Where shall I parallelize?

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

Eötvös Loránd University
and ELTE-Soft Kft.
Budapest, Hungary

Erlang User Conference
Stockholm, 9-10 June, 2014
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
Where shall I parallelize?

Tool to...

- find parallelizable code
- help making decisions
- reshape the code
- introduce parallelism
Parallel Patterns for Adaptive Heterogeneous Multicore Systems

ICT-288570    2011-2014    €4.2M budget
13 Partners, 8 European countries
http://www.paraphrase-ict.eu/
@paraphrase_fp7

Where shall I parallelize? (Tamás Kozsik)
Where shall I parallelize? (Tamás Kozsik)
• Programmability of heterogeneous parallel architectures
• Structured design and implementation of parallelism
• High-level parallel patterns
  map(reduce)          task farm       pipeline
  divide&conquer       orbit          stencil
• Dynamic (re)mapping on heterogeneous hardware
• C++/FastFlow and Erlang
Where shall I parallelize? (Tamás Kozsik)

Farm

Schedule tasks
Gather results

Pipeline

S1 → S2 → Sn

Map

Partition → Rebuild

Reduce

f → f → f

Divide & Conquer

D → B → C → D

B

C
• Identify (strongly hygienic) Components
• Patterns of parallelism
• Structure the components into a parallel program
  – Turn the patterns into concrete (skeleton) code
  – Take performance, energy etc. into account
• Restructure if necessary
• Use a refactoring tool!
Christopher Brown's talk

Bridging the Divide: A New Tool-Supported Methodology for Programming Heterogeneous Multicore Machines

- Skel library for Erlang (and C++)
- CPU/GPU systems
- Refactoring tool support (Wrangler)

\[
\text{OutputItems} = \text{skel:do(Skeleton, InputItems)}
\]

Where shall I parallelize? (Tamás Kozsik)
Pipeline

```
skel: do(
    [{pipe, [Skel1, Skel2, ..., SkelN]}],
    Inputs
)
```

Where shall I parallelize? (Tamás Kozsik)
Task farm

-sk el:do([\{farm, Skel, M\}], Inputs)
Nesting skeletons

{ feedback,
  [{pipe, [{farm, [{seq, fun find_solution/1}], NrW},
         {seq, fun pick_best/1}]
  }
},
  fun ant_feedback/1
}
Example

\[
\text{mul}(\text{Rows}, \text{Cols}) \to \\
\begin{array}{c}
[ [ \text{dotp}(\text{Row}, \text{Col}) \mid \text{Col} \leftarrow \text{Cols} ] \\
\mid \text{Row} \leftarrow \text{Rows} \\
].
\end{array}
\]

\[
\text{dotp}(\text{Row}, \text{Col}) \to \\
\text{lists:sum}(
\text{lists:zipwith}(
\text{fun erlang:}^* / 2, \text{Row}, \text{Col})).
\]
Our goal

Develop tool to...

- Find pattern candidates
- Rank and suggest
- Shape

=> introduce skeletons
Farm

mul(Rows,Cols) ->
[ [ dotp(Row,Col) || Col <- Cols ]
 || Row <- Rows
].

mul(Rows,Cols) ->
skel:do( [{ farm, [{seq, fun(Row) ->
 [ dotp(Row,Col) || Col <- Cols ] end
 }], 16}]], Rows ).

Where shall I parallelize? (Tamás Kozsik)
PaRTE

ParaPhrase Refactoring Tool for Erlang

Demo!
Expectations

PaRTE...

- changes the way you think about parallelism;
- completely redesigns your code;
- shows all places to introduce parallelism;
- predicts speedup dead exactly;
- is safe and automatic.
Expectations

PaRTEx...

- changes the way you think about parallelism;
- completely redesigns your code;
- shows all places to introduce parallelism;
- predicts speedup dead exactly;
- is safe and automatic.
Realistic expectations

PaRTE...

• can find many places to introduce parallelism;
• gives fair speedup predictions;
• works effectively with a smart programmer;
• offers performance gains with small effort.
Key idea

PaRTE can predict speedup by

- measuring sequential execution time on random input, and
- estimate parallel execution time.
Big picture

Where shall I parallelize? (Tamás Kozsik)
Pattern candidate discovery

- Syntactic (& semantic) information
  - List comprehensions
  - Recursive function definitions
- Side conditions
- Heuristics

\[
\text{mul}(\text{Rows}, \text{Cols}) \rightarrow \\
\left[ \left[ \text{dotp}(\text{Row}, \text{Col}) \mid \text{Col} \leftarrow \text{Cols} \right] \\
\mid \mid \text{Row} \leftarrow \text{Rows} \\
\right].
\]

Where shall I parallelize? (Tamás Kozsik)
Analyses

Where shall I parallelize? (Tamás Kozsik)
RefactorErl

Static source code analyzer and transformer

http://refactorerl.com

Where shall I parallelize? (Tamás Kozsik)
Features of RefactorErl

- Semantic Program Graph
- Gather information from the code
- Find dependencies
- Investigate bugs
- Share information among team members
- Refactor the code

Where shall I parallelize? (Tamás Kozsik)
Why shall I use RefactorErl?

- Shorten time-consuming daily jobs
- Improve teamwork
- Reduce human faults
- Facilitate the deployment of releases
- Minimize the training time of newbies

Effective software maintenance
Pattern candidate discovery

- List operations
  - List comprehensions
  - Library calls (lists:map/2)
  - Map-like recursive functions
- Task farms and pipelines
## Statistics

<table>
<thead>
<tr>
<th></th>
<th>RefactorErl</th>
<th>Wrangler</th>
<th>Mnesia</th>
<th>Dialyzer</th>
<th>ICE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELOC</td>
<td>106k</td>
<td>51k</td>
<td>23k</td>
<td>17k</td>
<td>13k</td>
</tr>
<tr>
<td>list compr.</td>
<td>2647</td>
<td>1014</td>
<td>101</td>
<td>247</td>
<td>59</td>
</tr>
<tr>
<td>lists:map/2</td>
<td>244</td>
<td>93</td>
<td>12</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>lists:filter/2</td>
<td>131</td>
<td>36</td>
<td>7</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>lists:foldl</td>
<td>189</td>
<td>32</td>
<td>28</td>
<td>24</td>
<td>22</td>
</tr>
<tr>
<td>lists:foldr</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>recursive</td>
<td>1028</td>
<td>605</td>
<td>362</td>
<td>278</td>
<td>436</td>
</tr>
</tbody>
</table>
Map-like function

- Recursive function with list parameter $P$
- Execution paths
  
  **Recursive:**
  - returns a list $R$
  - $\text{head}(R)$ does not depend on $P$, only on $\text{head}(P)$
  - $\text{tail}(R)$ is from the single recursive call on $\text{tail}(P)$
  - other parameters of the recursive call are the same

  **Non-recursive:**
  - $P = []$
  - returns $[ ]$
Components

- 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
Hygienic component

- Identify used resources
- Classify read/alter operations

\[
\text{use}(C, R) \in \{ \text{No, Read, Alter} \}
\]

- Component set:
  components executed in parallel

\[
\forall R \quad \forall C_1 \neq C_2 \in S: \\
\text{use}(C_1, R) = \text{Alter} \rightarrow \text{use}(C_2, R) = \text{No}
\]
Presenting pattern candidates
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
Random input?

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

- Find out the type of free variables
- QuickCheck generates values by type
Type inference

- Need to find a good type for free variables
  - Not the success type!
  - But describes well the possible values
- Currently we use TypEr
- Working on another approach
Presenting pattern candidates
Cost model

- Approximation
- E.g. for farm:

\[ T_{farm} := T_{work} \cdot \left[ \frac{L}{\min(N_p, N_w)} \right] + T_{spawn} \cdot (N_w + 2) + T_{copy}(L) \cdot 3 + T_{spawn} + T_{copy}(L) \cdot 2 \]

- Needs calibration!
Presenting pattern candidates

Where shall I parallelize? (Tamás Kozsik)
Pattern Candidate Browser

- After ranking pattern candidates
- Web-based interface
  - Information for decision making
  - Not too many details
- Work on better integration with Emacs
- Services
  - Multiple users
  - Persistent results
  - Export XML, JSON, CSV, Erlang terms
Shaping transformations

- Skel is kept simple and composable
- Shaping transformations
  - Accept many syntactic forms
  - Turn them into a canonical form
  - Polishing transformations might be necessary

```
lists:filter(fun pred/1, List)
lists:append([ if pred(Item)-> [Item]; _ -> [] end || Item <- List ])
```
Future work

• Learn from big examples
• Support more skeletons
• Provide better ways to customize
• Add heuristics
• Improve speed
• Make the tool stable and friendly
Conclusions

- ParaPhrase Refactoring Tool for Erlang
  - Wrangler + RefactorErl
- PaRTE can find parallelizable code
  - Discovers pattern candidates
  - Predicts speedup
- PaRTE offers refactorings
- PaRTE + programmer