University
of
St Andrews

SEVENTH FRAMEWORK
PROGRAMME

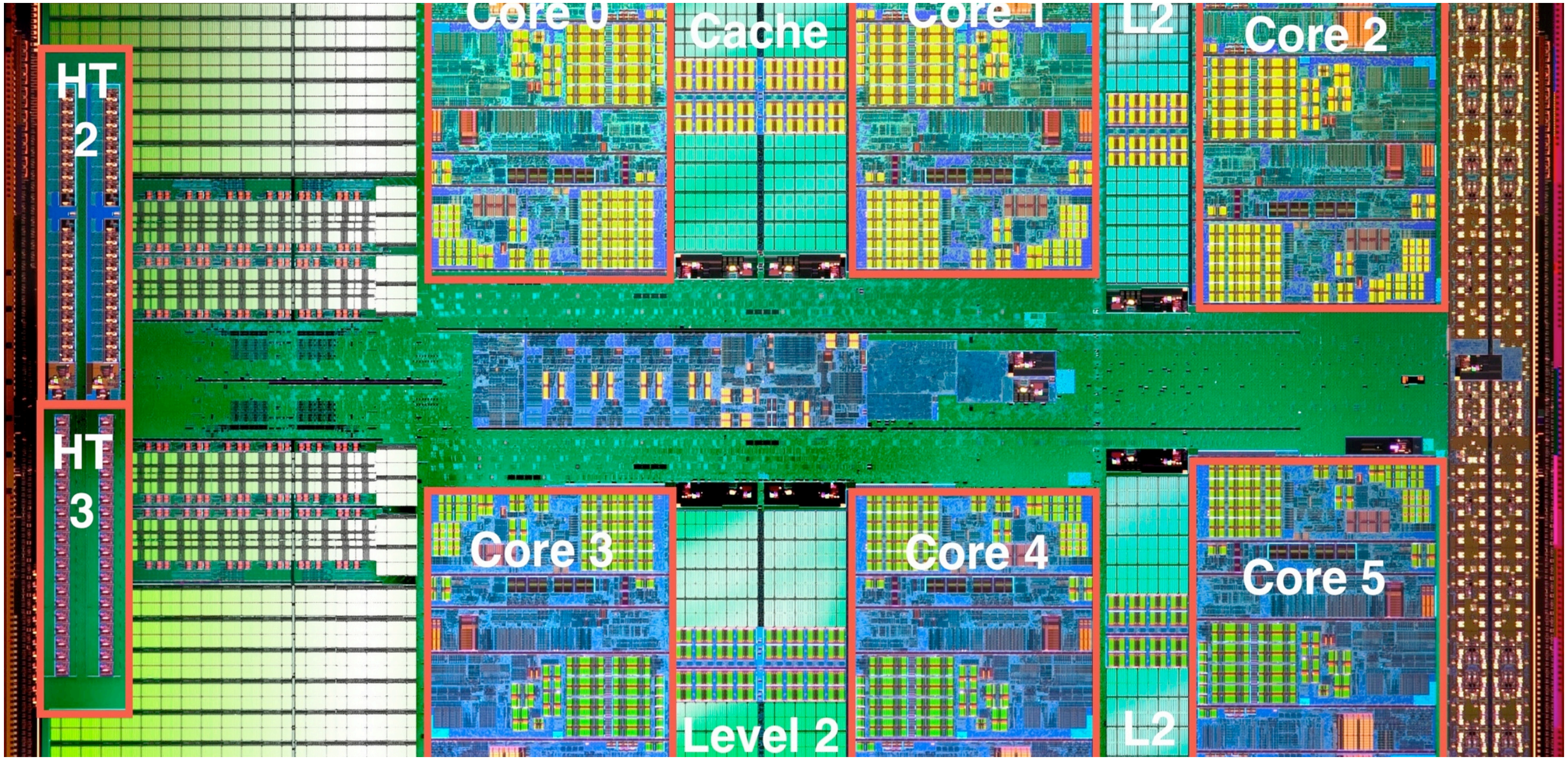
The Dawn of the Multicore Age: Using Refactoring and Skeletons to Generate Parallel Erlang Programs

Chris Brown

University of St Andrews
June 15th 2012

PARAPHRASE

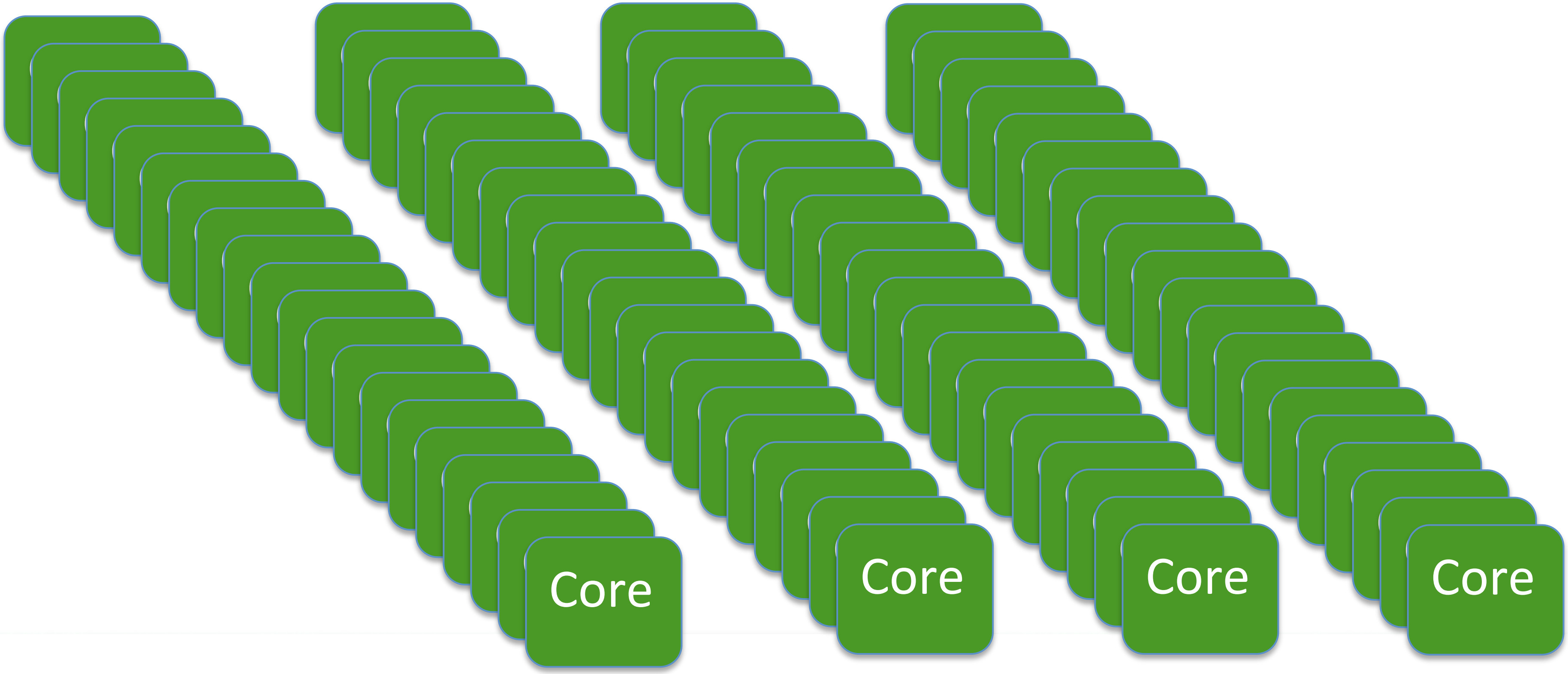
The Dawn of a New Multicore Age



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

The Future: "megacore" computers?

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



Programming Issues

- 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
    - *Concurrency* ***is not*** *parallelism!*

- Many approaches provide *libraries*
  - **they need to provide abstractions**

Patterns…

- **Patterns**
    - **Abstract** generalised expressions of common algorithms
        - Map, Fold, Function Composition, Divide and Conquer, etc.

…and Skeletons

- **Skeletons**
    - **Implementations** of patterns
        - Parallel Map, Task Farm, Workpool, etc.

Example Pattern: parallel Map

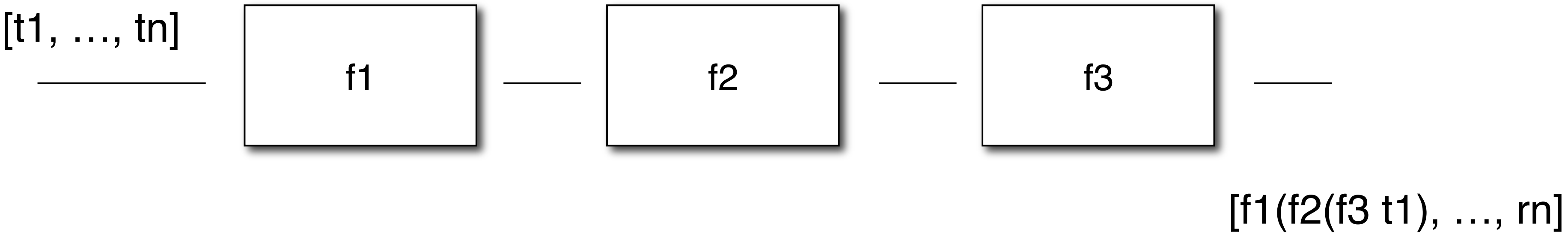
```
map(F, List) -> [ F(X) || X <- List ]
```

```
map(fun(X) -> X + 1 end, [ 1..10 ]) ->
                              [ 1 + 1, 2 + 1, ... 10 + 1 ]
```

map (Complexfun, Largelist) -> [Complexfun(X1), ...

Can be executed in parallel provided the results are *independent*

Skeleton: PipeLine



Skeletons in Erlang: PipeLine

```
pipe([F|Fs], Input) ->
    FirstStage = spawn(skeletons2, pipe_worker, [self(), F]),
    LastStage = pipe_rest(Fs, FirstStage),
    spawn(skeletons2, pipe_in, [Input, LastStage]),
    skeletons2:receive_output([]).
```

Skeletons in Erlang: PipeLine

```
pipe_rest([], LastStage) ->
    LastStage;
pipe_rest([F|Fs], PreviousStage) ->
    NextStage = spawn(skeletons2, pipe_worker,
                                        [PreviousStage, F]),
    pipe_rest(Fs, NextStage).
```

Skeletons in Erlang: PipeLine

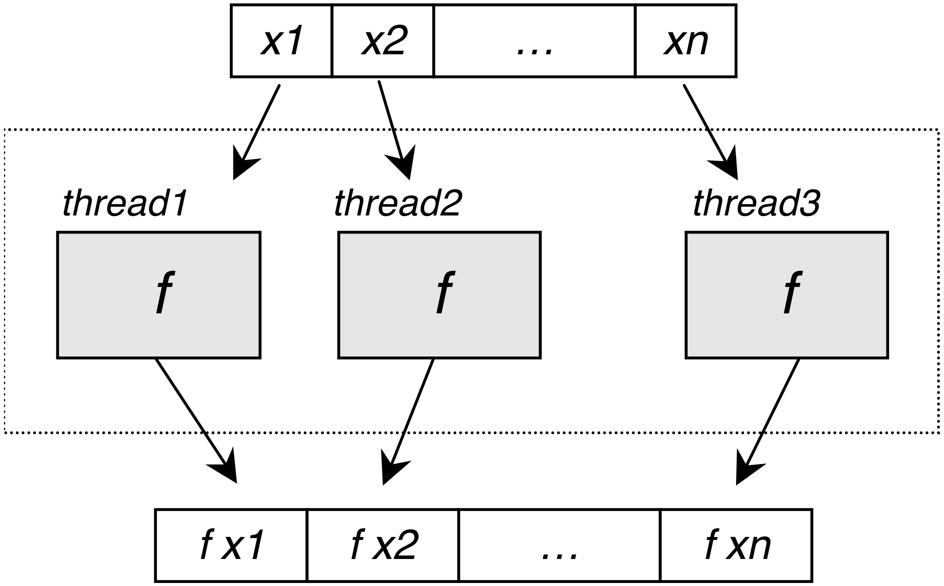
```
pipe_worker(Pid, Fun) ->
    receive
        {data, D} ->
            Pid ! {data, Fun(D)},
            pipe_worker(Pid, Fun);
        eod ->
            Pid ! eod
    end.
```

Skeletons in Erlang: PipeLine

```
pipe_in([], P2) ->  P2 ! eod;
pipe_in([H|Input], P2) ->
    P2 ! {data, H},
    pipe_in(Input, P2).

receive_output(Acc) ->
    receive
        {data, D} -> receive_output([D|Acc]);
        eod -> Acc
    end.
```

Example Skeleton: Data Parallel Map



Skeletons in Erlang: Data Parallel Map

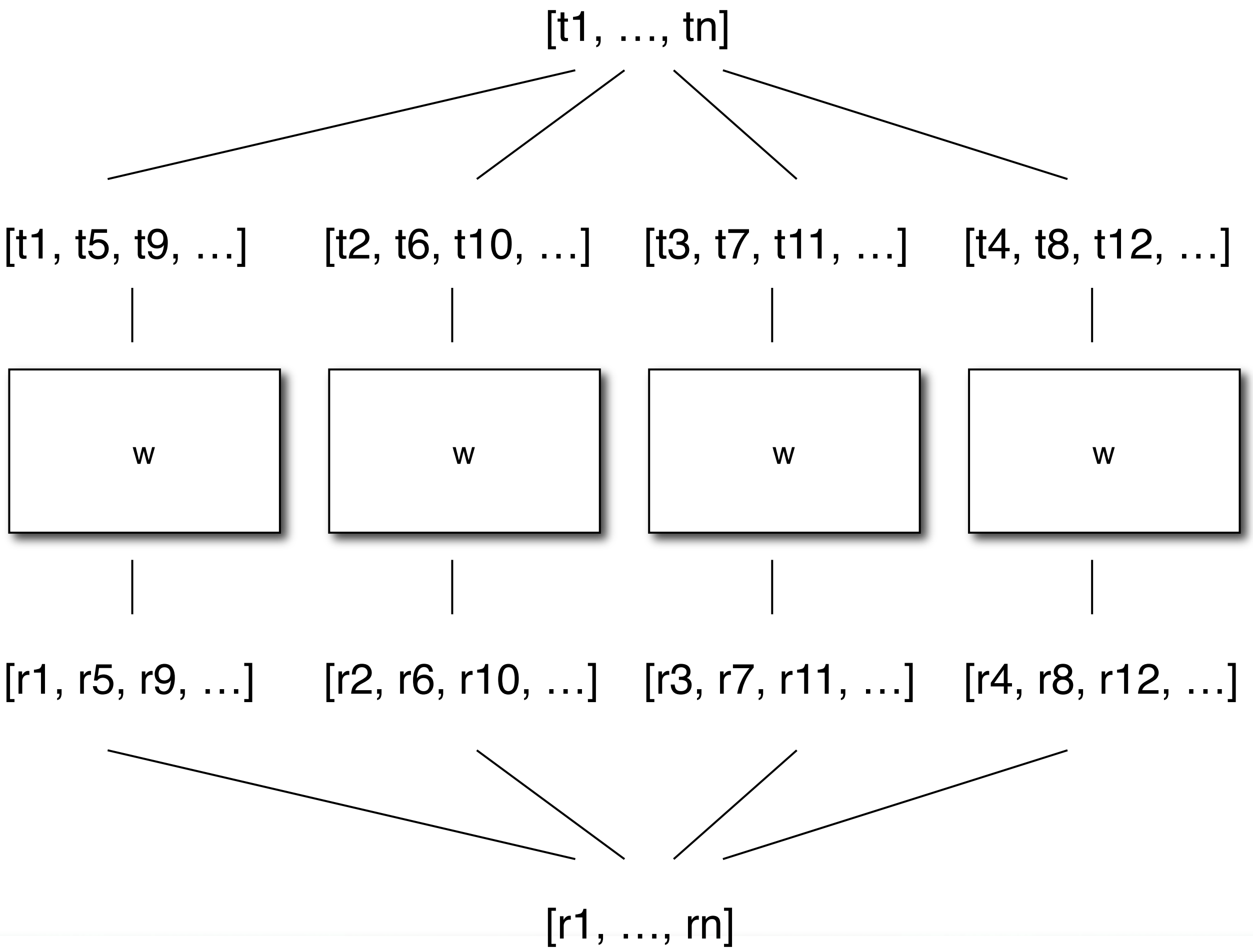
```
Parallel_map(F, List) ->
    lists:foreach(
      fun (Task) ->
              spawn(skeletons, worker, [self(), F, [Task]])
      end, List
     ),
    accumulate(length(List), []).
```

Skeletons in Erlang: Data Parallel Map

```
worker(Pid, F, [X]) ->
    Pid ! F(X).

accumulate(0, Result) ->
    Result;
accumulate(N, Result) ->
    receive
        Element ->
            accumulate(N-1, [Element|Result])
    end.
```

Skeleton: Task Farm



[t1, …, tn]

[t1, t5, t9, …]

[t2, t6, t10, …]

[t3, t7, t11, …]

[t4, t8, t12, …]

w

w

w

w

[r1, r5, r9, …]

[r2, r6, r10, …]

[r3, r7, r11, …]

[r4, r8, r12, …]

[r1, …, rn]

Skeletons in Erlang: Task Farm

```
make_farm(Fun, N, List) ->
    PC = spawn(skeletons,collector,[self(),N]),
    PWL= string(N,PC,Fun),
    spawn(skeletons,emitter,[PWL]).

emitter([W|RW], _, []) -> terminate_workers([W|RW]) ;
emitter([W|RW],Fun, [L|Ls]) ->
    W ! {data, apply(ff,Fun,[L])},
    emitter(lists:append(RW,[W]), Fun, Ls).
```

Skeletons in Erlang: Task Farm

```
string(N, Pid, Fun) -> string(N,Pid,Fun,[]).
string(0,_,_,L) -> L;
string(N,Pid,Fun,L) ->
              PP = spawn(skeletons,node,[Fun,Pid]),
             string((N-1),Pid,Fun,[PP|L]).

terminate_workers([]) -> [];
terminate_workers([W|RW]) ->
  W ! eos, terminate_workers(RW).
```

Skeletons in Erlang: Task Farm

```
node(WorkerFun, Pid) ->
  receive
    {data, D} ->
              Pid ! {data, apply(skeletons,WorkerFun,[D])},
                              node(WorkerFun, Pid);
    eos    ->  Pid ! eos end.
```

Skeletons in Erlang: Task Farm

```
collector(Fun, 1, Accum) ->
    receive
        {data, D} -> collector(Fun, 0, [D|Accum]);
        eos       -> Accum %% propagate eos
    end;
collector(Fun, N, Accum) ->
    receive
        {data, D} -> collector(Fun,N, [D|Accum]);
        eos       -> collector(Fun, (N-1), Accum)
    end.
```

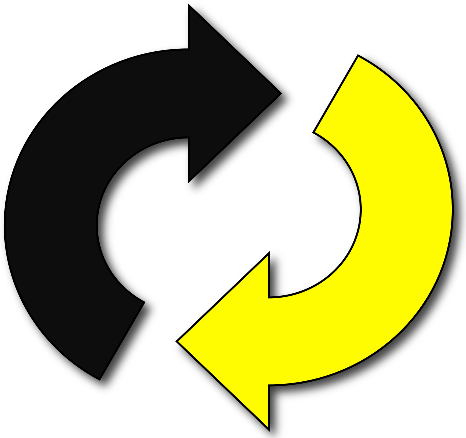
Thinking in Parallel

- 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!

Refactoring

- Refactoring is about **changing** the **structure** of a program’s **source code**

  … while **preserving** the semantics



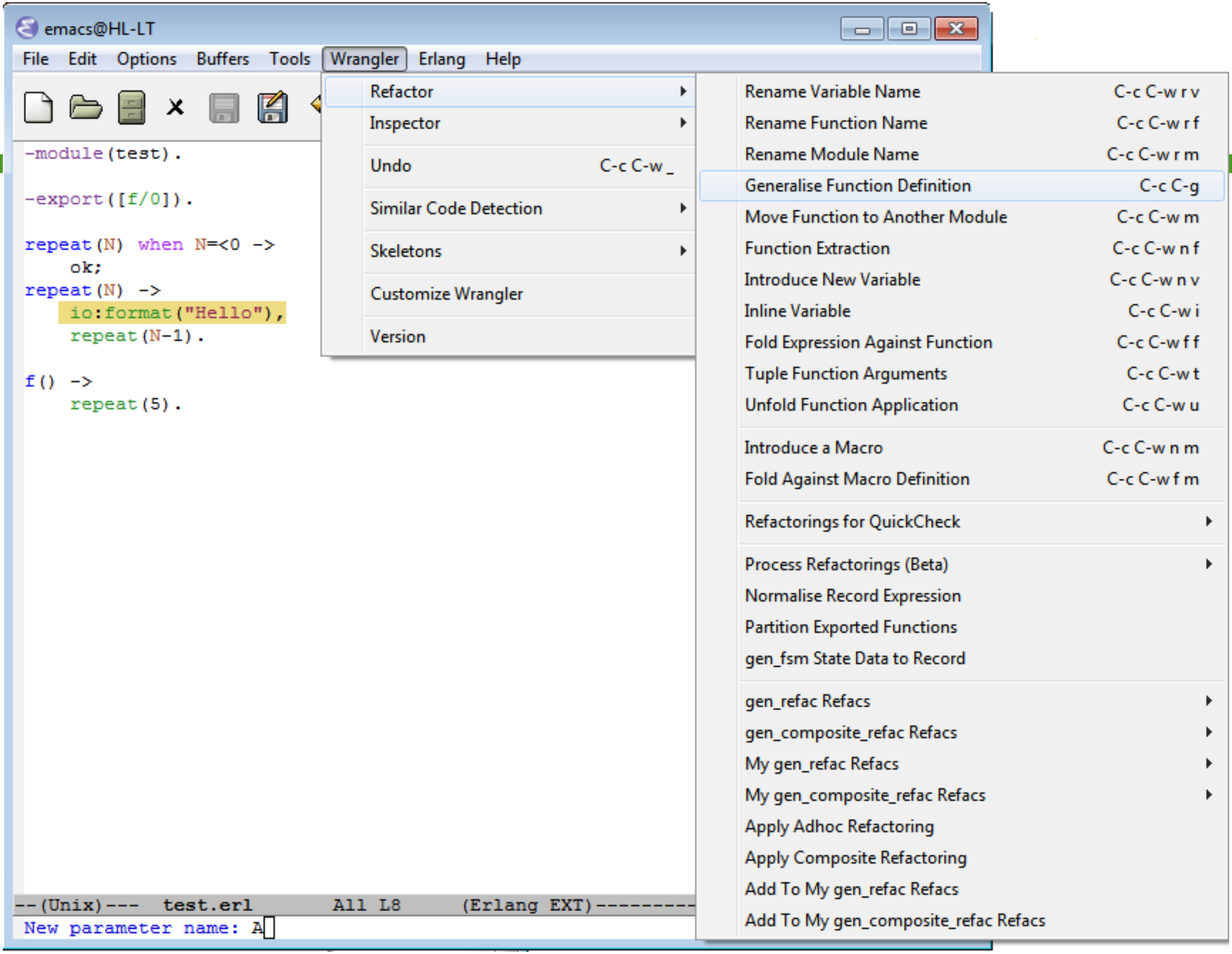
Review

Refactor

Refactoring = Condition + Transformation

Wrangler: the Erlang Refactorer

1. Developed at the University of Kent
    - **Simon Thompson and Huiqing Li**
2. Embedded in common IDEs: (X)Emacs, Eclipse.
3. Handles full Erlang language
4. Faithful to layout and comments
5. Undo
6. Built in Erlang, and applies to the tool itself



```
-module(test).

-export([f/0]).

repeat(N) when N=<0 ->
    ok;
repeat(N) ->
    io:format("Hello"),
    repeat(N-1).

f() ->
    repeat(5).
```

```
--(Unix)---   test.erl       All L8      (Erlang EXT)---------
New parameter name: A
```

Sequential Refactoring

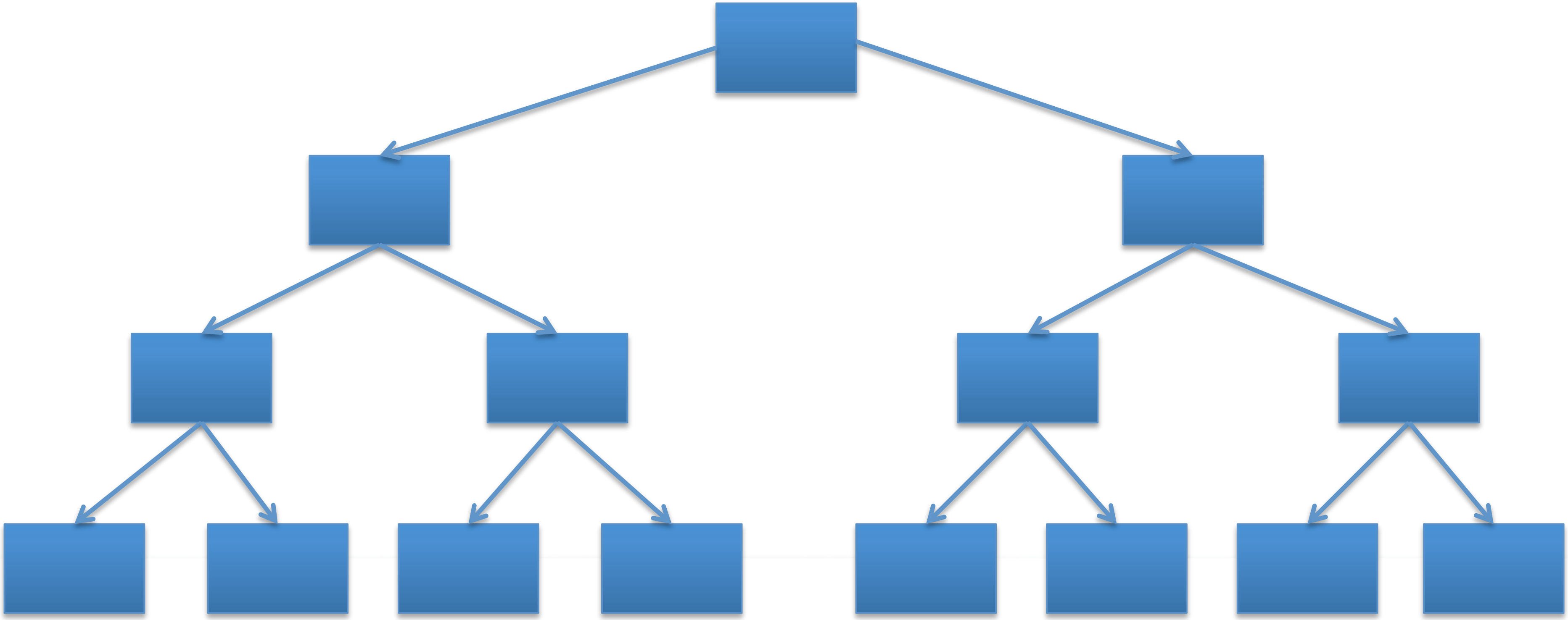
1. Renaming
2. Inlining
3. Changing scope
4. Adding arguments
5. Generalising Definitions
6. Type Changes

Parallel Refactoring!

- New approach to parallel programming
- Tool support allows programmers to *think in parallel*
  - Guides the programmer step by step
  - Database of transformations
  - Warning messages
  - Costing/profiling to give parallel guidance
- More structured than using e.g. Erlang **spawn** directly
  - Helps us get it "Just Right"

Classical Divide and Conquer

1. Split the input into N tasks
2. Compute over the N tasks
3. Combine the results



QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
    qsort([X || X <- T, X =< Pivot])
    ++ [Pivot] ++
    qsort([X || X <- T, X > Pivot]).
```

Introduce left branch as a local definition

QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
    L1 = qsort([X || X <- T, X =< Pivot]),
    L1 ++ [Pivot] ++
    qsort([X || X <- T, X > Pivot]).
```

QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
      L1 = qsort([X || X <- T, X =< Pivot]),
      L1 ++ [Pivot] ++
      qsort([X || X <- T, X > Pivot]).
```

Introduce right branch as a local definition

QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
      L1 = qsort([X || X <- T, X =< Pivot]),
      L2 = qsort([X || X <- T, X > Pivot]),
      L1 ++ [Pivot] ++ L2.
```

QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
     L1 = qsort([X || X <- T, X =< Pivot]),
     L2 = qsort([X || X <- T, X > Pivot]),
     L1 ++ [Pivot] ++ L2.
```

Introduce task parallelism for the left branch of the qsort...

QuickSort

```
qsort([]) -> [];
qsort([Pivot|T]) ->
    L1 = qsort([X || X <- T, X =< Pivot]),
    L2 = qsort([X || X <- T, X > Pivot]),
    spawn(?MODULE,qsort_worker,
                        [self(),1,L1],
    S1 = receive {1,R1} -> R1 end,
   S1 ++ [Pivot] ++ L2.
```

QuickSort

```
qsort_worker(Pid, l L) ->
    Pid ! {l, qsort(L)};

qsort_worker(Pid, r, R) ->
    Pid ! {r, qsort(R)}.
```

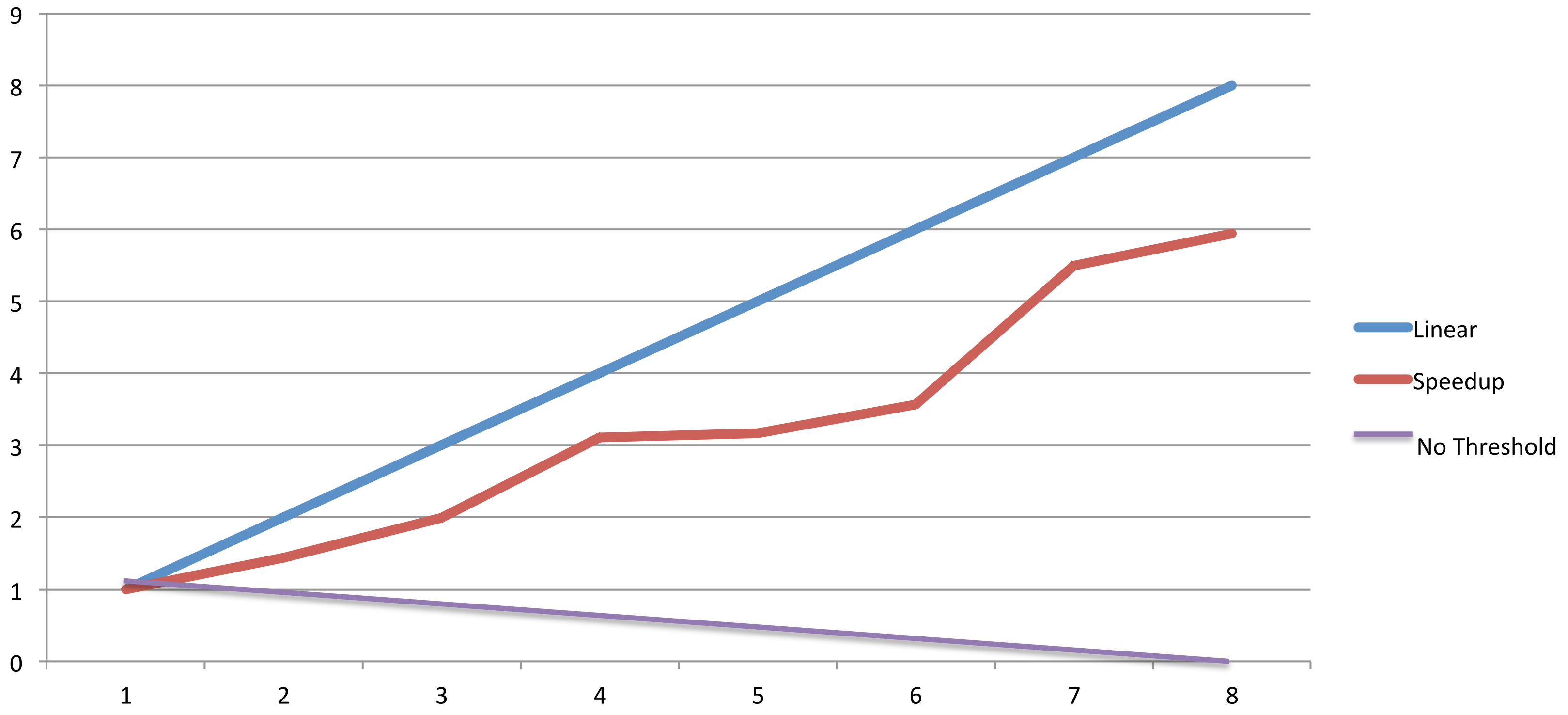
QuickSort

do the same for the right branch

…

```
qsort([]) -> [];
qsort([Pivot|T]) ->
    L1 = qsort([X || X <- T, X =< Pivot]),
    L2 = qsort([X || X <- T, X > Pivot]),
    spawn(?MODULE,qsort_worker,
                          [self(), l, L1],
    spawn(?MODULE,qsort_worker,
                          [self(), r, R1],
    S1 = receive {l, R1} -> R1 end,
    S2 = receive {r, R2} -> R2 end,
    S1 ++ [Pivot] ++ S2.
```

Speedups for quicksort



Conclusions

- Erlang has explicit concurrency:
  - Perfect for giving control for parallelism
  - Low level
  - Parallelism needs abstractions (Skeletons)
- Refactoring tool support:
  - Enhances creativity
  - Guides a programmer through steps to achieve parallelism
  - Warns the user if they are going wrong
  - Avoids common pitfalls
  - Helps with understanding and intuition

Future Work

- More Erlang skeletons
- More parallel refactorings
- Database of parallel skeleton templates
- Refactoring language (DSL) for expressing transformations + conditions
    - Language for expressing patterns?
- Cost directed refactoring
- Prove that refactorings improve parallelism

Funded by

- 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

Industrial Connections

SAP GmbH, Karlsrühe

BAe Systems

Selex Galileo

Biold GmbH, Stuttgart

Philips Healthcare

Software Competence Centre, Hagenberg

Mellanox Inc.

Erlang Solutions Ltd

Microsoft Research

Well-Typed

THANK YOU!

http://www.paraphrase-ict.eu

@paraphrase_fp7