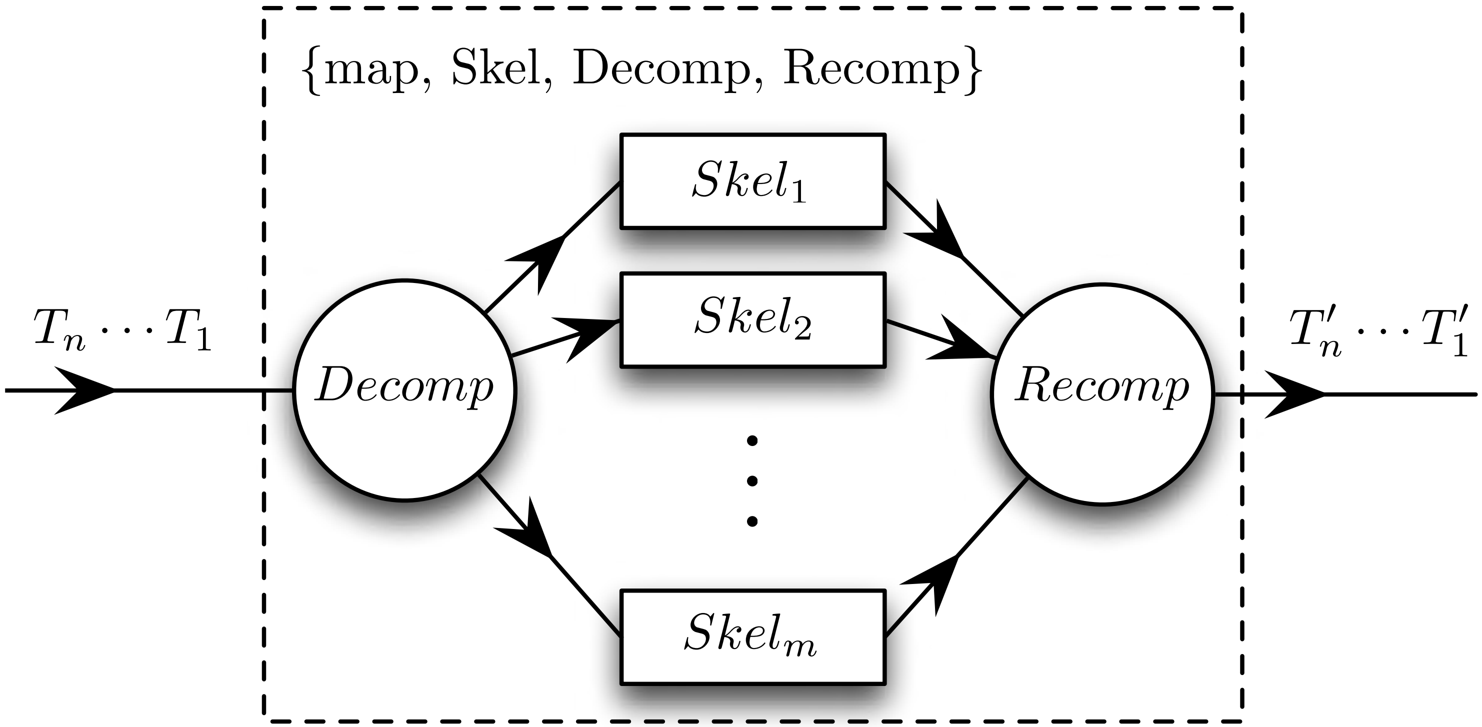


# Farm

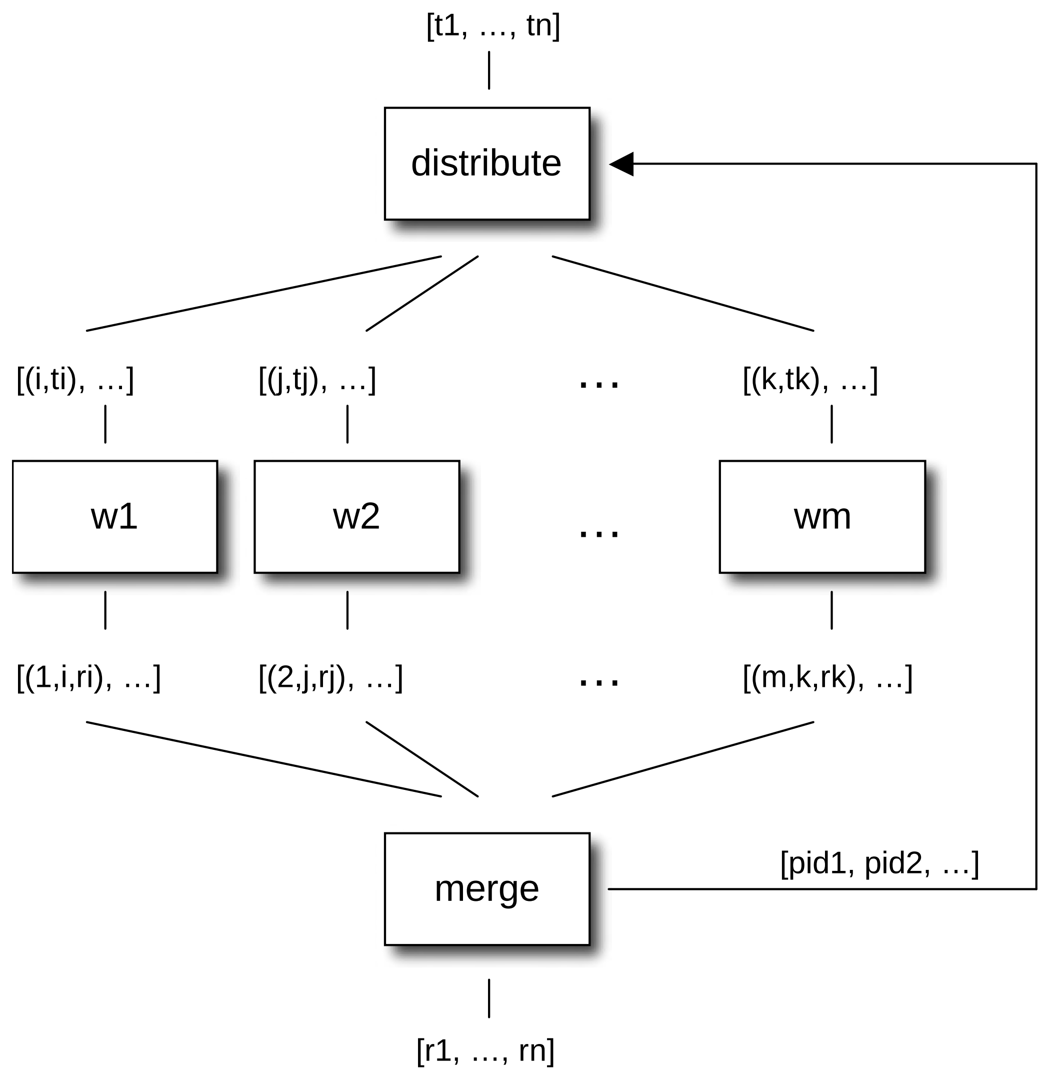
- Each worker is executed in parallel
- A bit like a 1-stage pipeline



# Map

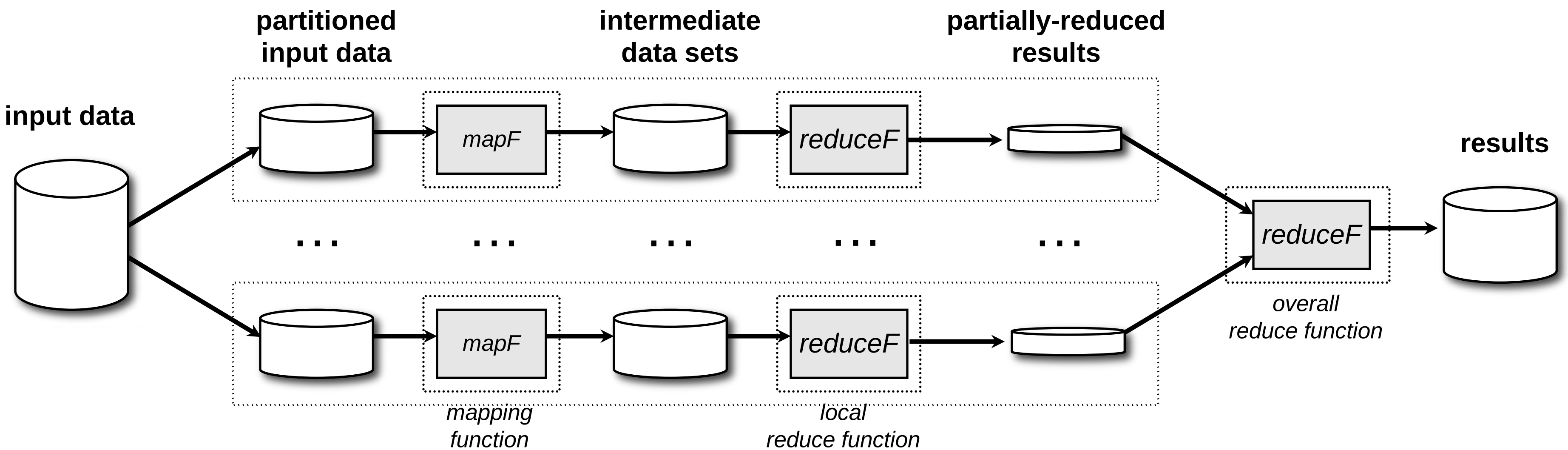

{map, Skel, Decomp, Recomp}
$T_n \cdots T_1$
Decomp
$Skel_1$
$Skel_2$
$Skel_m$
Recomp
$T'_n \cdots T'_1$

# Workpool



[t1, …, tn]
distribute
[(i,ti), …]
[(j,tj), …]
…
[(k,tk), …]
w1
w2
…
wm
[(1,i,ri), …]
[(2,j,rj), …]
…
[(m,k,rk), …]
merge
[pid1, pid2, …]
[r1, …, rn]

# mapReduce



partitioned input data
intermediate data sets
partially-reduced results
input data
mapF
reduceF
mapF
reduceF
. . .
. . .
. . .
. . .
. . .
reduceF
results
overall reduce function
mapping function
local reduce function

# Next Lectures…

- Introduction to Erlang
- Introduction to Parallel Patterns
- Writing parallel programs in Erlang
- Performance

# Thank you!

cmb21@st-andrews.ac.uk

*@chrismarkbrown*