Discovering Parallel Pattern Candidates

Tamás Kozsik
Eötvös Loránd Univ.


Melinda Tóth
ELTE-Soft Ltd.

Budapest, Hungary

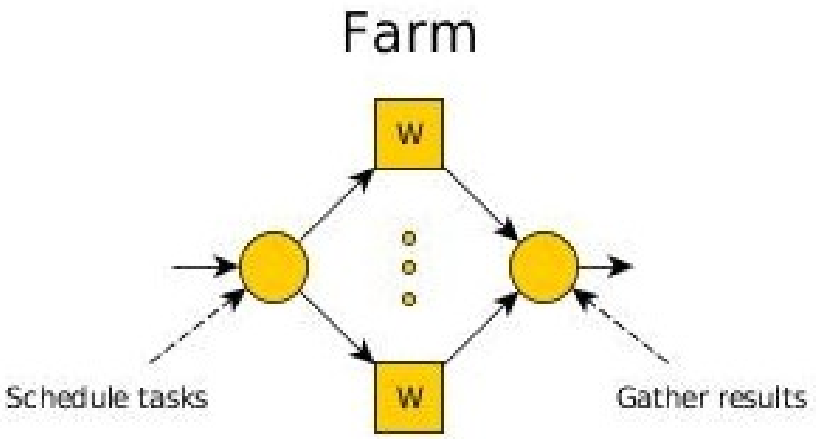
Motivation

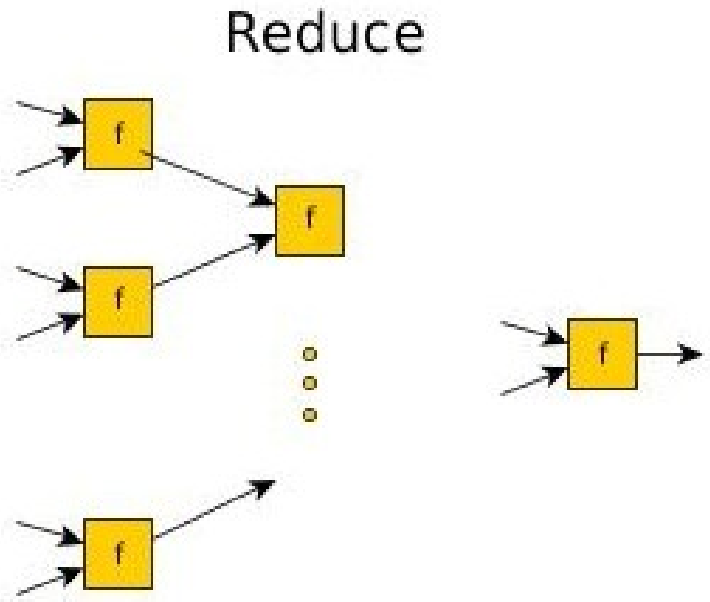
- Highly heterogeneous mega-core computers
- Performance and energy
- Think parallel
  - High-level programming constructs
  - Deadlocks etc. eliminated by design
  - Communication packaged/abstracted
  - Performance information is part of design
- Restructure legacy code

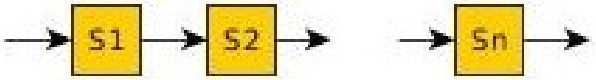
Pattern-based parallelism

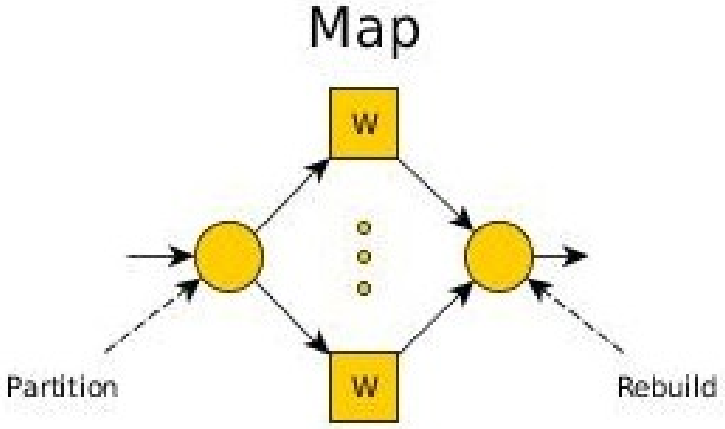
High-level approach to parallel programming

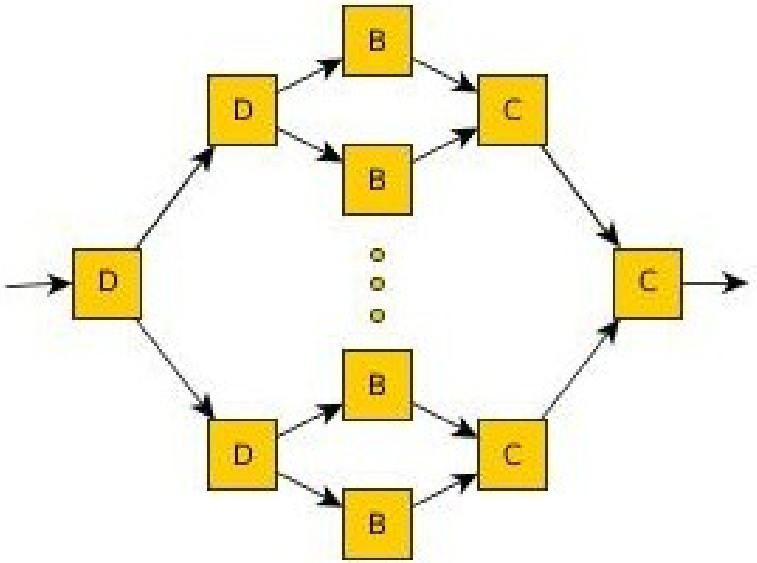
- Rely on a library of algorithmic skeletons
- Easier to develop code
- Easier to modify / maintain
- Better utilization of resources
  - Static resource management
  - Dynamic resource management


Farm
W
W
Schedule tasks
Gather results

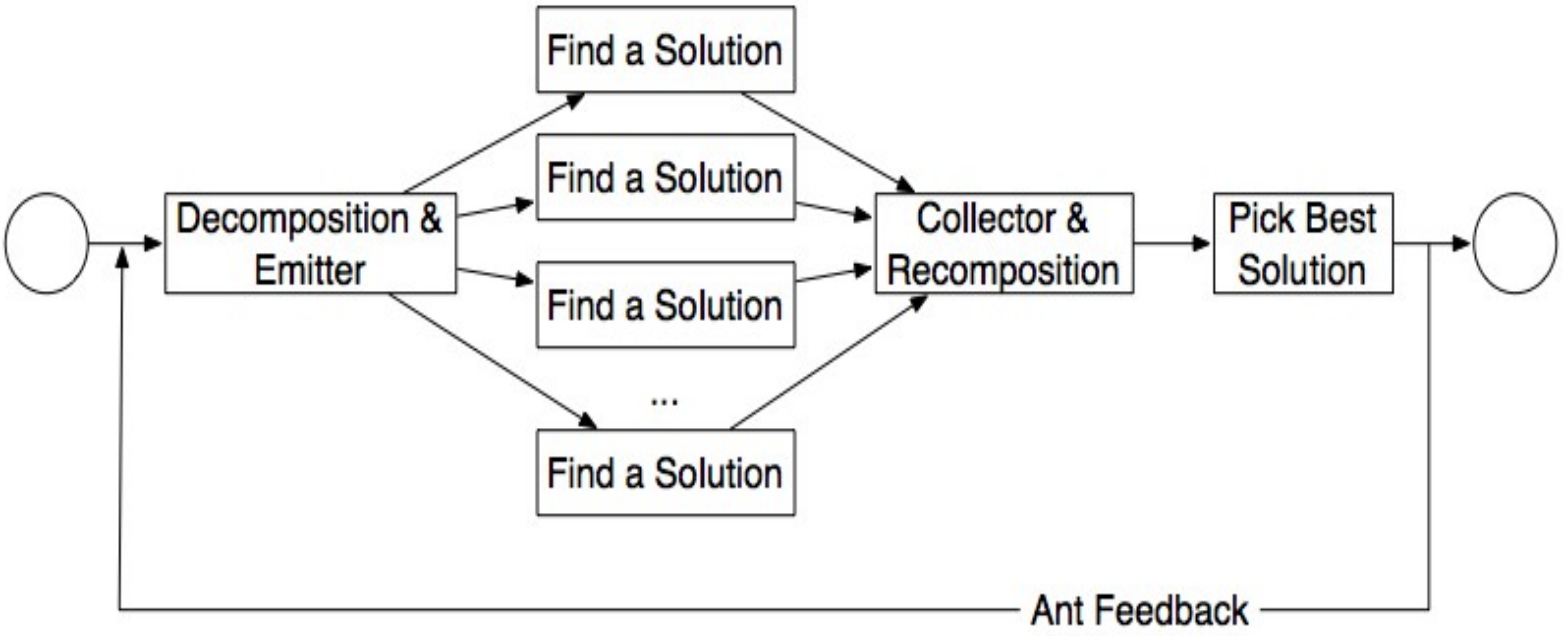

Reduce
f
f
f
f
f


Pipeline
S1
S2
Sn


Map
W
W
Partition
Rebuild


Divide&Conquer
D
D
D
B
B
B
B
C
C
C

Pool pattern



Ant Colony Optimization

Kozsik, Tóth: Discovering Parallel Pattern Candidates

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, Tóth: Discovering Parallel Pattern Candidates


PARAPHRASE

Parallel skeletons

skel library

- Available at: http://skel.weebly.com/
- Basic skeletons (farm, pipe, ord, feedback)

High-level patterns: skel hlp

- Available at: http://paraphrase-ict.eu/Deliverables/deliverable-2.6
- dc, evolutionPool

Heterogeneous skeletons: Lapedo

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

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 )
```

Example: radix sort

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

Kozsik, Tóth: Discovering Parallel Pattern Candidates

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 }
```

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, Tóth: Discovering Parallel Pattern Candidates

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

PaRTE

ParaPhrase Refactoring Tool for Erlang

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

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, Tóth: Discovering Parallel Pattern Candidates

ern Candidate Browser

nsformation sequences

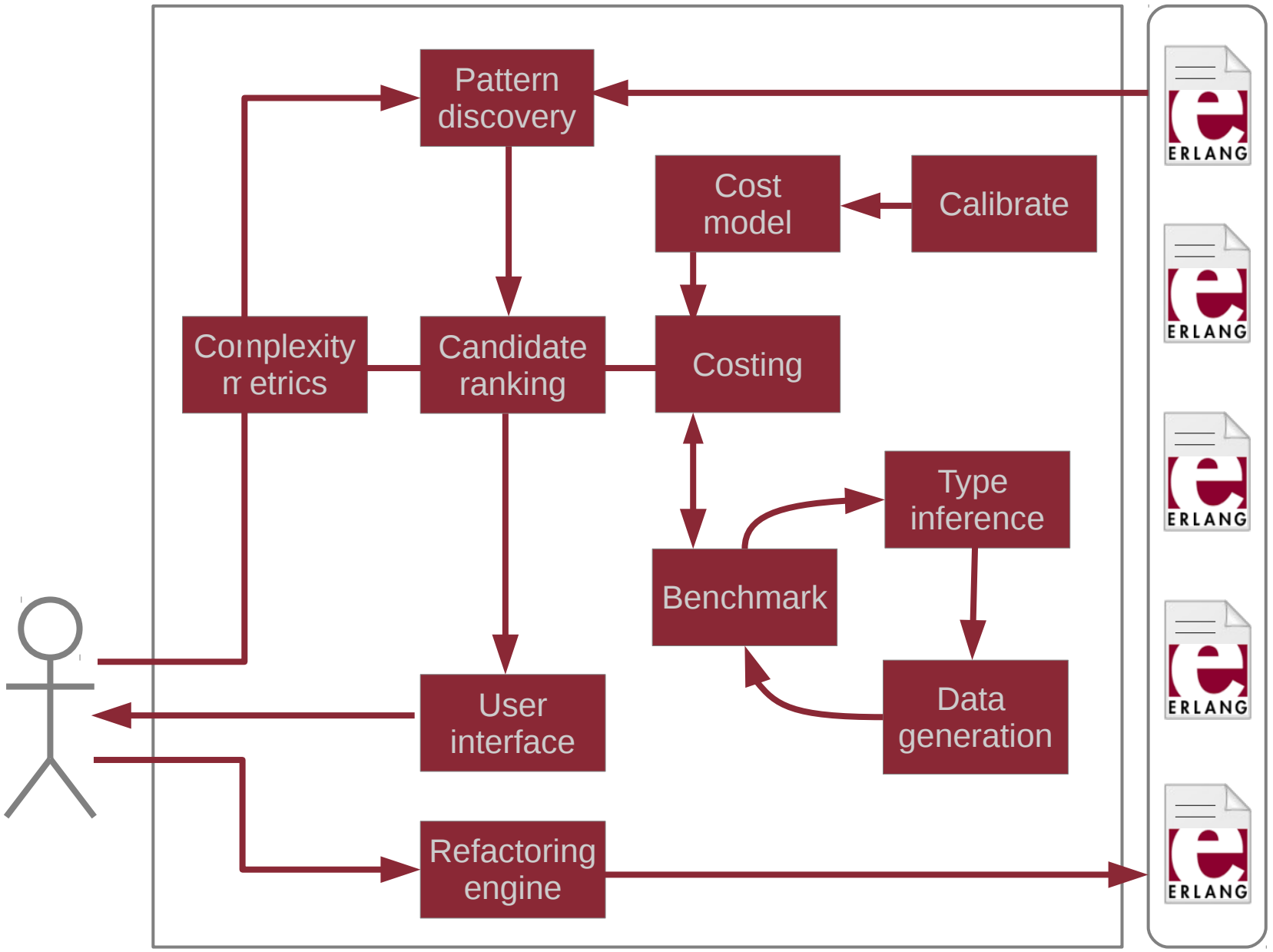
Configuration	Module	Function	Arity	Number of workers	Expected speedup (CPU)
(Δe295)	matrix_ex_paper	mult_matrix2	2	12	11,99
(Δe243)	matrix_ex_paper	mult_matrix	2	12	10,80
(Δ(Δe337))	matrix_ex_paper	mult_matrix2	2	12	6,58
(Δ(Δe337))	matrix_ex_paper	mult_matrix	2	12	6,58
(Δe292)	matrix_ex_paper	mult_matrix2	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, Tóth: Discovering Parallel Pattern Candidates

Pattern candidate discovery

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

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.
```

Karatsuba: d&c

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

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 ).
```



Minimax: d&c

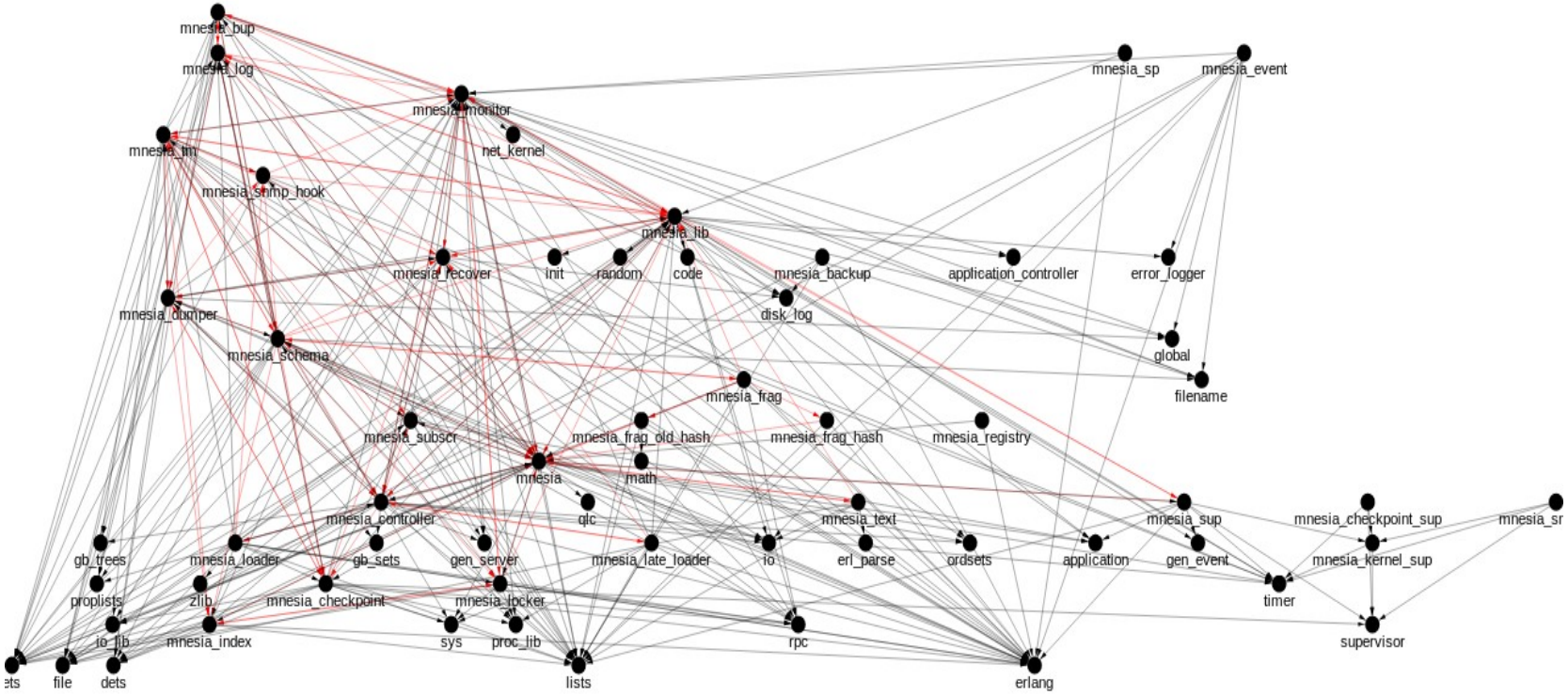
```
mm_max( Node, Depth ) ->
  case Depth == 0 orelse terminal(Node)
    true  -> value ( Node ) ;
    false -> lists:max([ mm_min(C, Depth-1)
                       || C <- children(Node)
                       ])
  end.

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

RefactorErl

Static source code analyzer and transformer

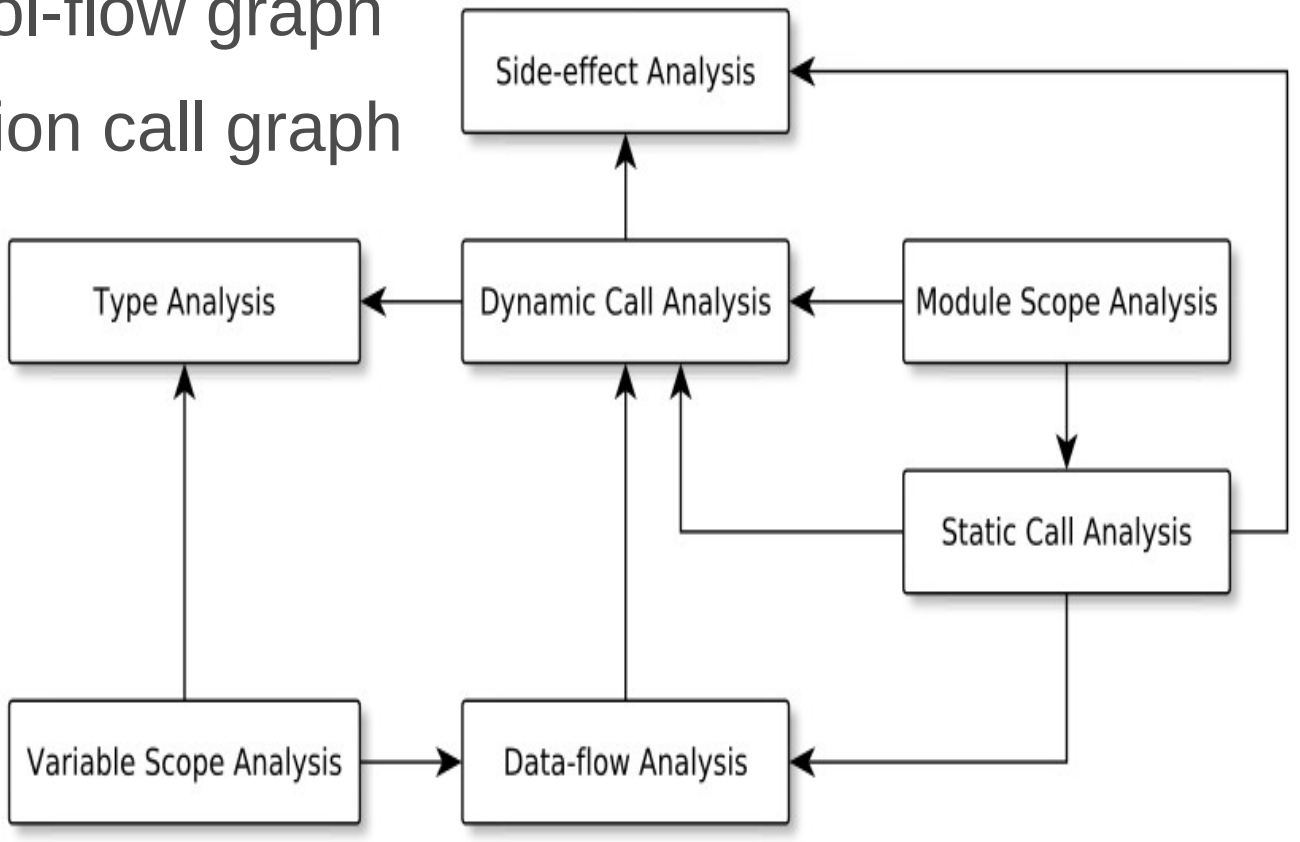
http://plc.inf.elte.hu/erlang



Kozsik, Tóth: Discovering Parallel Pattern Candidates

Standard static analyses

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



Kozsik, Tóth: Discovering Parallel Pattern Candidates

Element-wise processing

```
f(Parameter, List) ->
   case List of
      [] -> [];
      [ Head | Tail ] ->
         X = ... Head ... ,
         [ X | f(Parameter,Tail) ]
   end.
```

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 dependences ... compact data flow reaching ...

Kozsik, Tóth: Discovering Parallel Pattern Candidates

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:*

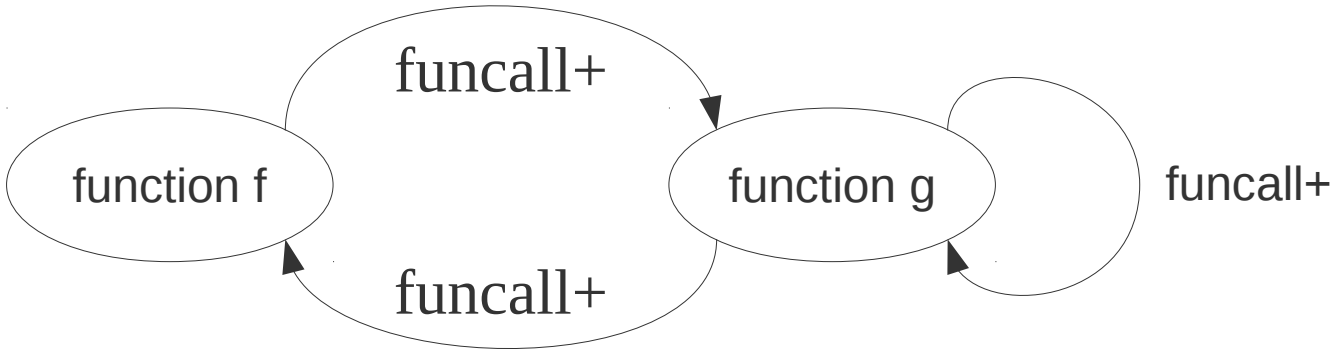
  the interprocedural CFG must contain an execution path from the "starting node" of *f* to a "call-node" of *f*

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

- etc.

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



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

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

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

Some interesting candidates

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

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, Tóth: Discovering Parallel Pattern Candidates

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)
   ].
```

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, Tóth: Discovering Parallel Pattern Candidates

Another D&C (RefactorErl)

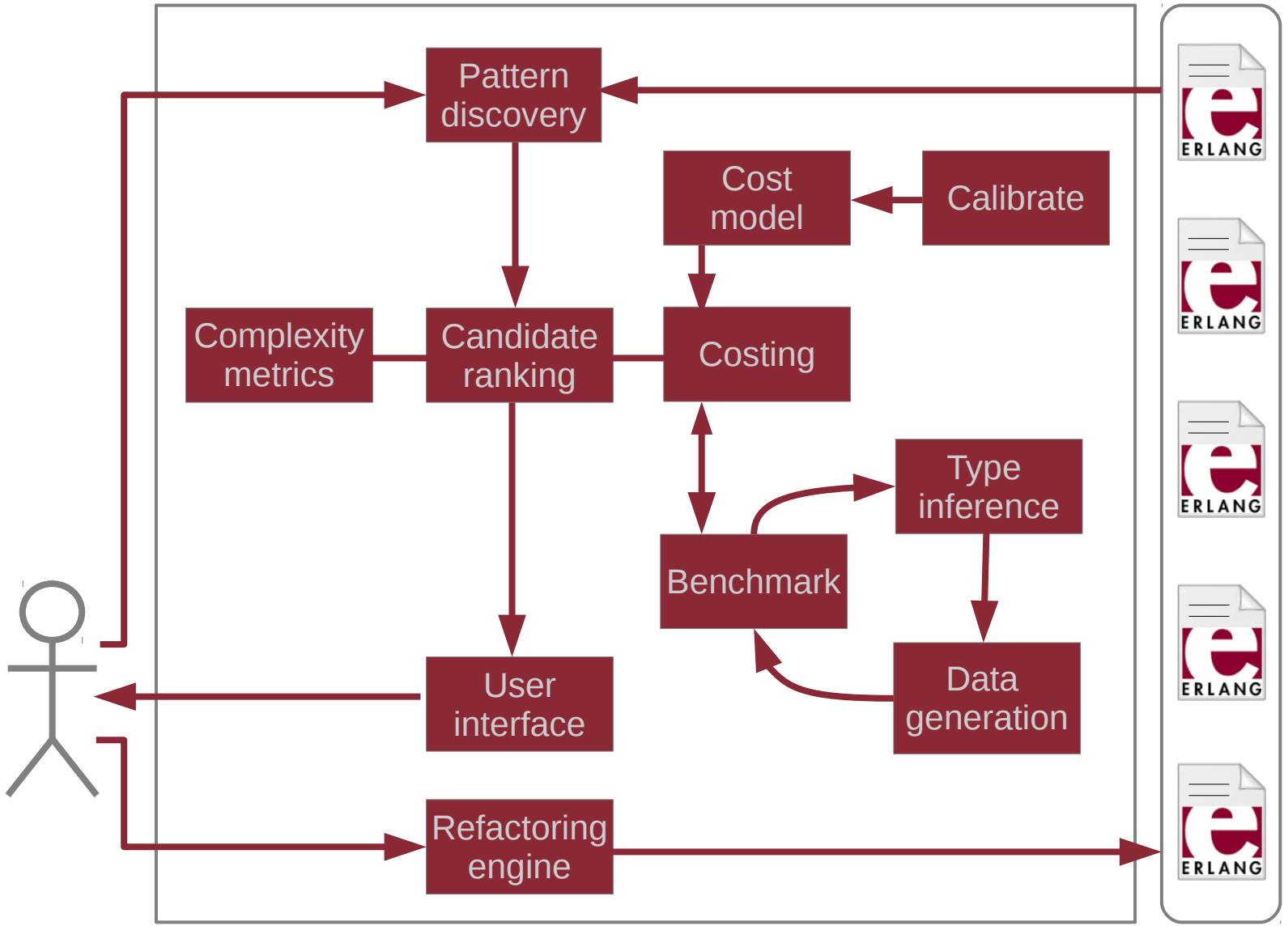
```
realtoken_neighbour(Node, DirFun, DownFun) ->
  case lists:member(?Graph:class(Node),[clause,expr,form,typexp,lex]) of
    false -> no;
    _     -> case ?Syn:parent(Node) of
               []           -> no;
               [{_,Parent}] -> case lists:dropwhile( fun({_T,N}) -> N/=Node end,
                                                     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, Tóth: Discovering Parallel Pattern Candidates

Other components of PaRTE



Kozsik, Tóth: Discovering Parallel Pattern Candidates

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, Tóth: Discovering Parallel Pattern Candidates

Conclusions

- Pattern-based parallelism
- **Pa**raPhrase **R**efactoring **T**ool for **E**rlang
- Pattern discovery and refactoring
- Candidates prioritized,
  support for decision making

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

Docs & Download:
http://pnyf.inf.elte.hu/trac/refactorerl/wiki/parte