ParaForming: Forming Parallel Haskell Programs using Novel Refactoring Techniques

Kevin Hammond, Chris Brown
University of St Andrews, Scotland

Hans-Wolfgang Loidl
Heriot-Watt University, Edinburgh, Scotland
Scotland (Schottland)

- St Andrews Uni, 1411
- Glasgow Uni, 1452
- Edinburgh Uni, C18th
- Highlands
- Speyside
- Lowlands

University of St Andrews
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, Karlsruhe
BAe Systems
Selex Galileo
Biold GmbH, Stuttgart
Philips Healthcare
Software Competence Centre, Hagenberg
Mellanox Inc.
Erlang Solutions Lrd
Microsoft Research
Well-Typed
The Dawn of a New Age
Scaling toward Manycore
The future: megacore computers?

- Probably *not* just scaled versions of today’s multicore
  - Perhaps hundreds of dedicated lightweight integer units
  - Hundreds of floating point units (enhanced GPU designs)
  - A *few* heavyweight general-purpose cores
  - Some specialised units for graphics, authentication, network etc
  - possibly *soft* cores (FPGAs etc)
  - *Highly* heterogeneous

- Probably *not* uniform shared memory
  - NUMA is likely, even hardware distributed shared memory
  - or even message-passing systems on a chip
The Implications for Programming

- We must program heterogeneous systems in an *integrated* way

- it will be *impossible* to program each kind of core differently

- it will be *impossible* to take static decisions about placement etc
The Challenge

“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
Parallelism in the Mainstream

- Mostly Procedural
- Parallelism is a “bolt-on” afterthought:
  - Threads
  - Message passing
  - Mutexes
  - Shared Memory
- Results in lots of pain
  - Deadlocks, race conditions, synchronization, non-determinism, etc. etc.
Programming Issues

- We can muddle through on 2-8 cores
  - maybe even 16
  - modified sequential code may work
  - we may be able to use multiple programs to soak up cores
  - BUT larger systems are much more challenging

- 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!
A critique of typical current approaches

- Applications programmers must be systems programmers
  - insufficient assistance with abstraction
  - too much complexity to manage
- Difficult/impossible to scale, unless the problem is simple
- Difficult/impossible to change fundamentals
  - scheduling
  - task structure
  - migration
- Many approaches provide libraries
  - they need to provide abstractions
void *fib(void *fibToFind);
main(){
    pthread_t mainthread;
    long fibToFind = 15;
    long finalFib;
    pthread_create(&mainthread,NULL,fib,(void*)fibToFind);
    pthread_join(mainthread,(void*)&finalFib);
    printf("The number is: %d\n",finalFib);
}
void *fib(void *fibToFind){
    ...
    pthread_t minusone;
    pthread_t minustwo;
    if(newFibToFind == 0 || newFibToFind == 1)
        return newFibToFind;
    else{
        long newFibToFind1 = ((long)fibToFind) - 1;
        long newFibToFind2 = ((long)fibToFind) - 2;
    }
Parallel fib in C

```c
pthread_create(&minusone, NULL, fib, (void*) newFibToFind1);
pthread_create(&minustwo, NULL, fib, (void*) newFibToFind2);

pthread_join(minusone, (void*)&returnMinusOne);
pthread_join(minustwo, (void*)&returnMinustwo);

return returnMinusOne + returnMinustwo;
}
```
The Solution?

Programmers should be able to *think in parallel*
“The only thing that works for parallelism is functional programming”

Bob Harper, Carnegie Mellon
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
Glorious Parallel Haskell

- GpH:
  - Conservative extension of Haskell
  - Implicit parallelism
  - Provides two control primitives

  - \texttt{par} :: a \rightarrow b \rightarrow b
  - \texttt{pseq} :: a \rightarrow b \rightarrow b
fibonacci – In Haskell

\[
\text{fib } n \begin{cases} 
    n \leq 1 & = n \\
    \text{otherwise} & = \text{fib} (n-1) + \text{fib} (n-2) \end{cases}
\]
Parallel fibonacci

\[
pfib \ n \ = \ \begin{cases} 
    n & \text{if } n \leq 1 \\
    \text{otherwise} & \text{otherwise} = n1 \ `\ par` \ n2 \ `\ pseq` \ n1 + n2 \\
\end{cases}
\]

where
\[
    n1 = pfib \ (n-1) \\
    n2 = pfib \ (n-2)
\]

This includes all locking, placement/mapping, synchronisation, communication, lazy thread creation, data/task serialisation, load balancing, thread migration, scheduling, parallel garbage collection, remote caching, and other memory management.
Parallel fib – strategic version

pfib n
  | n <= 1 = 1
  | otherwise = n1 + n2 `using` strat
  where
      n1 = pfib (n-1)
      n2 = pfib (n-2)

      strat res = do { n1' <- rpar n1;
                      n2' <- rpar n2;
                      return res
                         }
How Can We Think Parallel?

- How can we get from `fib` to `pfib`?
- Writing parallel Haskell program still requires
  - Expertise
  - Knowledge
  - Trial and error!?
- Most people do not think parallel
- Tool support?
Refactoring

- Refactoring is about **changing** the **structure** of a program’s **source code**
  - ... while **preserving** the semantics
**Refactoring = Condition + Transformation**

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ensure change at all points needed</td>
<td>Is the refactoring applicable?</td>
</tr>
<tr>
<td>Ensure change at only those points</td>
<td>Will it preserve the semantics of the module?</td>
</tr>
<tr>
<td>needed</td>
<td>The program?</td>
</tr>
</tbody>
</table>
Tool Support - What Do We Need?

- Static Semantics
  - Binding information
- Abstract Syntax Tree
- Lexical structure
- Types
- Modules
- Undo/redo
- Layout Preservation
HaRe: the Haskell Refactorer

- Semi-automated refactoring tool
- Haskell 98
- Structural, data-type and module level
- API for expressing transformations + conditions
- Embedded in vim and emacs
- Uses the Programatica framework...
  ... + Strafunski program transformation library
The Code

- [https://github.com/RefactoringTools/HaRe](https://github.com/RefactoringTools/HaRe)
- `cabal install HaRe`
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. `par` and strategies directly
Refactoring = (in)formal

Renaming an identifier

"The existing binding structure should not be affected. No binding for the new name may intervene between the binding of the old name and any of its uses, since the renamed identifier would be captured by the renaming. Conversely, the binding to be renamed must not intervene between bindings and uses of the new name."

Huiqing Li – PhD Thesis “Refactoring Functional Programs”, 2006
Refactoring Rules

\[ \text{Refactoring}(x_0, ..., x_n) = \{ \text{Rule} \times \{ \text{Condition} \} \} \]

\[
\begin{align*}
D[.] & : \text{Declaration} \rightarrow \text{Declaration} \\
E[.] & : \text{Expr} \rightarrow \text{Expr} \\
T[.] & : \text{Type} \rightarrow \text{Type}
\end{align*}
\]
Introduce Task Parallelism

\( IntroduceTaskPar(\rho, x) = \)
\[ D[ f \overrightarrow{p} = \text{let decls in } e ] \]
\[ \Rightarrow [ f \overrightarrow{p} = \text{let decls in } (\text{let } x' = \text{runEval } \lambda \text{do } \{ x' \leftarrow \text{rpar } x; \text{ return } x' \} \text{ in } e[x'/x])] \]
\[ \{ x \in \rho \lor x \in \text{decls} \lor x \in \overrightarrow{p}, x' \text{fresh} \} \]
Transformation Rule

- Adds `runEval`
- Adds any necessary imports
- Introduces any qualifications to imports
  - `import qualified ...`
Conditions

- $x$ is not a function application
- If expanding
  - runEval must be in scope
  - runEval must be activated
- runEval and rpar must not be hidden ...
Introduction Task Parallelism (2)

\[ IntroduceTaskPar'(\rho, x, d) = \]
\[ \mathcal{D}[f \overset{\rho}{\to} = \text{let decls in } e] \]
\[ \Rightarrow [f \overset{\rho}{\to} = \text{let decls}[d'/d] \text{ in } e[x_{n+1}/x] ] \]
\[ \{x \in \rho \lor x \in \text{decls} \lor x \in \overset{\rho}{\to}, d \in \text{decls}, x \not\in \text{bound}(d), x' \text{ fresh}\} \]

where
\[ d' = [ (x'_0, \ldots, x'_n, x'_{n+1}) = \text{runEval $\{x'_0 \leftarrow \text{rpar } x_0; \ldots; x'_n \leftarrow \text{rpar } x_n; \]
\[ \quad x'_{n+1} \leftarrow \text{rpar } x_{n+1}; \text{return } (x'_0, \ldots, x'_n, x'_{n+1})\} ] \]
\[ d = [ (x'_0, \ldots, x'_n) = \text{runEval $\{x'_0 \leftarrow \text{rpar } x_0; \ldots; x'_n \leftarrow \text{rpar } x_n; \text{return } (x'_0, \ldots, x'_n)\} ] \]
relprime x y = hcf x y == 1

-- parallel versions

-- main refactoring

-- boring sequential version
sumEuler :: Int -> Int
sumEuler = sum . map euler . mkList

sumEulerParList :: Int -> Int
sumEulerParList n = sum (map euler (mkList n)
  `using` parList rdeepseq)

-- naive parallel version w/ parList
sumEulerParListChunk :: Int -> Int -> Int
sumEulerParListChunk n c = sum (map euler (mkList n)
  `using` parListChunk c rdeepseq)

sumEulerParCluster :: Int -> Int -> Int
sumEulerParCluster n z = sum (map euler (mkList n)
  using evalCluster z (ClustStrat (rpar `dot` rdeepseq))

{- OLD versions
-- strategic function application
sumEulerSl :: Int -> Int
sumEulerSl n = sum (map euler (mkList n)
  using`}

(Haskell HaRe 28/06/2010 Ind Doc)--L67--26%------------------------
Demo
sumEulerSeq :: Int -> Int
sumEulerSeq = sum . map euler . mkList

-- divide-and-conquer versions
sumEulerDnc :: Int -> Int
sumEulerDnc n = sumEulerDnc' (mkList n)

sumEulerDnc' :: [Int] -> Int
sumEulerDnc' xs | length xs == 0 = 0
                | length xs == 1 = euler (head xs)
                | otherwise = sumEulerDnc' left + sumEulerDnc' right
                             where (left, right) = splitAt (length xs `div` 2) xs

-- smallest input for euler
base :: Int
base = 0

-- produce a list of input values
mkList :: Int -> [Int]
mkList = reverse . enumFromTo base . (+ base)
-- random numbers
-- mkList seed n = take n (randoms seed)

main :: IO ()
main = putStrLn "HI"

CMD: ["refacClearEvalCache", "/Users/chrisbrown/hare/parallel/example/sumEuler.hs"]
refacClearEvalCache
Completed.
Introduce Threshold

\[ \text{IntroduceThreshold}(\rho, d, x, t, v) = \]
\[ \{ \text{let decls in } e \} \Rightarrow \{ \text{let } d' = \text{abs}/d \text{ in } e \} \]
\[ \{ t \in \rho \lor t \in \text{decls} \lor t \in \overline{p'}, d \in \text{decls}, t \in \text{free}(d), \text{abs fresh}, x \text{ fresh,} \]
\[ \text{typeof}(t) = \text{instanceof}(\text{Ord } a \Rightarrow a), \text{typeof}(x) = \text{instanceof}(\text{Ord } a \Rightarrow a) \} \]

where
\[ d' = \{ (x_0', \ldots, x_n') = \text{runEval} \; \text{do}\{x_0' \leftarrow \text{rabs } x_0; \ldots; x_n' \leftarrow \text{rabs } x_n; \text{return } (x_0', \ldots, x_n') \} \} \]
\[ \text{abs} = \{ \text{rabs } = \text{if } t > x \text{ then rpar else rseq } \} \]
\[ d = \{ (x_0', \ldots, x_n') = \text{runEval} \; \text{do}\{x_0' \leftarrow \text{rpar } x_0; \ldots; x_n' \leftarrow \text{rpar } x_n; \text{return } (x_0', \ldots, x_n') \} \} \]
Introduce Threshold (2)

IntroduceThreshold$(\rho, d, x, t, v) = \begin{cases} \rightarrow p \Rightarrow \{ \} \\ \{ t \in \rho \lor t \in decls \lor t \in \rightarrow p, d \in decls, t \in \text{free}(d), \text{abs fresh}, x \text{ fresh}, \text{typeof}(t) = \text{instanceof}(\text{Ord a} \Rightarrow a), \text{typeof}(x) = \text{instanceof}(\text{Ord a} \Rightarrow a) \} \end{cases}$
Transformation Rules

- Threshold added as an argument
  - First position for partial applications
  - Call sites updated
  - Type signature updated
- runEval updated
  - Parallelism turned on
  - Identity strategy for otherwise
- Imports for rseq
- Qualifications
Conditions

- No conflicts for new threshold name
- Must have an activated Eval monad
- `rseq` must not be hidden in import
- types
Performance Results

- 8 Core Dell PowerEdge
  - 2 quad core Xeon 2.66 GHz
  - 16GB RAM
- Compiled with GHC 6.12 with –o2
- Runtimes averaged over 10 runs
- Preliminary results indicate a speedup of 6.4 on 8 cores
- Some easy steps have given good speedups
- With more scope for tuning, could get a better speedup
Speedup for FFT on 8 cores

![Graph showing speedup for FFT on 8 cores]
Second Example: sumeuler

- Problem: summing Euler totient functions up to some value

```haskell
sumEulerSeq :: Int -> Int
sumEulerSeq = sum . map euler . mkList
```
Data Parallel SumEuler

- Data parallel version
  - use parList
  - add clustering
First eta-expand

```
sumEulerSeq :: Int -> Int
sumEulerSeq = sum . map euler . mkList

sumEulerSeq n = sum (map euler (mkList n))
```
Then Introduce Data Parallelism

```haskell
sumEulerSeq :: Int -> Int
sumEulerSeq n = sum (map euler (mkList n))
```

```haskell
sumEulerPar1 :: Int -> Int
sumEulerPar1 n = sum (map euler (mkList n)
  `using` parList rdeepseq)
```
Now add chunking

```haskell
sumEulerPar1 :: Int -> Int
sumEulerPar1 n = sum (map euler (mkList n)
    'using' parList rdeepseq )

sumEulerParListChunk :: Int -> Int -> Int
sumEulerParListChunk c n = sum (map euler (mkList n)
    'using'
    parListChunk c rdeepseq )
Speedups for sumeuler (data parallel)
Control Parallel Version

- Control parallel version
  - use divide-and-conquer version
  - introduce task parallelism
  - add thresholding
Start with a Divide-and-Conquer Version

```haskell
sumEulerDnc :: [Int] -> Int
sumEulerDnc xs = sumEulerDnc (splitAt (length xs `div` 2) xs)
    where (left, right) = splitAt (length xs `div` 2) xs
          s1 = sumEulerDnc left
          s2 = sumEulerDnc right
          length xs == 0  = 0
          length xs == 1  = euler (head xs)
          otherwise      = s1 + s2
```
Introduce Task Parallelism

```
sumEulerDnc :: [Int] -> Int
sumEulerDnc xs
    | length xs == 0 = 0
    | length xs == 1 = euler (head xs)
    | otherwise = let s1_2 = runEval $ do
                      s1_2 <- rpar s1
                      return (s1_2)
                   in s1_2 + s2

where (left, right) = splitAt (length xs `div` 2) xs
    s1 = sumEulerDnc left
    s2 = sumEulerDnc right
```
Now Introduce Thresholding

```
sumEulerDnc :: Int -> [Int] -> Int
sumEulerDnc t xs  | length xs == 0 = 0
   | length xs == 1 = euler (head xs)
   | otherwise      = let (s1_2, s2_2) = runEval $ do
   s1_2 <- (rabs 'dot' rdeepseq) s1
   s2_2 <- (rabs 'dot' rdeepseq) s2
   return (s1_2, s2_2)

   rabs = if (lenXs > t) then rpar else rseq
   in s1_2+s2_2

where (left, right) = splitAt (length xs 'div' 2) xs
s1 = sumEulerDnc t left
s2 = sumEulerDnc t right
lenXs = length xs
```
Speedups for \texttt{sumEuler} (task parallel)
The ParaPhrase Project (ICT-2011-288570)

€3.5M FP7 STReP Project
9 partners in 5 countries
3 years
Starts 1/10/11
Coordinated from St Andrews
### Project Consortium

<table>
<thead>
<tr>
<th>Project partners</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>University of St Andrews</td>
<td>UK</td>
</tr>
<tr>
<td>Robert Gordon University</td>
<td>UK</td>
</tr>
<tr>
<td>Mellanox Technologies Ltd.</td>
<td>Israel</td>
</tr>
<tr>
<td>Software Competence Center Hagenberg</td>
<td>Austria</td>
</tr>
<tr>
<td>Erlang Solutions Ltd</td>
<td>UK</td>
</tr>
<tr>
<td>University of Pisa</td>
<td>Italy</td>
</tr>
<tr>
<td>High Performance Computing Center Stuttgart</td>
<td>Germany</td>
</tr>
<tr>
<td>University of Torino</td>
<td>Italy</td>
</tr>
<tr>
<td>Queen’s University Belfast</td>
<td>UK</td>
</tr>
</tbody>
</table>
How to think in Parallel

- Direct programming in e.g. Parallel Haskell
- Parallel stream-based approaches
- Coordination approaches
- Pattern-based approaches

*Avoid* issues such as deadlock etc...

*Parallelism by Construction!*
Project Vision

Application Design

Pattern-based Development/Refactoring

Parallelised Application

Parallelised Application

Parallelised Application

Dynamic Mapping

Heterogeneous Hardware Pool
The ParaPhrase Model

Costing/ profiling

C/C++

Erlang

Haskell

Patterns

C/C++

Erlang

Haskell
Example: MasterWorker pattern
Patterns of Symbolic Computation

- **Standard functional algorithmic skeletons**
  
  - `parMap:: (a->b) -> [a] -> [b]`
  - `parZipWith:: (a->b->c) -> [a] -> [b] -> [c]`
  - `parReduce:: (a->b->b) -> b -> [a] -> b`
  - `parMapReduce:: (a->b->b) -> (c->[((d,a)]) -> c -> [((d,b)])`
  - `masterWorker:: (a->[([a],b)) -> [a] -> [b]`

- **New parallel domain-specific patterns**
  
  - **orbit calculation:** generate unprocessed neighbouring states
  - **duplicate elimination:** merge two lists
  - **completion algorithm:** generate new objects from any pair
  - **chain reduction:** generate new objects from any pair
  - **partition backtracking:** search for *basis objects*

- **others?? search skeleton, classification skeleton, modular skeleton+CRA, backtracking search**
Refactoring from Patterns

Objective 5: Adaptive Mapping Technology.

Objective 5 is to develop methods to map software components onto the resources of a heterogeneous multi-core platform, matching them against the available hardware characterisation that is exposed through the hardware/software virtualisation layers. This mapping needs to take into account both computations, which will be mapped to the available hardware resources, and any communication that is induced by this mapping. In this way, we will achieve our aim of developing new dynamic mechanisms to support adaptivity and heterogeneity.
Static Mapping

Implementation in Software Virtualisation Layer

Implementation on Heterogeneous Platform
Dynamic Re-Mapping
Feedback-Directed Compilation

Compilation Route

Source

Compiler

Object + LPEL + CAL

Mapping

Object + SVP

Hardware Platform

Dynamic Adaptation

Runtime Measurement

Analysis Route

Source + CAL

Statistical Analysis

Markov Model

Performance Aggregation

Measurements + SVP

Feedback Route
ParaPhrase Research Directions

- What patterns are needed to cover our applications?
  - standard patterns
  - domain-specific patterns
  - special patterns for heterogeneity

- Can we program heterogeneous systems without knowing the target?
  - what virtualisation mechanisms do we need
  - abstract memory access, communication, state

- What information is needed to exploit multicore/manycore effectively?
  - metrics: execution time, memory, power
  - historical v. predicted information

- Is static or dynamic mapping best?
ParaPhrase Refactorer

- Use cost/profiling information to guide the user
  - Issue warnings
  - Suggest which pattern to apply
- Switch between a parallel and a sequential view
- Improves understanding of writing parallel programs
  - “think in parallel”
Conclusions

- Functional Programming
- Purity
- Refactoring

makes parallelism much easier
good speedups are possible
Conclusions

- 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

- Need technology:
  - Compilers, traversals, DSLs

Brown. Loidl and Hammond
“ParaForming: ...”
To Appear in Proc. 2011 Trends in Functional Programming (TFP), Madrid, Spain
Future Work

- More parallel refactorings
- Transformations between GpH/GdH/Eden...
- Database of parallel skeleton templates
- Refactoring support for parallel patterns
  - New (unified) framework for C/C++/Erlang/etc.
- Refactoring language (DSL) for expressing transformations + conditions
  - Language for expressing patterns?
- Cost directed refactoring
- Proving refactorings correct
Parallel Haskell: Lightweight Parallelism for Heavyweight Functional Programs

(DRAFT – please do not redistribute without permission.)

Kevin Hammond, Chris Brown and Phil Trinder

In Preparation
THANK YOU!

http://www.paraphrase-ict.eu

@paraphrase_fp7
Modify Evaluation Degree

\[\text{ModifyEvalDegree}(\rho, x) =\]
\[\varepsilon[ f \; x ] \Rightarrow [ (f \; \text{dot'} \; \text{rdeepseq}) \; x ]\]
\[\{ \text{typeof}(f) = \text{Strategy} \; a, \text{rdeepseq} \in \rho, \text{dot} \in \rho \}\]

\[\triangleright\]
\[\tau[ x :: \tau ] \Rightarrow [ x :: \text{NFData} \; \tau \Rightarrow \tau ]\]
\[\{\}\]