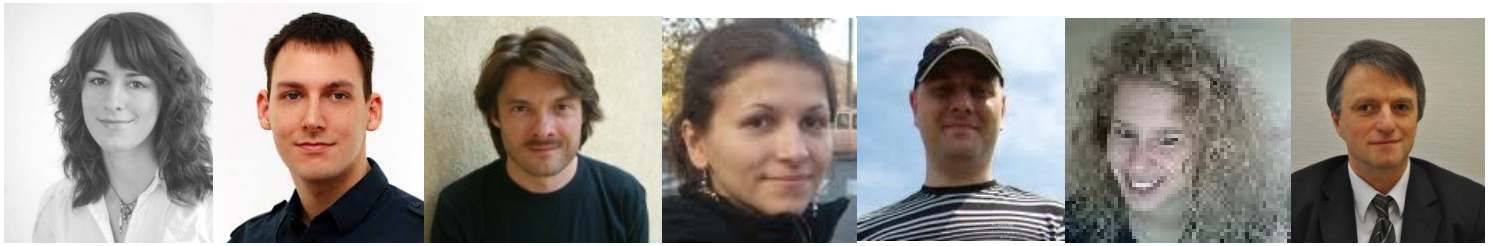
Parallelization by Refactoring

Dept. Programming Languages and Compilers
Eötvös Loránd University, Hungary



Judit Kőszegi
Dániel Horpácsi
Tamás Kozsik
Melinda Tóth
István Bozó
Viktória Fördős
Zoltán Horváth

LET'S PARTY!

- Have fun...

LET'S PaRTE!

- Have fun...

LET'S PaRTE!

- Have functional programming!

Kozsik, T. et al.: Parallelization by Refactoring

LET'S PaRTE!

- Have functional programming!

map-like function speedup prediction

pattern candidate discovery

static analysis task farm

refactoring pipeline parallel patterns

algorithmic skeletons divide and conquer

ParaPhrase Refactoring Tool for Erlang

RefactorErl Wrangler

Kozsik, T. et al.: Parallelization by Refactoring

Motivation

- Highly heterogeneous mega-core computers
- Performance and energy
- Think parallel
  - High-level programming constructs
  - Deadlocks etc. eliminated by design
  - Communication packaged/abstracted
  - Performance information is part of design
- Restructure existing code

Kevin Hammond

Kozsik, T. et al.: Parallelization by Refactoring

How shall I parallelize?

- Refactor
  - Use a tool!
  - Guided, semi-automatic transformations
- Experiment
  - Measure, validate
- Repeat
- Applicable for legacy code as well

Kozsik, T. et al.: Parallelization by Refactoring

Where shall I parallelize?

- Independent computations
- Good potential for speedup
  - Complex computation?
  - Low sequential overhead?
- Find candidates automatically
  - Use a tool!
  - Static program analysis

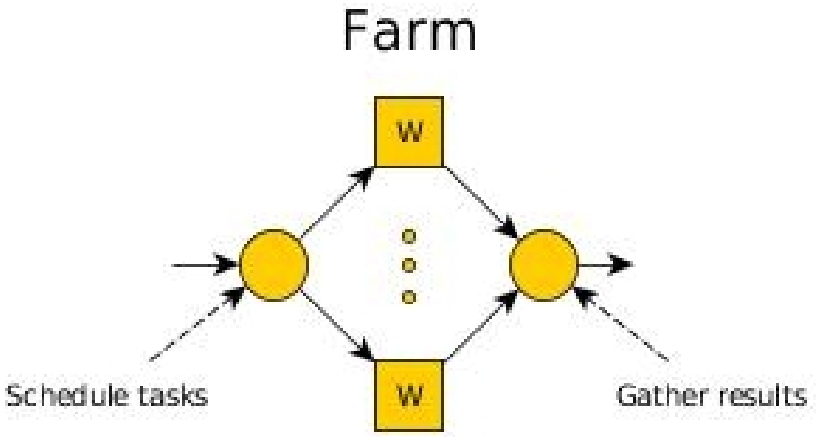
Kozsik, T. et al.: Parallelization by Refactoring

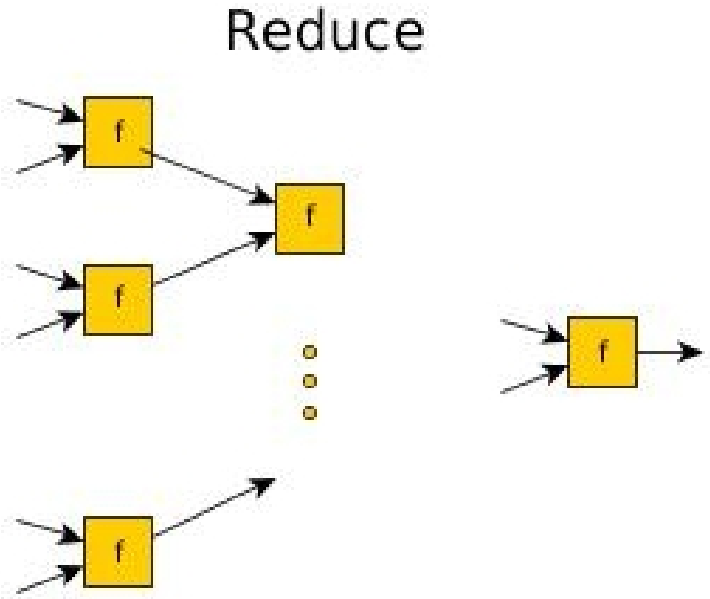
Pattern-based parallelism

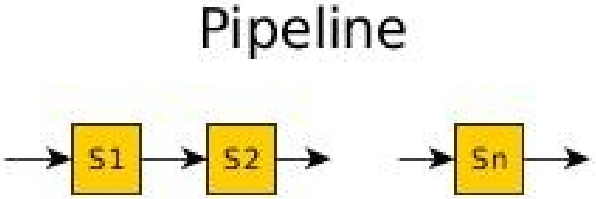
High-level approach to parallel programming

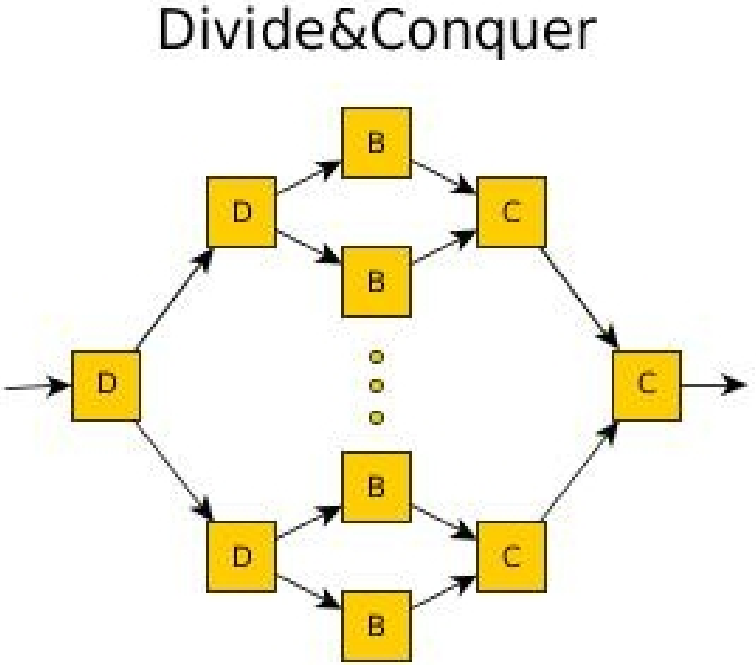
- Rely on a library of algorithmic skeletons
- Easier to develop code
- Easier to modify / maintain
- Better utilization of resources
  - Static resource management
  - Dynamic resource management

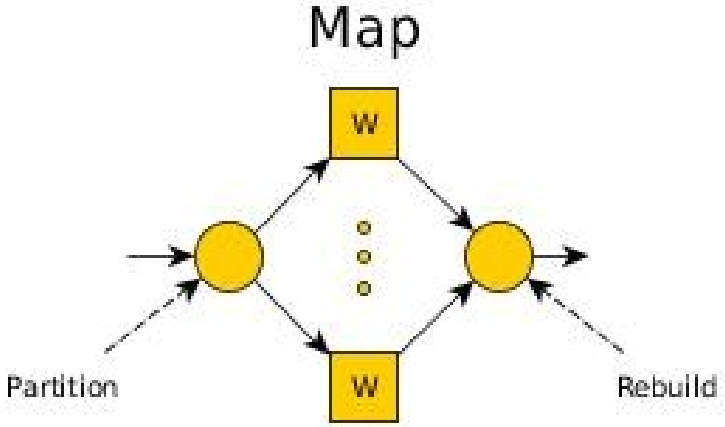
Kozsik, T. et al.: Parallelization by Refactoring





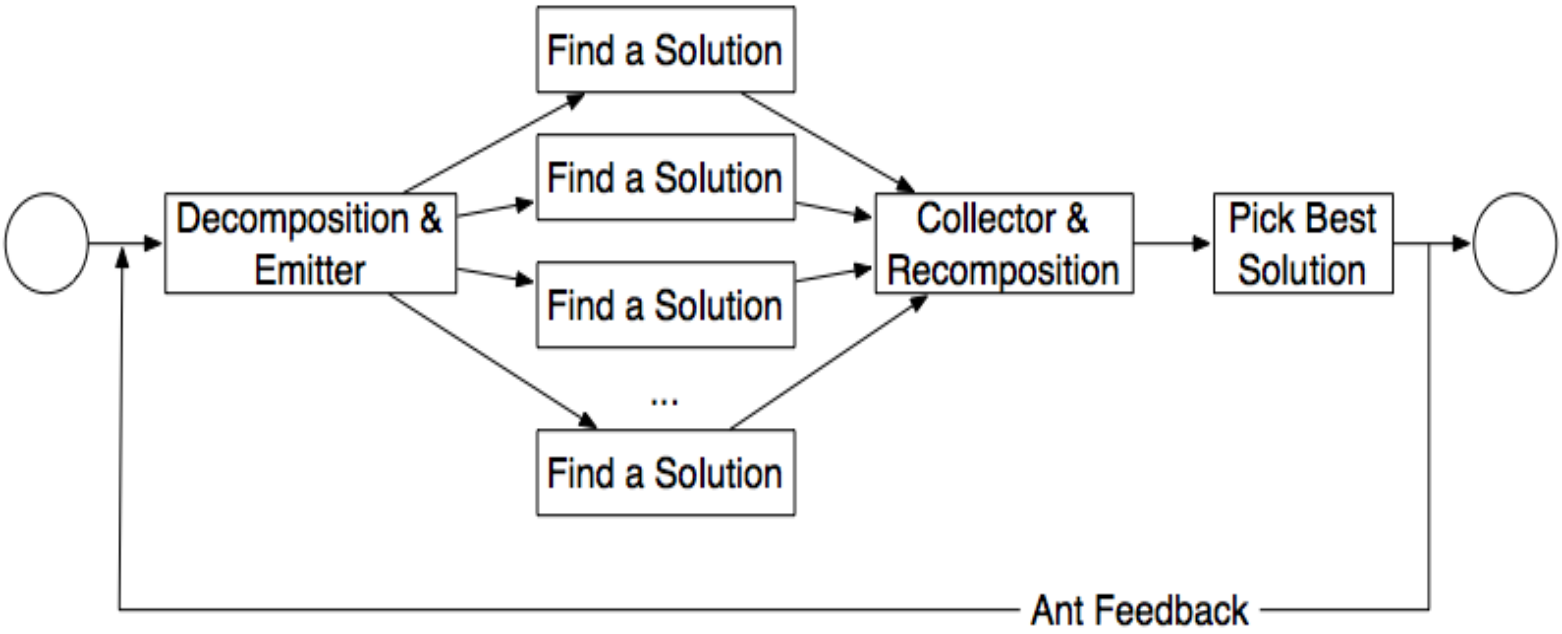






Kozsik, T. et al.: Parallelization by Refactoring

Pool pattern



Ant Colony Optimization

Kozsik, T. et al.: Parallelization by Refactoring

picture from Kevin Hammond

Kozsik, T. et al.: Parallelization by Refactoring

Parallel Patterns for Adaptive Heterogeneous Multicore Systems

- Programmability of heterogeneous parallel architectures
- Structured design and implementation of parallelism
- High-level parallel patterns
- Dynamic (re)mapping on heterogeneous hardware

Kozsik, T. et al.: Parallelization by Refactoring

University of St Andrews

ROBERT GORDON UNIVERSITY·ABERDEEN

National College of Ireland

Cloud Competency Center

ELTE Soft

Queen's University Belfast

AGH

AGH University PL

Erlang SOLUTIONS

UNIVERSITÀ DEGLI STUDI DI TORINO

ALMA UNIVERSITAS TAURINENSIS

UNIVERSITÀ DI PISA

Universität Stuttgart

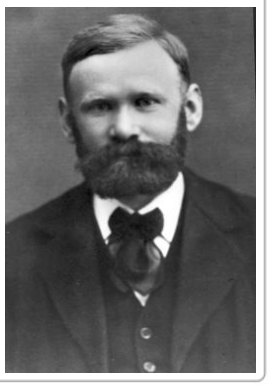
HLRS

Mellanox TECHNOLOGIES

scch software competence center hagenberg

Kozsik, T. et al.: Parallelization by Refactoring

- functional language
- strict, impure, dynamically typed
- concurrency, actor model
- distribution
- fault tolerance
- Open Telecom Platform



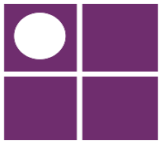
Kozsik, T. et al.: Parallelization by Refactoring

Companies using Erlang

SMARKETS

Kozsik, T. et al.: Parallelization by Refactoring

Products using Erlang



Kozsik, T. et al.: Parallelization by Refactoring

Let's PaRTE!

Assist parallelization of Erlang code with the

ParaPhrase Refactoring Tool for Erlang

Kozsik, T. et al.: Parallelization by Refactoring

So what is Erlang like?

- Functional (strict, impure, dynamically typed)
- Garbage collected
- Designed for concurrency and distribution
  - Lightweight processes
  - Actor model (message passing)
- Processing binary data
- Fault tolerance (failure recovery)
- Compiles to **`beam`**, runs in VM
- Hot code swap

Kozsik, T. et al.: Parallelization by Refactoring

Erlang: a functional language

- Variables bound only once
- Recursion instead of loops
  - Tail recursive
- Higher-order functions instead of recursion
- Lambdas, pattern matching
- Properties
  - Strict evaluation (call-by-value)
  - Impurity (e.g. communication)
  - No partial applications

Kozsik, T. et al.: Parallelization by Refactoring



Terms

- Literals, e.g. `42` or `42.0`
- Atoms, e.g. `leaf`, `blue`, `ok`, `error`, `true`
- Compound data

Funs	`fun(X)-> X+1 end`
Lists	`[0,1,1,2,3,5,8,13,21]`
Tuples	`{may, 10, 2014}`
Records	`#date{month=june,day=7,year=2015}`
Binaries	`<<0,1,1,2,3,5,8,13,21>>`

- Pids, ports, refs

Kozsik, T. et al.: Parallelization by Refactoring



A complex term

```
[
    {
        farm,
        [
            {
                pipe,
                [
                    {seq, fun scan/1},
                    {seq, fun parse/1}
                ]
            }
        ],
        12
    }
]
```

Kozsik, T. et al.: Parallelization by Refactoring

Expressions

- Terms (literals, atoms, compound data)
- Variables, e.g. `X`, `Long_Variable_Name`
- Function/operator calls, e.g. `fib(N-1)+fib(N-2)`
- Data structures, e.g. `{june,Day,fib(18)-569}`
- Control structures
  - Branching (**`case`** and **`if`**)
  - Sending and receiving a message
  - Error handling

Kozsik, T. et al.: Parallelization by Refactoring

Defining a named function

```
increment(N) -> N+1.
```

Kozsik, T. et al.: Parallelization by Refactoring

Pattern matching

```
fib(N) ->
    case N of
        0 -> 0;
        1 -> 1;
        _ -> fib(N-1) + fib(N-2)
    end.
```

Kozsik, T. et al.: Parallelization by Refactoring

Multiple function clauses

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

Kozsik, T. et al.: Parallelization by Refactoring

Guards

```
fib(N) when N < 2 -> N;
fib(N) -> fib(N-1) + fib(N-2).
```

Kozsik, T. et al.: Parallelization by Refactoring

Using an `if`-expression

```
fib(N) ->
    if
        N < 2 -> N;
        true  -> fib(N-1) + fib(N-2)
    end.
```

Kozsik, T. et al.: Parallelization by Refactoring

Recursion

```
factorial(1) -> 1;
factorial(N) -> N * factorial(N-1).
```

Kozsik, T. et al.: Parallelization by Refactoring


Tail recursion

```
factorial(1) -> 1;
factorial(N) -> N * factorial(N-1).
```

```
factorial(N) -> factorial_acc(N,1).

factorial_acc(1,Acc) ->
  Acc;
factorial_acc(N,Acc) ->
  factorial_acc(N-1, Acc*N).
```

Kozsik, T. et al.: Parallelization by Refactoring



Overloading on arity

```
prime(1) -> false;
prime(N) when N > 1 -> prime(N,2).

% no proper divisors of N
% between M and sqrt(N)

prime(N,M) ->
   M*M>N orelse ( N rem M =/= 0 andalso
                  prime(N,M+1)
                ).
```

- Overloaded functions `prime/1` and `prime/2`
- `prime/2` is helper function

Kozsik, T. et al.: Parallelization by Refactoring



Lists

```
[0,1,1,2,3,5,8]
[0 | [1,1,2,3,5,8]]
[0 | [1 | [1 | [2 | [3 | [5 | [8 | []]]]]]]]
[0,1,1,2 | [3 | [5 | [8 | []]]]]
[0,1,1,2 | [3,5,8]]
```

- `[Head | Tail]` notation
- List comprehension

```
[N || N <- lists:seq(1,100), prime(N)]
```

Kozsik, T. et al.: Parallelization by Refactoring

Scalar-vector multiplication

```
mul(Scalar,[]) -> [];
mul(Scalar,[Head|Tail]) ->
   [ Scalar*Head | mul(Scalar,Tail) ].
```

Kozsik, T. et al.: Parallelization by Refactoring



Scalar-vector multiplication

```
mul(Scalar,[]) -> [];
mul(Scalar,[Head|Tail]) ->
   [ Scalar*Head | mul(Scalar,Tail) ].
```

```
mul(Scalar,List) ->
   [ Scalar*Item || Item <- List ].
```

Kozsik, T. et al.: Parallelization by Refactoring

Scalar-vector multiplication

```
mul(Scalar,[]) -> [];
mul(Scalar,[Head|Tail]) ->
   [ Scalar*Head | mul(Scalar,Tail) ].
```

```
mul(Scalar,List) ->
   [ Scalar*Item || Item <- List ].
```

```
mul(Scalar,List) ->
   map( fun(Item) -> Scalar*Item end, List ).
```

Kozsik, T. et al.: Parallelization by Refactoring

Higher-order functions

```
map(Fun,[]) -> [];
map(Fun,[Head|Tail]) ->
   [ Fun(Head) | map(Fun,Tail) ].

filter(Pred,List) ->
   [ Item || Item <- List, Pred(Item) ]
```

Kozsik, T. et al.: Parallelization by Refactoring

fun-expressions

```
map(  fun(Item) -> Scalar*Item end, List )
```

```
filter( fun prime/1, List )
```

Kozsik, T. et al.: Parallelization by Refactoring

Variable binding

- Formal parameters:

```
fib(N)-> ...
mul(Scalar,[Head|Tail])-> ...
```

- Generator in list comprehension:

```
[ ... | Item <- List]
```

- Pattern matching expression

  Syntax: `Pattern = Expression`

  Primes = primes(List)

  [Head | Tail] = primes(List)

Kozsik, T. et al.: Parallelization by Refactoring

Sequence of expressions

```
area( {square, Side} ) ->
    Side * Side;

area( {circle, Radius} ) ->
    % almost :-)
    3.14 * Radius * Radius;

area( {triangle, A, B, C} ) ->
    S = (A + B + C)/2,
    math:sqrt(S*(S-A)*(S-B)*(S-C)).
```

- from http://www.erlang.org/course/course.html
- Grouping with **`begin`** ... **`end`**

Kozsik, T. et al.: Parallelization by Refactoring

Modules

- Code in `.hrl` and `.erl` files
- Compilation unit: *module*
- Modules contain *forms*
  (e.g. function and macro defs)

Kozsik, T. et al.: Parallelization by Refactoring

Example: mymath.erl

```
-module(mymath).

-export([fib/1,prime/1,pi/0]).

-define(PI,3.14).

pi() -> ?PI.

fib(N) when N<2 -> N;
fib(N) -> fib(N-1) + fib(N-2).

prime(1) -> false;
prime(N) when N > 1 -> prime(N,2).

prime(N,M) when M*M > N -> true;
prime(N,M) when N rem M == 0 -> false;
prime(N,M) -> prime(N,M+1).
```

Kozsik, T. et al.: Parallelization by Refactoring

Calling functions

- Full name of exported functions:
  `mymath:prime/1`
- They can be called from other modules:
  `mymath:prime(1987)`
- They can be imported and called without module prefix
- Many built-in functions (BIFs) are auto-imported
- Calling through a variable:
  `F = fun mymath:prime/1, F(1987)`
- Dynamic call: `apply(Module,Function,Args)`

Kozsik, T. et al.: Parallelization by Refactoring

Compiling and running

```
$ ls mymath.erl
mymath.erl
$ erl
Erlang R16B (erts-5.10.1) [source] [smp:4:4]
[async-threads:10] [hipe] [kernel-poll:false]

Eshell V5.10.1 (abort with ^G)
1> c(mymath).
{ok,mymath}
2> mymath:prime(1987).
true
3> q().
ok
4> $ ls mymath*
mymath.beam mymath.erl
```

Kozsik, T. et al.: Parallelization by Refactoring

What else?

- Concurrency, distributed programming
  - processes, ports, nodes
  - sending and receiving messages
- Exception handling
- Standard libraries, OTP
- Programming patterns (behaviors)

Kozsik, T. et al.: Parallelization by Refactoring

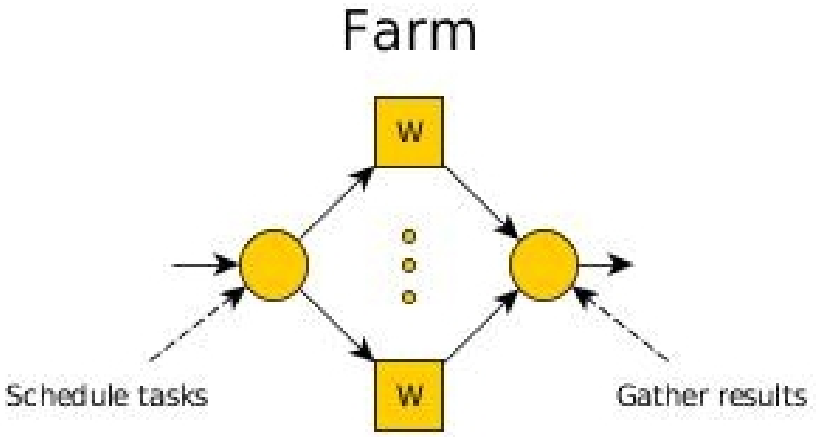
Coming back to PaRTE...

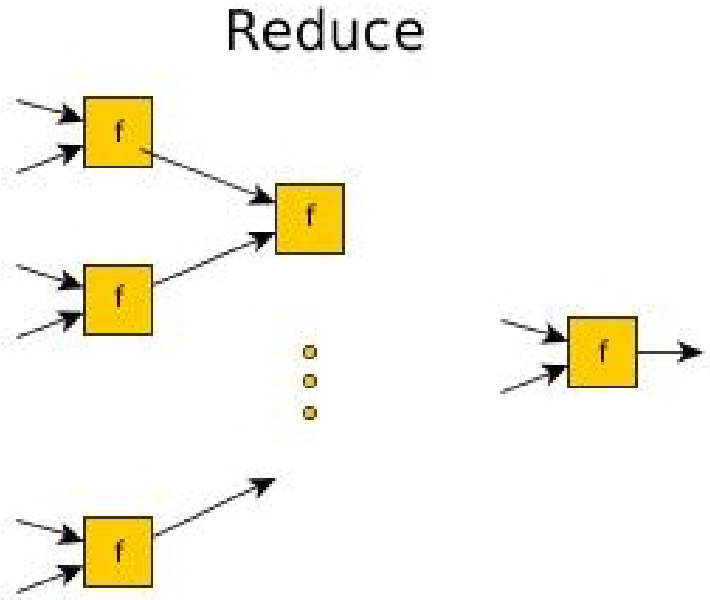
Assist parallelization of Erlang code with the

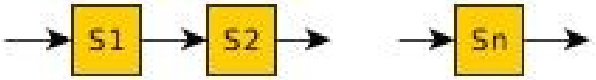
ParaPhrase Refactoring Tool for Erlang

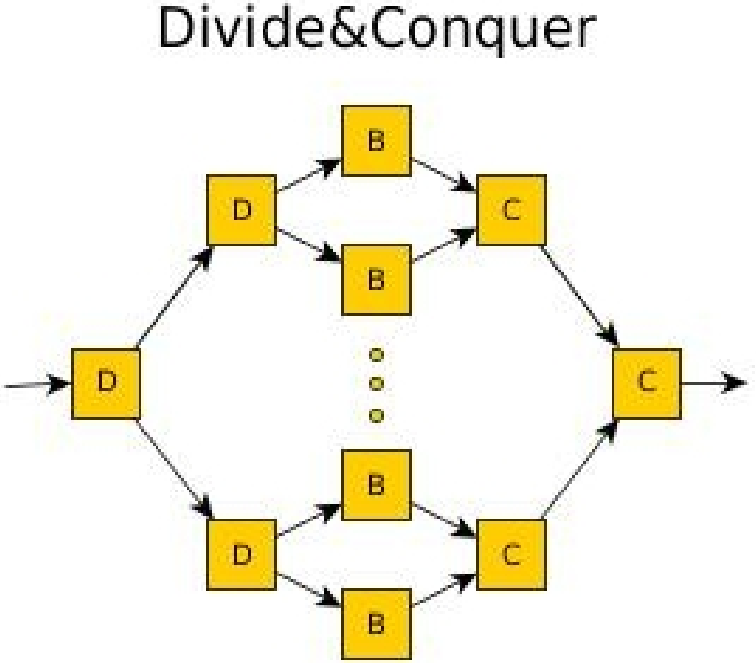
- Discover parallelizable code fragments
- Predict speedup, make suggestions
- Perform guided automated refactorings
- Pattern-based parallelism

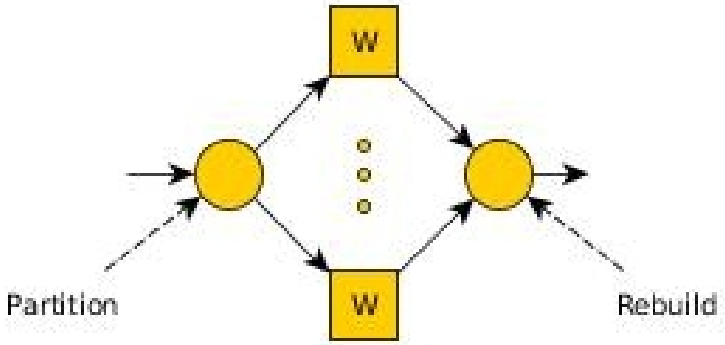
Kozsik, T. et al.: Parallelization by Refactoring











Kozsik, T. et al.: Parallelization by Refactoring

Parallel skeletons

http://paraphrase-ict.eu/Deliverables/deliverable-2.6

The **skel** library

- Basic algorithmic skeletons
  farm, pipe, map, reduce, ord, feedback etc.
- High-level patterns: skel hlp
  dc, evolutionPool etc.
- Heterogeneous skeletons: Lapedo
  OpenCL kernels for CPU and GPU

  http://paraphrase-ict.eu/Deliverables/d27prototype.tar.gz

Kozsik, T. et al.: Parallelization by Refactoring

Example: parsing modules

```
[ parse ( scan ( read ( Module ) ) )
                    || Module <- Modules ]
```

Kozsik, T. et al.: Parallelization by Refactoring



Example: parsing modules

```
[ parse ( scan ( read ( Module ) ) )
                        || Module <- Modules ]


skel:do([
  { farm, [{ pipe, [ { seq, fun read/1  },
                     { seq, fun scan/1  },
                     { seq, fun parse/1 }
                   ] }
           ], 5 }
  ], Modules )
```

Kozsik, T. et al.: Parallelization by Refactoring



Example: radix sort

```
sort( List ) -> sort(List,0).

sort( List,   _    ) when length(List) < 2 ->
   List;

sort( List, Level ) ->
   lists:append(
      [ sort( Bucket, Level+1 )
        || Bucket <- divide( List, Level )
      ]
   ).

divide( List, Level ) -> ...
```

Kozsik, T. et al.: Parallelization by Refactoring

Divide-and-conquer pattern

```
{ dc, IsBase, BaseFun, Divide, Combine }


{ dc, IsBase, BaseFun, Divide, Combine,
  MaxProcesses }


SEQ_DC =
{ seq_dc, IsBase, BaseFun, Divide, Combine },
PAR_DC =
{ dc, IsSeq, SEQ_DC, Divide, Combine }
```

Kozsik, T. et al.: Parallelization by Refactoring

Example: radix sort

```
sort( List ) -> skel:do( [{ dc,
      fun({Lst,Level}) -> length(Lst) < 2 end,
      fun({Lst,Level}) -> Lst end,
      fun({Lst,Level}) ->
            [ {Bucket,Level+1}
              || Bucket <- divide(Lst, Level)
            ]
         end,
      fun lists:append/1
  }], {List,0}).
```

Kozsik, T. et al.: Parallelization by Refactoring

Example: radix sort

```
sort( List ) -> sk_hlp:dc(
      fun({Lst,Level}) -> length(Lst) < 2 end,
      fun({Lst,Level}) -> Lst end,
      fun({Lst,Level}) ->
            [ {Bucket,Level+1}
              || Bucket <- divide(Lst, Level)
            ]
         end,
      fun lists:append/1
   )  ({List,0}).
```

Kozsik, T. et al.: Parallelization by Refactoring

ParaPhrase approach

- Identify (strongly hygienic) components
- Discover patterns of parallelism
- Structure the components into a parallel program
  - Turn the patterns into concrete code (skeletons)
  - Take performance, energy etc. into account
- Restructure if necessary
- Use a refactoring tool

Kozsik, T. et al.: Parallelization by Refactoring

Effectiveness of the approach

Parallelization	Manual	ParaPhrasing
Convolution	3 days	3 hours
Ant Colony	1 day	1 hour
BasicN2	5 days	5 hours
Graphical Lasso	12 hours	2 hours

Kozsik, T. et al.: Parallelization by Refactoring

PaRTE

ParaPhrase Refactoring Tool for Erlang

- Locate parallel pattern candidates
- Estimate speedup for different configurations
- Advise programmer
- Assist with refactoring
- Enforce preservation of functionality

Kozsik, T. et al.: Parallelization by Refactoring

Googling the code for patterns

Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Expected speedup (GPU)	Recommended?
1	(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99	1,00	✓
2	(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80	1,00	✓
6	(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58	1,00	✓
3	(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58	1,00	✓
5	(Δe292)	matrix_ex_paper	mult_matrix2	2	12	2,98	1,00	✓
4	(Δe337)	matrix_ex_paper	scalar_product	2	12	1,06	1,00	✓

Chart options

Apply selected transformations

Details of the transformation sequence

Configuration	Location information	Program text	Number of workers	Sequential CPU time	Sequential GPU time	Parallel CPU time	Parallel GPU time	Expected speedup (CPU)	Expected speedup (GPU)	Used stream length
e337	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,15},{18,25}} - {{18, 30}, {18, 30}}	mult_scalar(A,B)	1	0,14	0,00	0,14	0,00	1,00	1,00	1
(Δe337)	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,13},{18,13}} - {{19, 39}, {19, 39}}	[mult_scalar(A,B) || {A,B} <- lists:zip(R,C)]	1	1 375,42	0,00	2 506,26	0,00	0,55	1,00	10 000
(Δ(Δe337))	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{6,3},{6,3}} - {{7, 26}, {7, 26}}	[scalar_product(R,C) || R <- Rows, C <- Cols]	12	13 754 154,08	0,00	2 091 407,67	0,00	6,58	1,00	10 000

Chart options

Kozsik, T. et al.: Parallelization by Refactoring

ern Candidate Browser

nsformation sequences

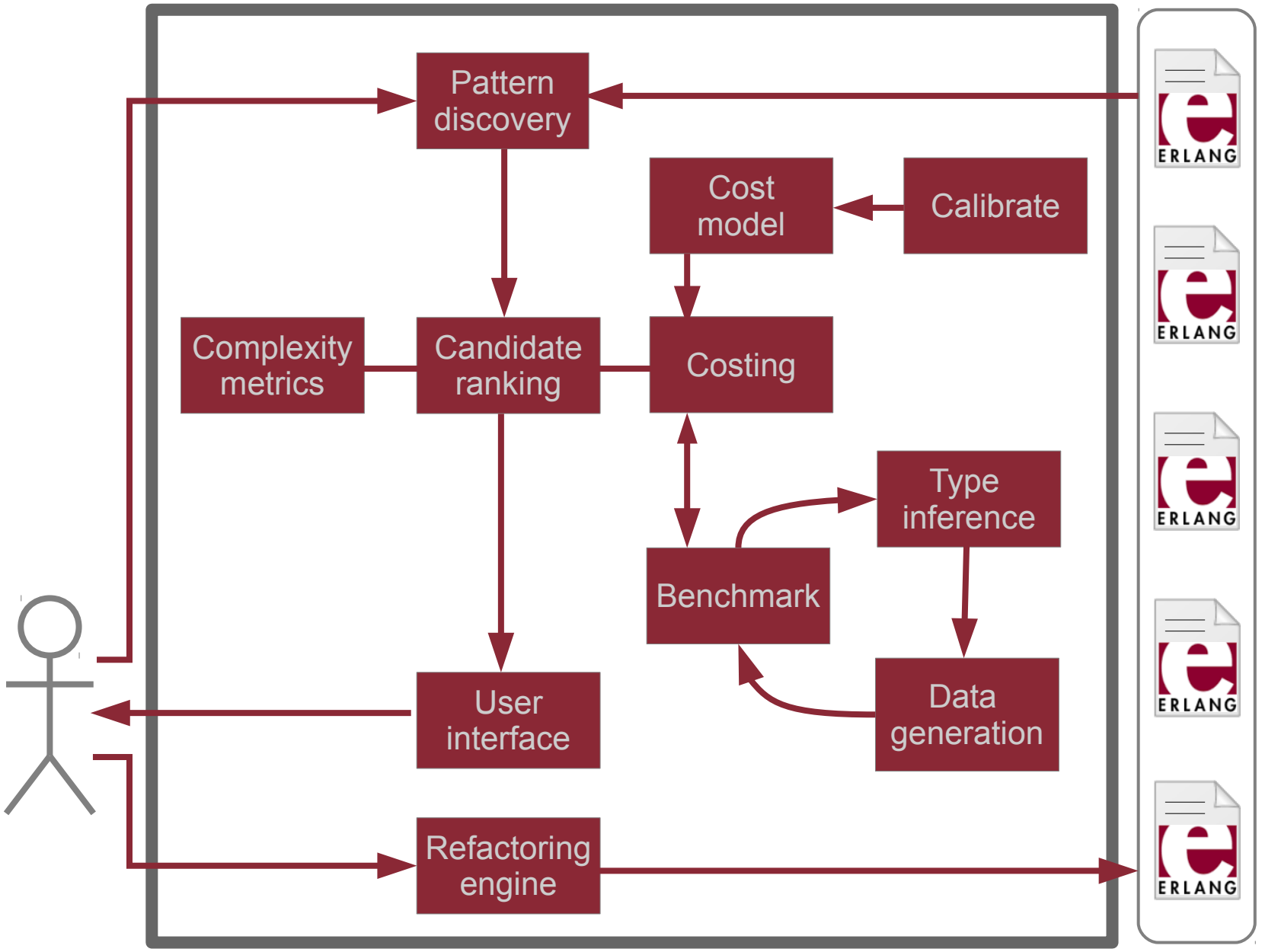
Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)
(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99
(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80
(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58
(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58
(Δe292)	matrix_ex_paper	mult_matrix2	2	12	2,98
(Δe337)	matrix_ex_paper	scalar_product	2	12	1,06

t options

ls of the transformation sequence

guration	Location information	Program text	Number of workers	Sequential CPU time	Sequential GPU time	Para CPU t
	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,15},{18,25}} - {{18, 30}, {18, 30}}	mult_scalar(A,B)	1	0,14	0,00	
7)	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{18,13},{18,13}} - {{19, 39}, {19, 39}}	[mult_scalar(A,B) || {A,B} <- lists:zip(R,C)]	1	1 375,42	0,00	2
337))	/Users/V/paraphrase/referl/tool/matrix/matrix_ex_paper.erl : {{6,3},{6,3}} - {{7, 26}, {7, 26}}	[scalar_product(R,C) || R <- Rows, C <- Cols]	12	13 754 154,08	0,00	2 091

t options



Kozsik, T. et al.: Parallelization by Refactoring

Pattern candidate discovery

- Collect syntactic & semantic information
  - List comprehensions
  - Library calls (`lists:map/2`)
  - Recursion structure
  - Side conditions
- Task farm, pipeline, divide-and-conquer, feedback
- Heuristics

Kozsik, T. et al.: Parallelization by Refactoring

Task farm

```
[ parse(scan(read( Module )))
                        || Module <- Modules ]
```

```
Work = fun(Module) ->
           parse(scan(read(Module))) end,
skel:do([{farm,[{seq,Work}],5}], Modules)
```

Kozsik, T. et al.: Parallelization by Refactoring



Pipeline

```
[ parse(scan(read( Module )))
                        || Module <- Modules ]
```

```
Stages =
   [{seq,read/1},{seq,scan/1},{seq,parse/1}],
skel:do([{pipe,Stages}], Modules)
```

Kozsik, T. et al.: Parallelization by Refactoring

Farm of pipes

```
[ parse(scan(read( Module )))
                        || Module <- Modules ]
```

```
Stages =
   [{seq,read/1},{seq,scan/1},{seq,parse/1}],
Work = {pipe,Stages},
skel:do([{farm,Work,5}], Modules)
```

Kozsik, T. et al.: Parallelization by Refactoring

Functional “quicksort”: d&c

```
qs ( List ) ->
  case List of
    []    -> [] ;
    [H|T] ->
       {List1,List2} = lists:partition(
                          fun(X) -> X < H end,
                          T),
       qs(List1) ++ [H] ++ qs(List2)
  end.
```

Kozsik, T. et al.: Parallelization by Refactoring



Karatsuba: d&c

```
karatsuba( Num1 , Num2 ) ->
    ...
    Z0 = karatsuba( Low1 , Low2 ),
    Z1 = karatsuba( add(Low1,High1),
                    add(Low2,High2) ),
    Z2 = karatsuba( High1 , High2 ),
    ...
```

Kozsik, T. et al.: Parallelization by Refactoring

Radix sort: d&c

```
sort( []  ,   _    ) -> [] ;
sort( [V] ,   _    ) -> [V] ;
sort( List, Level ) ->
  Buckets = divide( List, Level ),
  SortedLists =
    lists:map( fun(B) -> sort(B,Level+1) end,
               Buckets ),
  lists:append( SortedLists ).
```

Kozsik, T. et al.: Parallelization by Refactoring



Minimax: d&c

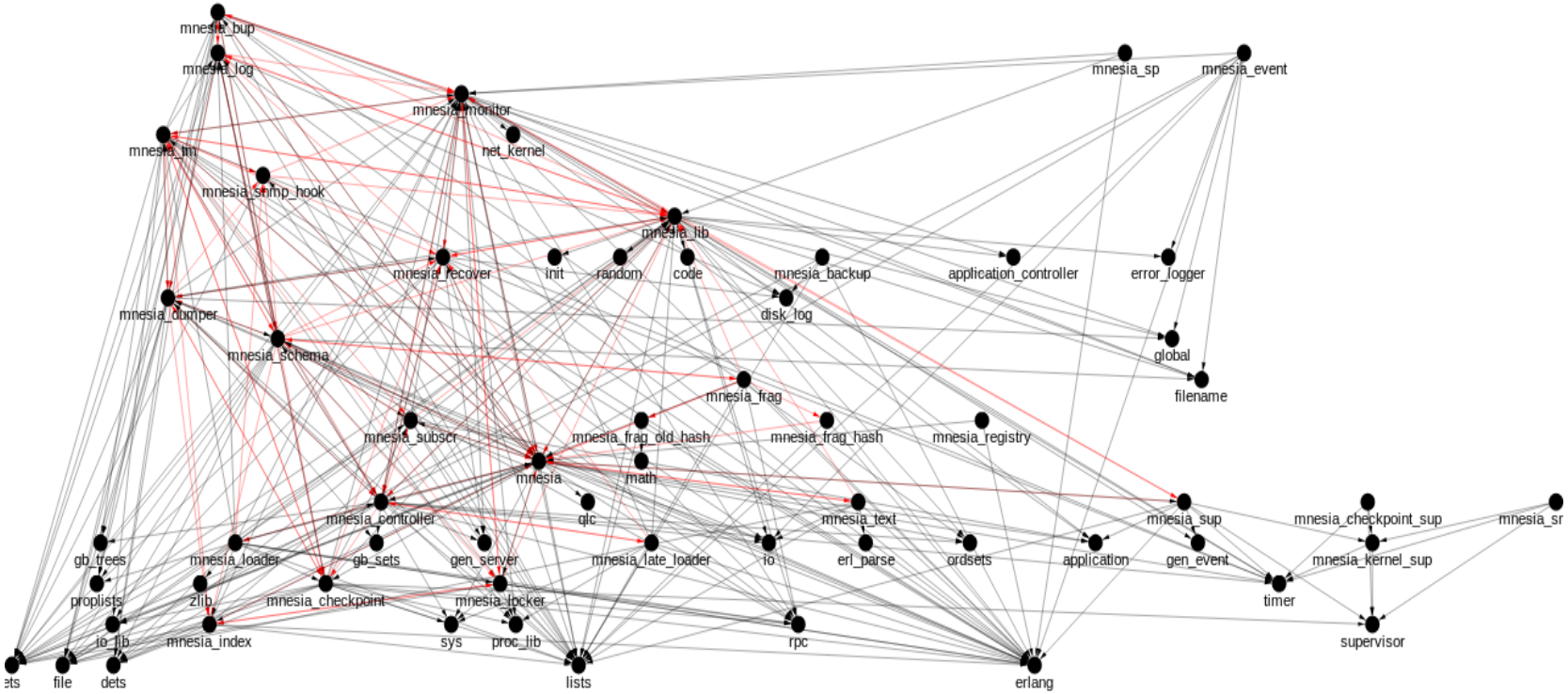
```
mm_max( Node, Depth ) ->
  case Depth == 0 orelse terminal(Node)
    true  -> value ( Node ) ;
    false -> lists:max([ mm_min(C, Depth-1)
                       || C <- children(Node)
                       ])
  end.

mm_min( Node, Depth ) -> ... mm_max ...
```

Kozsik, T. et al.: Parallelization by Refactoring

RefactorErl

Static source code analyzer and transformer

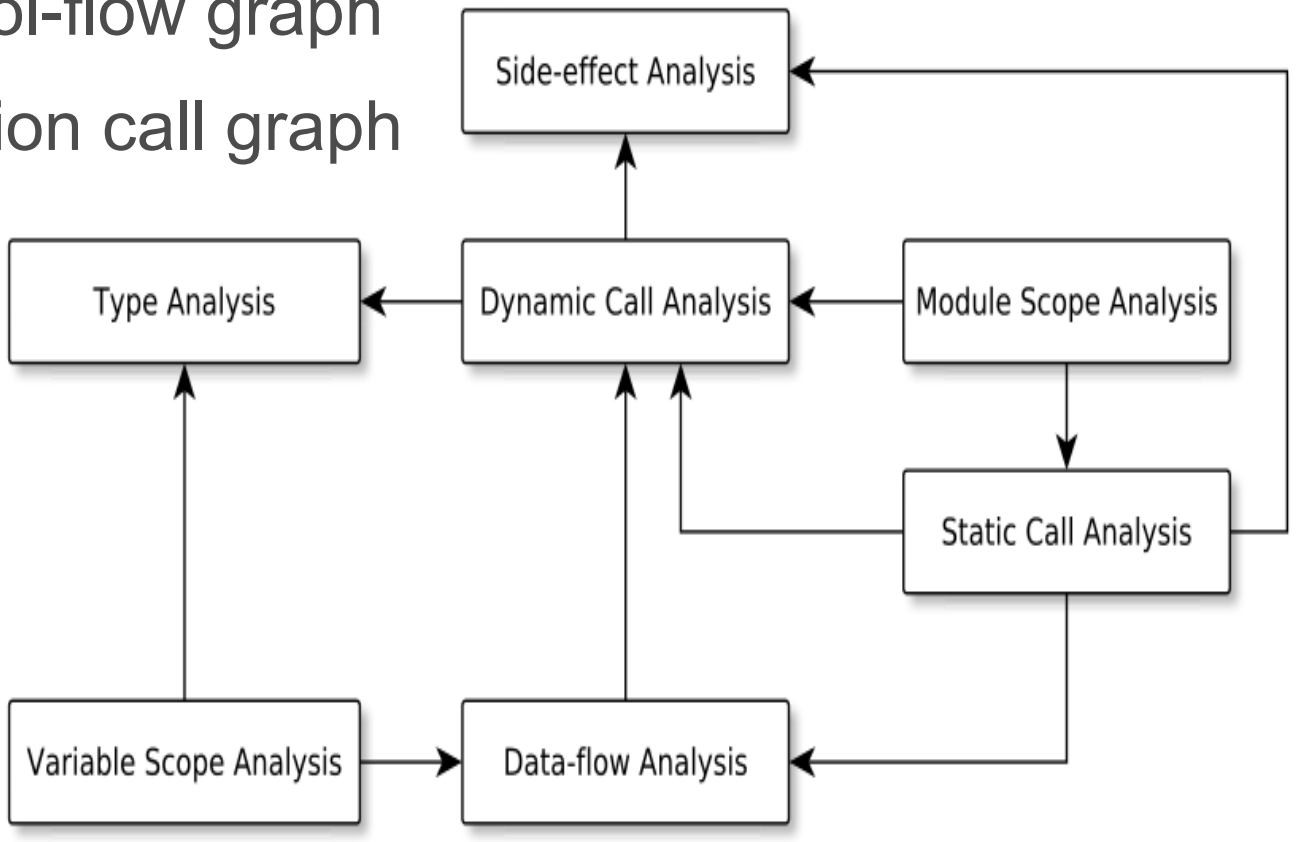
http://plc.inf.elte.hu/erlang



Kozsik, T. et al.: Parallelization by Refactoring

Standard static analyses

- Data-flow graph
- Control-flow graph
- Function call graph



Kozsik, T. et al.: Parallelization by Refactoring

Pattern-specific analyses

- Identify components
  - Side effects
- Identify patterns
  - Data dependency

Kozsik, T. et al.: Parallelization by Refactoring

Component

Action performed by Worker (farm) or Stage (pipe)

- Side-effect analysis
  - Message passing
  - NIFs and global variables
  - ETS etc.
  - Process dictionary, node names
  - Exceptions
- Hygiene rather than purity

Kozsik, T. et al.: Parallelization by Refactoring

Hygienic component

- Identify used resources
- Classify read/alter operations

  $use(C, R) \in \{ No, Read, Alter \}$

- Component set:
  components executed in parallel

$$\forall R \quad \forall C_1 \neq C_2 \in S:$$
$$use(C_1, R) = Alter \rightarrow use(C_2, R) = No$$

Kozsik, T. et al.: Parallelization by Refactoring

Element-wise processing

```
[ parse(scan(read( Module )))
                        || Module <- Modules ]
```

Kozsik, T. et al.: Parallelization by Refactoring

Element-wise processing

```
[ parse(scan(read( Module )))
                        || Module <- Modules ]
```

```
Work = fun(Module) ->
           parse(scan(read(Module))) end,
lists:map(Work,Modules)
```

```
psr([]) -> [];
psr([H|T]) -> [parse(scan(read(H)))|psr(T)].
          ... psr(Modules) ...
```

Kozsik, T. et al.: Parallelization by Refactoring



Element-wise processing

```
f(P1, P2, P3, P4) ->
   case P3 of
      [] -> [];
      [ Head | Tail ] ->
         X = ... Head ... ,
         [ X | f(P1,P2,Tail,P4) ]
   end.
```

a map-like function

Kozsik, T. et al.: Parallelization by Refactoring

Element-wise processing

```
f(P1, P2, P3, P4) ->
   case P3 of
      [] -> [];
      [ Head | Tail ] ->
         X = ... Head ... ,
         [ X | f(P1,P2,Tail,P4) ]
   end.
```

a map-like function

- *f* must be recursive:

  the interprocedural CFG must contain an execution path from the "starting node" of *f* to a "call-node" of *f*

  $\exists p \in EP(start_f)$ such that $call_f \in p$

- ... base case ... single recursive call ... regularities ... ... data dependencies ... compact data flow reaching ...

Kozsik, T. et al.: Parallelization by Refactoring



Divide-and-conquer

- A function has multiple recursive calls to itself
- The arguments of a recursive call do not depend on the result of another recursive call

Costly!

- *f must be recursive:*

  the interprocedural CFG must contain an execution path from the "starting node" of *f* to a "call-node" of *f*

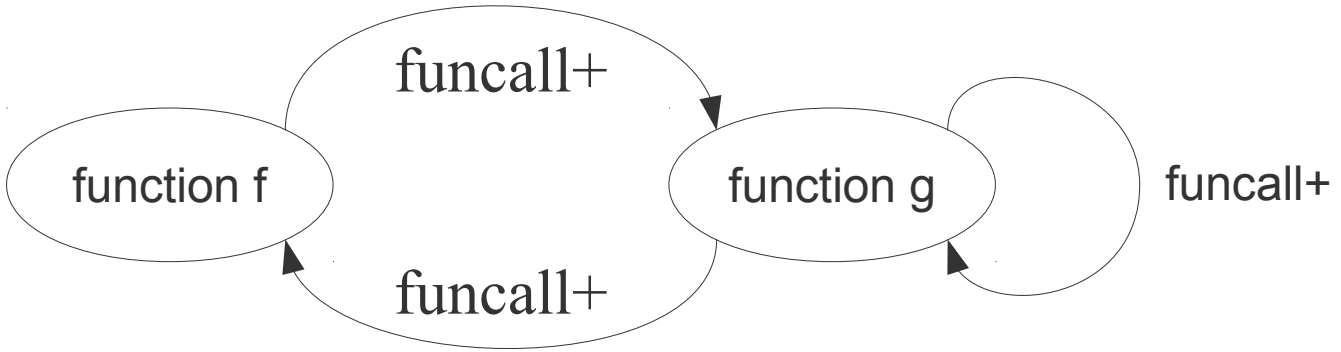
  $\exists p \in EP(start_f)$ such that $call_f \in p$

- etc.

Kozsik, T. et al.: Parallelization by Refactoring

Faster rules

- If *h* operates on lists, and
  - contains list comprehension with *h* in head
  - passes *h* to *lists:map/2* or a map-like function
- Analyze the function call graph



Kozsik, T. et al.: Parallelization by Refactoring

Experiments

- Ant Colony Optimization
  - Single Machine Total Weighted Tardiness Problem
- Image merging case study
- Intensional Computing Engine
  - Evaluator of the abstract syntax tree of ICE
- Evolutionary Multi-Agent Systems framework
- Thorn
  - Map-reduce framework
- Mnesia
  - Distributed database management system
- RefactorErl core
  - Static program analysis&transformation framework

Kozsik, T. et al.: Parallelization by Refactoring

Problem sizes

	ELOC	Functions	Files
Ant Colony Optimization	483	56	21
Image merging	779	104	6
ICE evaluator	1094	141	21
Thorn	1313	158	15
EMAS framework	1646	177	25
RefactorErl core	19694	1534	53
Mnesia	22653	1693	31

Kozsik, T. et al.: Parallelization by Refactoring

Discovery results

	farm	pipe	reduce	d&c	pool
Ant Colony Optimization	10				
Image merging	34	4	2		
ICE evaluator	7		4		
Thorn	17		5		
EMAS framework	71	20	16		9
RefactorErl core	486	49	55	31	
Mnesia	135	8	36	57	2

Map-like functions: 9

Kozsik, T. et al.: Parallelization by Refactoring

Some interesting candidates

- map-like functions
- divide-and-conquer algorithms

Kozsik, T. et al.: Parallelization by Refactoring



Map-like function (Mnesia)

```
reverse([]) -> [];
reverse([ H=#commit{ ram_copies      = Ram,
                     disc_copies     = DC,
                     disc_only_copies= DOC,
                     snmp            = Snmp }
        | R ]) ->
  [ H#commit{
     ram_copies      = lists:reverse(Ram),
     disc_copies     = lists:reverse(DC),
     disc_only_copies = lists:reverse(DOC),
     snmp            = lists:reverse(Snmp)
    }
  | reverse(R) ].
```

Kozsik, T. et al.: Parallelization by Refactoring



Map-like function (EMAS)

```
count_funstats(_,[]) -> [];

count_funstats(
      Agents,
      [{Stat, MapFun, ReduceFun, OldAcc} | T]
    ) ->

    NewAcc =
       lists:foldl( ReduceFun, OldAcc,
          [MapFun(Agent) || Agent <- Agents] ),

    [ {Stat, MapFun, ReduceFun, NewAcc}
      | count_funstats(Agents,T)
    ].
```

Kozsik, T. et al.: Parallelization by Refactoring

D&C (RefactorErl)

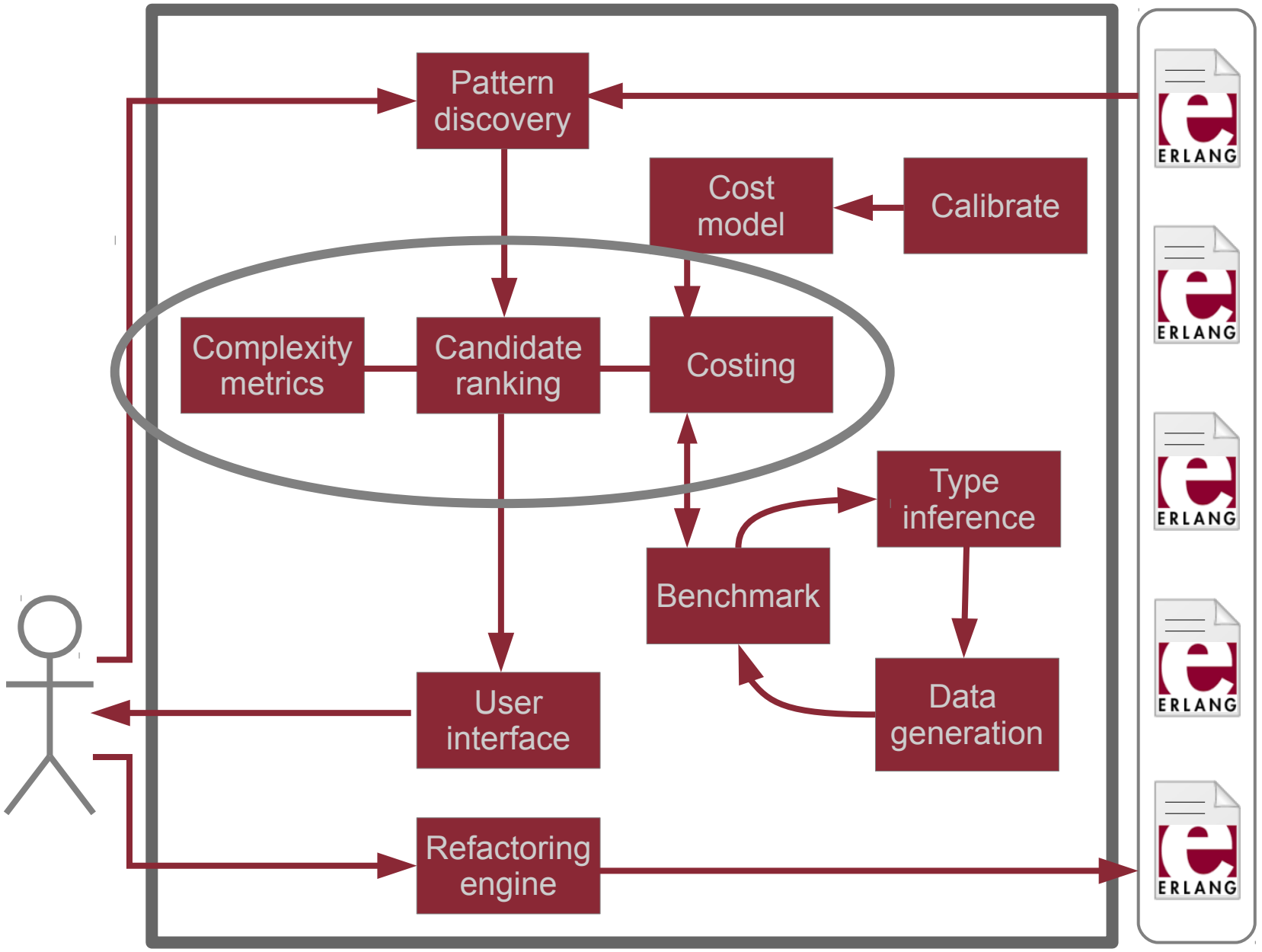
```
...
listcons_length(N, #expr{}) ->
  Ns = ?Dataflow:?reach([N], [back], true),
  L1 = [N2 || N2 <- Ns, N2 /= N,
              ?Graph:class(N2) == expr],
  {L2, L3} = lists:partition(
                    fun is_cons_expr/1, L1 ),
  if L2 == [] orelse L3 /= [] ->
       incalculable;
     true ->
       lists:append(lists:map(
                  fun listcons_length/1, L2))
  end;
...
```

Kozsik, T. et al.: Parallelization by Refactoring

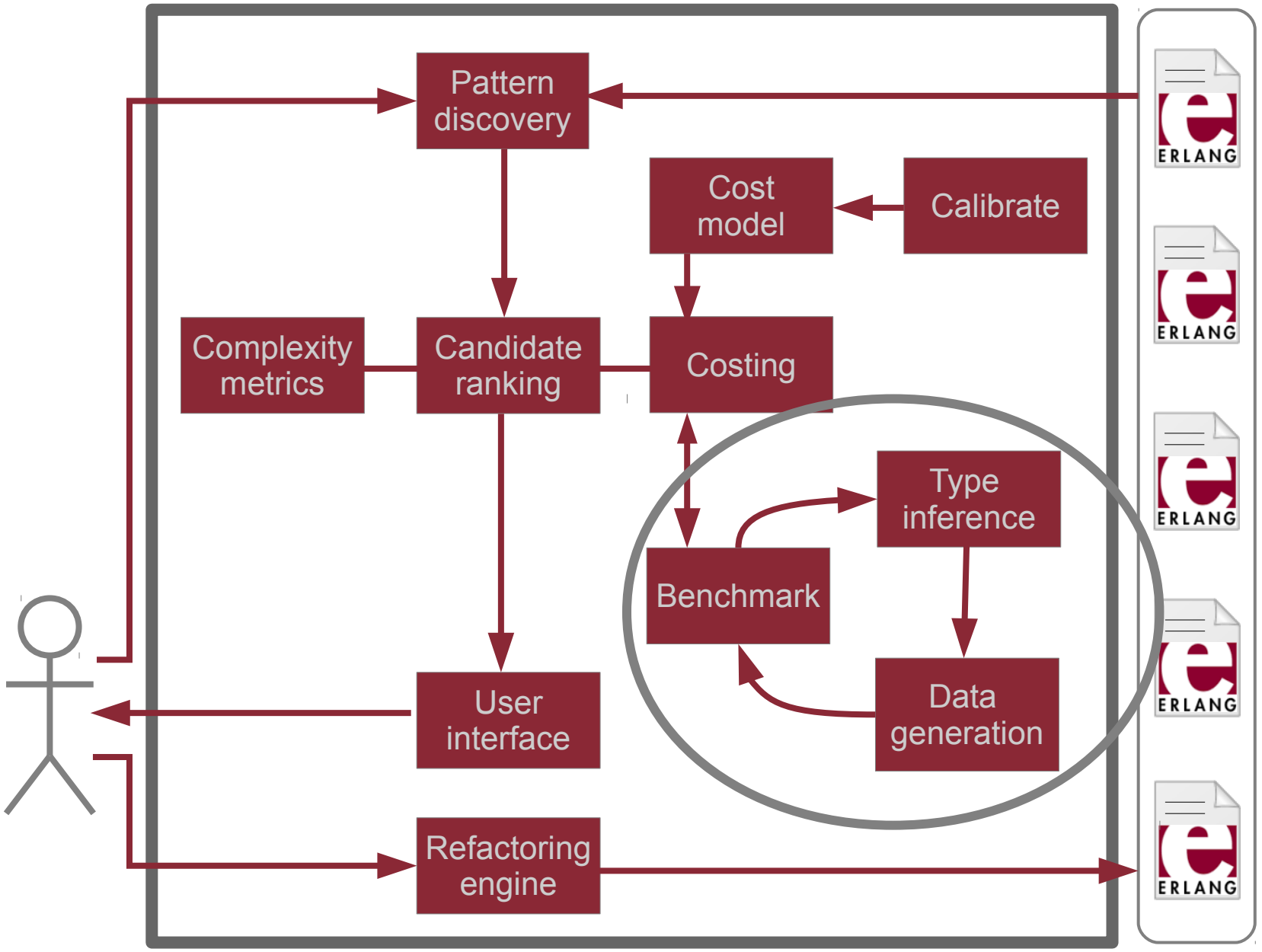
Another D&C (RefactorErl)

```
realtoken_neighbour(Node, DirFun, DownFun) ->
  case lists:member(?Graph:class(Node),[clause,expr,form,typexp,lex]) of
    false -> no;
    _     -> case ?Syn:parent(Node) of
               []           -> no;
               [{_,Parent}] -> case lists:dropwhile( fun({_T,N}) -> N/=Node end,
                                                     DirFun(?Syn:children(Parent))
                                                   ) of
                                  [{_,Node},{_,NextNode}|_] -> DownFun(NextNode);
                                  _                         ->
                                     realtoken_neighbour(Parent, DirFun, DownFun)
                               end;
               Parents      -> realtoken_neighbour_( Parents, DownFun(Node),
                                                     DirFun, DownFun )
             end
  end.

% Implementation helper function for realtoken_neighbour/3

realtoken_neighbour_([], _FirstLeaf,_DirFun,_DownFun) -> no;
realtoken_neighbour_([{_,Parent}|Parents], FirstLeaf, DirFun, DownFun) ->
  case realtoken_neighbour(Parent, DirFun, DownFun) of
    FirstLeaf -> realtoken_neighbour_(Parents, FirstLeaf, DirFun, DownFun);
    NextLeaf  -> NextLeaf
  end.
```

Kozsik, T. et al.: Parallelization by Refactoring



Kozsik, T. et al.: Parallelization by Refactoring



Kozsik, T. et al.: Parallelization by Refactoring

Benchmarking

- Split up pattern candidates into components
- Determine free variables (inputs)
- Assemble a new module
  - Components turned into functions
  - Instrumented with time measurements
- Load module
- Generate random input and profile
- Make statistics

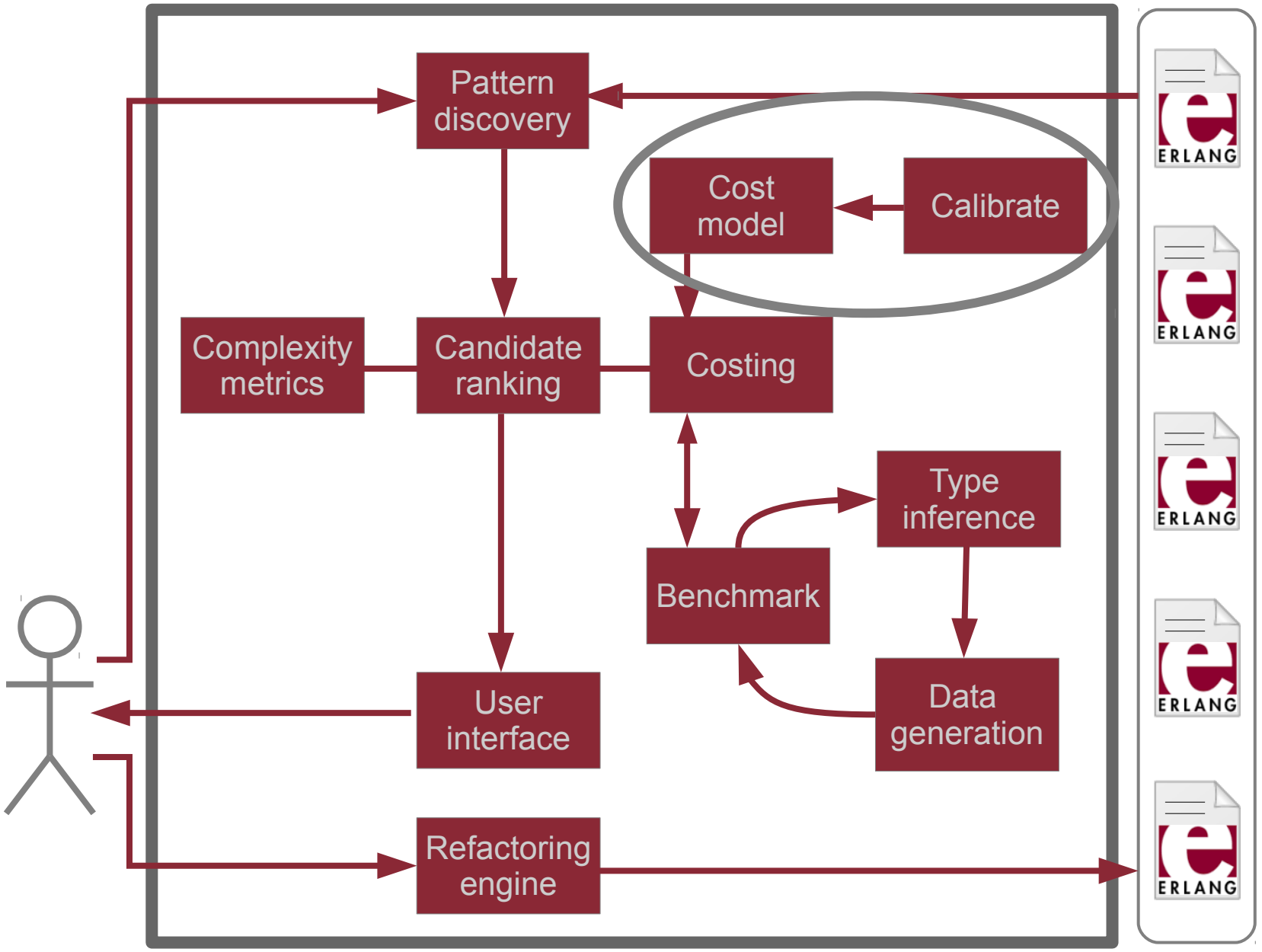
Kozsik, T. et al.: Parallelization by Refactoring

Random input?

- Not always meaningful...
- ... but easy to automate!

- Find out the type of free variables (TypEr)
- QuickCheck generates values by type

Kozsik, T. et al.: Parallelization by Refactoring



Kozsik, T. et al.: Parallelization by Refactoring

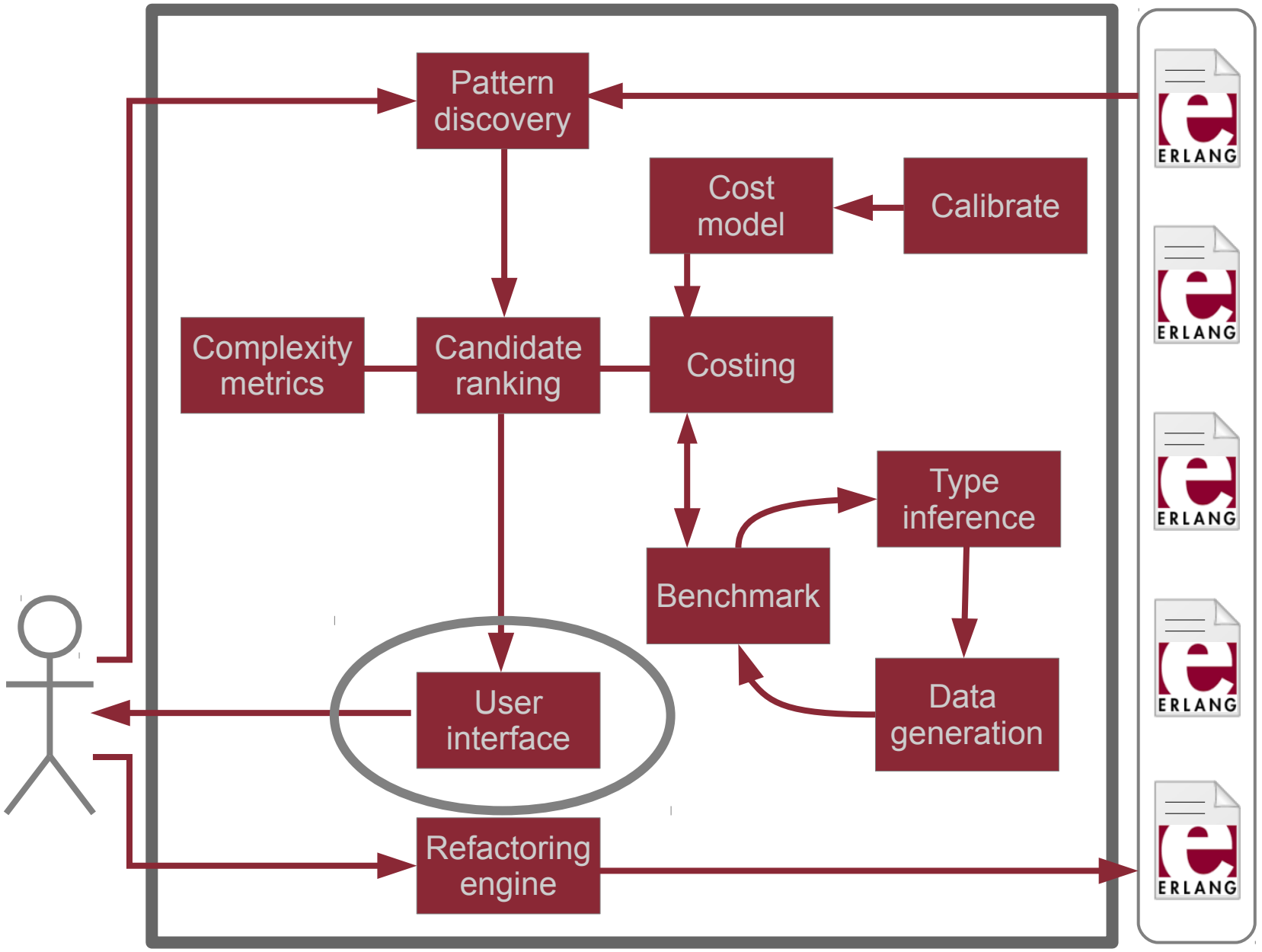
Cost model

- Approximation
- E.g. for farm:

$$T_{farm} := T_{work} * \lceil L/min(N_p, N_w) \rceil + T_{spawn} * (N_w + 2) + T_{copy}(L) * 3 + T_{spawn} + T_{copy}(L) * 2$$

- Needs calibration!

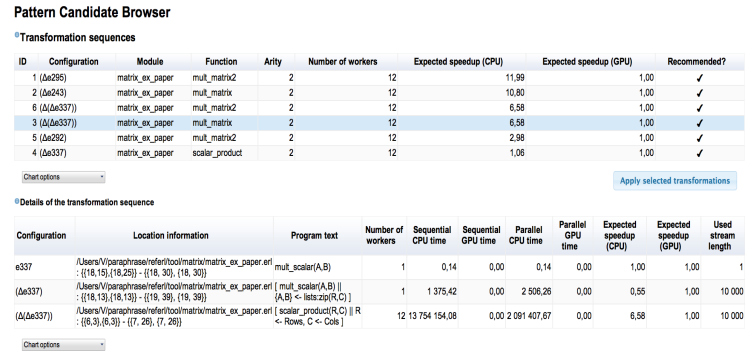
Kozsik, T. et al.: Parallelization by Refactoring



Kozsik, T. et al.: Parallelization by Refactoring

Pattern Candidate Browser

- After ranking pattern candidates
- Web-based interface
  - Information for decision making
  - Not too many details
- Services
  - Multiple users
  - Persistent results
  - Export XML, JSON, CSV, Erlang terms



Pattern Candidate Browser

Transformation sequences

ID	Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)	Expected speedup (GPU)	Recommended?
1	(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99	1,00	✓
2	(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80	1,00	✓
6	(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58	1,00	✓
3	(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58	1,00	✓
5	(Δe292)	matrix_ex_paper	mult_matrix2	2	12	2,98	1,00	✓
4	(Δe337)	matrix_ex_paper	scalar_product	2	12	1,06	1,00	✓

Chart options

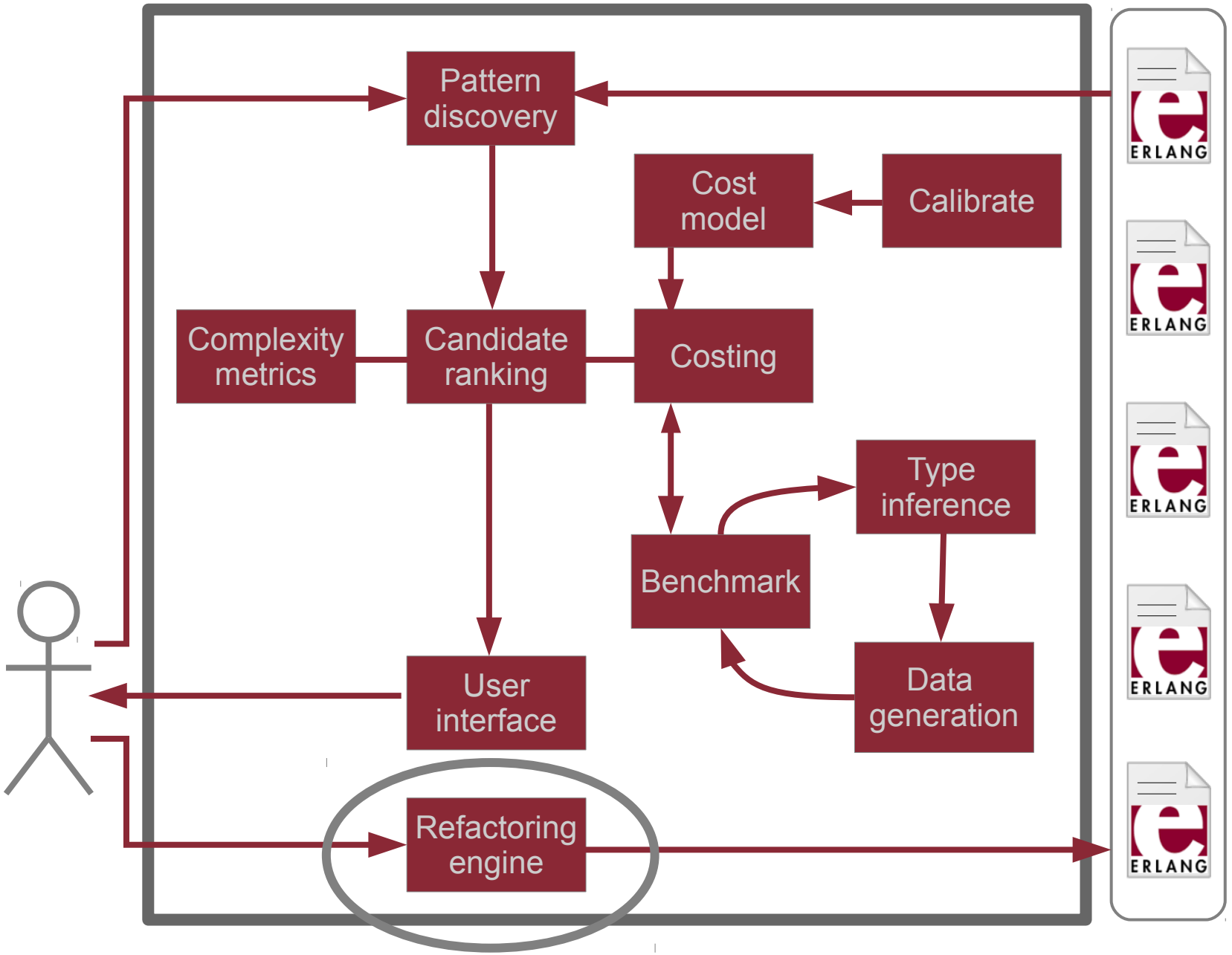
Apply selected transformations

Details of the transformation sequence

Configuration	Location information	Program text	Number of workers	Sequential CPU time	Sequential GPU time	Parallel CPU time	Parallel GPU time	Expected speedup (CPU)	Expected speedup (GPU)	Used stream length
e337	/Users/V/paraphrase/referi/tool/matrix/matrix_ex_paper.erl : {{18,15},{18,25}} - {{18, 30}, {18, 30}}	mult_scalar(A,B)	1	0,14	0,00	0,14	0,00	1,00	1,00	1
(Δe337)	/Users/V/paraphrase/referi/tool/matrix/matrix_ex_paper.erl : {{18,13},{18,13}} - {{19, 39}, {19, 39}}	[mult_scalar(A,B) || {A,B} <- lists:zip(R,C)]	1	1 375,42	0,00	2 506,26	0,00	0,55	1,00	10 000
(Δ(Δe337))	/Users/V/paraphrase/referi/tool/matrix/matrix_ex_paper.erl : {{6,3},{6,3}} - {{7, 26}, {7, 26}}	[scalar_product(R,C) || R <- Rows, C <- Cols]	12	13 754 154,08	0,00	2 091 407,67	0,00	6,58	1,00	10 000

Chart options

Kozsik, T. et al.: Parallelization by Refactoring



Kozsik, T. et al.: Parallelization by Refactoring

Refactoring

- Program Shaping
- Introduction of Skeletons
- Cleanup Transformations

Either directly or after PC discovery

Kozsik, T. et al.: Parallelization by Refactoring

Program shaping

- Restructures and tunes program to get it into appropriate shape for skeleton introduction
- Removal of:
  - Dependencies
  - Locks
  - Global State
  - Nestings
  - Copying

Kozsik, T. et al.: Parallelization by Refactoring

Future work: more refactorings

```
sort( List ) -> sort(List,0).
sort( List,   _   ) when length(List) < 2 -> List;
sort( List, Level ) ->
  lists:append(
    [sort( Bucket, Level+1 ) || Bucket <- divide( List, Level )]
  ).
```

```
sort( List ) -> skel:do( [{ dc,
     fun({Lst,Level}) -> length(Lst) < 2 end,
     fun({Lst,Level}) -> Lst end,
     fun({Lst,Level}) ->
          [ {Bucket,Level+1} || Bucket <- divide(Lst, Level) ]
        end,
     fun lists:append/1
  }], {List,0}).
```

Kozsik, T. et al.: Parallelization by Refactoring

Resources

ParaPhrase project (FP7 contract no. 288570)
http://paraphrase-ict.eu/

ParaPhrase @ ELTE
http://paraphrase-enlarged.elte.hu/

PaRTE docs & download:
http://pnyf.inf.elte.hu/trac/refactorerl/wiki/parte

CEFP 2015 instructions:
http://pnyf.inf.elte.hu/trac/refactorerl/wiki/parte/cefp

Kozsik, T. et al.: Parallelization by Refactoring

Conclusions

- Pattern-based parallelism
- **Pa**raPhrase **R**efactoring **T**ool for **E**rlang

  Pattern discovery and refactoring

  Candidates prioritized,
  support for decision making
  - Finds many places to introduce parallelism
  - Supports candidate ranking
  - Works effectively with a smart programmer
  - Offers performance gains with small effort

Kozsik, T. et al.: Parallelization by Refactoring