Thinking Parallel: Generating Parallel Erlang Programs from High-Level Patterns

Kevin Hammond
University of St Andrews, Scotland
Invited Talk at goto; Conference, Zurich, April 2013

T: @paraphrase_fp7
E: kh@cs.st-andrews.ac.uk
W: http://www.paraphrase-ict.eu
The Present

Pound versus Dollar
2013: a ManyCore Odyssey
The Future: “megacore” computers?

- Hundreds of thousands, or millions, of (small) cores
Laki (NEC Nehalem Cluster) and hermit (XE6)

Laki
- 700 dual socket Xeon 5560 2,8GHz (“Gainestown”)
- 12 GB DDR3 RAM / node
- Infiniband (QDR)
- 32 nodes with additional Nvidia Tesla S1070
- Scientific Linux 6.0

hermit (phase 1 step 1)
- 38 racks with 96 nodes each
- 96 service nodes and 3552 compute nodes
- Each compute node will have 2 sockets AMD Interlagos @ 2.3GHz 16 Cores each leading to 113,664 cores
- Nodes with 32GB and 64GB memory reflecting different user needs
- 2.7PB storage capacity @ 150GB/s IO bandwidth
- External Access Nodes, Pre- & Postprocessing Nodes, Remote Visualization Nodes
“Ultimately, developers should start thinking about tens, hundreds, and thousands of cores now in their algorithmic development and deployment pipeline.”

Anwar Ghuloum, Principal Engineer, Intel Microprocessor Technology Lab

“The dilemma is that a large percentage of mission-critical enterprise applications will not `automagically'' run faster on multi-core servers. In fact, many will actually run slower. We must make it as easy as possible for applications programmers to exploit the latest developments in multi-core/many-core architectures, while still making it easy to target future (and perhaps unanticipated) hardware developments.”

Patrick Leonard, Vice President for Product Development
Rogue Wave Software
Doesn’t that mean millions of threads on a megacore machine??
All future programming will be parallel

- No future system will be single-core
  - parallel programming will be essential

- It’s not just about performance
  - it’s also about energy usage

- If we don’t solve the multicore challenge, then all other CS advances won’t matter!
  - user interfaces
  - cyber-physical systems
  - robotics
  - games
  - ...
How to build a wall

(with apologies to Ian Watson, Univ. Manchester)
How to build a wall faster
How NOT to build a wall

Task identification is not the only problem...
Must also consider Coordination, communication, placement, scheduling, ...
We need structure
We need abstraction

We don’t need another brick in the wall
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 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!
A Solution?

“The only thing that works for parallelism is functional programming”

Bob Harper, Carnegie Mellon University
Parallel Functional Programming

- No explicit ordering of expressions
- Purity means no side-effects
  - Impossible for parallel processes to interfere with each other
  - Can debug sequentially but run in parallel
  - **Enormous** saving in effort
- Programmer concentrate on solving the problem
  - Not porting a sequential algorithm into a (ill-defined) parallel domain
- **No locks, deadlocks or race conditions!!**
- **Huge productivity gains!**
- Much shorter code
The ParaPhrase Approach

- Start bottom-up
  - identify (strongly hygienic) **COMPONENTS**
  - *using semi-automated refactoring*

- Think about the **PATTERN** of parallelism
  - e.g. map(reduce), task farm, parallel search, parallel completion, ...

- **STRUCTURE** the components into a parallel program
  - *turn the patterns into concrete (skeleton) code*
  - Take performance, **energy** etc. into account (multi-objective optimisation)
  - also using refactoring

- **RESTRUCTURE** if necessary! *(also using refactoring)*
The ParaPhrase Approach

Pattern Library

Refactorer

Erlang

C/C++

Haskell

Costing/Profiling

Erlang

C/C++

Haskell

...

Nvidia Tesla

Nvidia GPU

AMD Opteron

Nvidia GPU

AMD Opteron

Intel Core

Intel GPU

Intel Core

Intel GPU

Intel Xeon Phi

Mellanox Infiniband
Example: Simple matrix multiplication

- Given two NxN matrices, A and B

\[ A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix} \]

- Their product is

\[ AB = \begin{pmatrix} (AB)_{11} & (AB)_{12} & \cdots & (AB)_{1p} \\ (AB)_{21} & (AB)_{22} & \cdots & (AB)_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (AB)_{n1} & (AB)_{n2} & \cdots & (AB)_{np} \end{pmatrix} \]

where \( (AB)_{ij} = \sum_{k=1}^{m} A_{ik}B_{kj} \).
Example: Simple matrix multiplication

- The sequential Erlang algorithm iterates over the rows
  - \textit{mult} (A, B) multiplies the \textit{rows} of A with the \textit{columns} of B

\[
\text{mult} (\text{Rows, Cols}) \rightarrow [\ \text{mult1row}(R,\text{Cols}) \ | \ | \ R \leftarrow \text{Rows} \ ].
\]

- \[[\ \text{mult1Row}(R,\text{Cols}) \ | \ | \ R \leftarrow \text{Rows} \ ]\) does \text{mult1Row}(R,\text{Cols})\) with R set to each row in turn
Example: Simple matrix multiplication

- The sequential Erlang algorithm iterates over the rows
  - `mult(A, B)` multiplies the *rows* of `A` with the *columns* of `B`
  - `mult1row(R, B)` multiplies one *row* of `A` with all the *columns* of `B`

```erlang
mult(Rows, Cols) -> [ mult1row(R,Cols) || R <- Rows ].
```

```erlang
mult1row (R, Cols) ->
    lists:map(fun(C) -> ... end, Cols).
```

- `lists:map` maps an in-place function over all the columns
Example: Simple matrix multiplication

- The sequential Erlang algorithm iterates over the rows
  - `mult(A, B)` multiplies the rows of A with the columns of B
  - `mult1row(R, B)` multiplies one row of A with all the columns of B
  - `mult1row1col(R, C)` multiplies one row of A with one column of B

```
mult(Rows, Cols) -> [ mult1row(R,Cols) || R <- Rows ].
mult1row (R, Cols) ->
  lists:map(fun(C) -> mult1row1col(R,C) end, Cols).
...
```

- `lists:map` maps an in-place function over all the columns
Example: Simple matrix multiplication

- The sequential Erlang algorithm iterates over the rows
  - \textit{mult}\ (A, B) multiplies the \textit{rows} of A with the \textit{columns} of B
  - \textit{mult1row}\ (R, B) multiplies one \textit{row} of A with all the \textit{columns} of B
  - \textit{mult1row1col}\ (R, C) multiplies one \textit{row} of A with one \textit{column} of B

\begin{verbatim}
mult (Rows, Cols) -> [ mult1row(R,Cols) || R <- Rows ].
mult1row (R, Cols) ->
    lists:map(fun(C) -> mult1row1col(R,C) end, Cols).
multi1row1col(R,C) -> ... \textit{multiply one row by one column} ...
Example: Simple matrix multiplication

- To parallelise it, we can *spawn* a process to multiply each row.

```
mult (Rows, Cols) ->

  ... join(
    [ spawn( fun() -> ... mult1row(R,Cols) end ) || R <- Rows ]
  ).

...
Speedup Results

- 24 core machine at Uni. Pisa
- AMD Opteron 6176. 800 Mhz
- 32GB RAM

![Graph showing speedup results for matrix multiplication. The graph indicates a decrease in speedup as the number of cores increases.](image)
What’s going on?

- We have too many small processes
  - 1,000,000 for our 1000x1000 matrix
  - each process carries setup and scheduling overhead
  - Erlang does not automatically merge processes!
And how can we solve this?

Introduce a \textit{Task Farm}

- A high-level \textit{pattern of parallelism}
- A farmer hands out tasks to a fixed number of worker processes
  - This increases granularity and reduces process creation costs
Some Common Patterns

- High-level abstract patterns of common parallel algorithms
Refactoring changes the structure of the source code using well-defined rules semi-automatically under programmer guidance.
Refactoring: Farm Introduction

\[ S \equiv Farm(S) \]

Farm

Diagram of Farm with three connected components.
Demo: Adding a Farm

```
-module(main).
-export([main/2]).

-define(INT_MAX,2147483647).
-define(RAND_MAX,100).
-define(LCG_A,1664525).
-define(LCG_C,1013904223).

index([HIT], 0) -> H;
index([HIT], N) -> index(T, N-1).

rows(MatrixB) -> MatrixB .

mult([], _) -> [] ;
mult (Rows, Cols) -> [ mult1row(R,Cols) || R <- Rows ].

mult1row(R, Cols) ->
    lists:map(fun(C) ->
        mult1row1col(R, C) end, Cols).

mult1row1col(R, C) ->
    lists:sum([ A*B || {A,B} <- lists:zip(R,C) ]).

cols(MatrixB) ->
    [ index(Row, I) || Row <- MatrixB ]
    || I <- lists:seq(0, length(MatrixB)-1) ].

randvec(0, _) -> [] ;
randvec(Ncols, S) ->

:- main.erl  Top(14,17)  Erlang/OTP 7.0.1 (Erlang/OTP 6.3.0)  [debugger]
```
This uses the new Erlang ‘skel’ Library

Available from

https://github.com/ParaPhrase/skel
Speedup Results

- 24 core machine at Uni. Pisa
- AMD Opteron 6176. 800 Mhz
- 32GB RAM

This is much better!
But I don’t want to give you that...

- I want to give you more...

- There are ways to improve task size further
  - e.g. “chunking” – combine adjacent data items to increase granularity
  - a poor man’s *mapReduce*

- *Just change the pattern slightly!*
Adding Chunking

-module(main).
-export([[main/2]]).

-dlfneme(INT_MAX, 2147483647).
-dlfneme(RAND_MAX, 100).
-dlfneme(LCG_A, 1664525).
-dlfneme(LCG_C, 101390423).

index([[HIT], 0] -> H;
index([[HIT], N] -> index(T, N-1).

rows(MatrixB) -> MatrixB.

mult([], _) -> [];
mult(Rows, Cols) -> skel:run([[fam,
               [[seq, fun (R) -> mult
               24]],
               Rows]).

mult1row(R, Cols) ->
lists:map(fun(C) ->
mult1row1col(R, C) end, Cols).

mult([[R]Rows], Cols) -> [ lists:map(fun(X) -> mult_prime(R, X) end, Cols) l mult(Rows, Cols)].
Speedup Results

- 24 core machine at Uni. Pisa
- AMD Opteron 6176. 800 Mhz
- 32GB RAM

Chunking gives more improvements!
Conclusions

- Functional programming makes it easy to introduce parallelism
  - No side effects means any computation could be parallel
    - millions of *ultra-lightweight* threads (sub micro-second)
  - Matches pattern-based parallelism
  - Much detail can be abstracted
  - automatic mechanisms for granularity control, synchronisation etc

- Lots of problems can be avoided
  - e.g. Freedom from Deadlock
  - Parallel programs give the same results as sequential ones!

- *But still not completely trivial!!*
  - Need to choose granularity carefully!
    - e.g. thresholding
  - May need to understand the execution model
    - e.g. pseq
Isn’t this all just wishful thinking?
NO!

- C++11 has lambda functions
- Java 8 will have lambda (closures)
- Apple uses closures in Grand Central Dispatch
ParaPhrase Parallel C++ Refactoring

- Integrated into Eclipse
- Supports full C++(11) standard
- Uses strongly hygienic components
  - functional encapsulation (closures)
Performance of FastFlow C++ Library

- 5.5 speedup on 12 cores

Compared with 5.1 speedup from a hand-optimised version
Chris Brown, Marco Danelutto, Kevin Hammond, Peter Kilpatrick and Sam Elliot
“Cost-Directed Refactoring for Parallel Erlang Programs”
Proc. 2013 International Symposium on High-level Parallel Programming and Applications (HLPP), Paris, France, June 2013

Chris Brown, Hans-Wolfgang Loidl and Kevin Hammond
“ParaForming: Forming Parallel Haskell Programs using Novel Refactoring Techniques”
Proc. 2011 Trends in Functional Programming (TFP), Madrid, Spain, May 2011

Henrique Ferreiro, David Castro, Vladimir Janjic and Kevin Hammond
“Repeating History: Execution Replay for Parallel Haskell Programs”
Funded by

- **ParaPhrase (EU FP7), Patterns for heterogeneous multicore**, €2.6M, 2011-2014

- **SClEnce (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

- **TACLE: European Cost Action on Timing Analysis**
  - €300K, 2012-2015
Industrial Connections

Mellanox Inc.
Erlang Solutions Ltd
SAP GmbH, Karlsruhe
BAe Systems
Selex Galileo
Biold GmbH, Stuttgart
Philips Healthcare
Software Competence Centre, Hagenberg
Microsoft Research
Well-Typed LLC
THANK YOU!

http://www.paraphrase-ict.eu

http://www.project-advance.eu

@paraphrase_fp7