Profiling-Based Characterisation of Glasgow Parallel Haskell Applications

Evgenij Belikov\textsuperscript{1}, Hans-Wolfgang Loidl, Greg Michaelson

Glasgow Parallelism Group Seminar

March 26, 2014

\textsuperscript{1}eb120@hw.ac.uk
Heterogeneous Architectures and Performance Portability

Motivation
- Parallelism is a key source of performance
- Parallel architectures are increasingly heterogeneous and hierarchical
- Hardware evolves faster than software
- High-level languages appear most promising in balancing productivity and performance across diverse architectures

Key Research Question
- Design and implement run-time system support for adaptive policy control based on a system cost model for a high-level parallel functional language (Glasgow Parallel Haskell)
- Quantify the effects of using cost models wrt performance portability and scalability on modern architectures
Cost-Model-Based Adaptive Policy Control for GUM

Finalised
- Monitoring and Profiling

Ongoing Work
- Application Classification
- Cost Modelling

Next Steps
- Tuning: Load Balancing, Scheduling, Granularity
- Empirical Evaluation on Heterogeneous Architectures

Figure: per-PE Load Profile

Figure: GUM Control Model
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging

```
par :: a -> b -> b  pseq :: a -> b -> b
pfib n = x 'par' y 'pseq' (x + y)
  where x = pfib (n-1)
        y = pfib (n-2)
```
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging
- High-level, semi-explicit, architecture-independent, functional

```
par :: a -> b -> b
pseq :: a -> b -> b
pfib n = x 'par' y 'pseq' (x + y)
    where x = pfib (n-1)
        y = pfib (n-2)
```
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging
- High-level, semi-explicit, architecture-independent, functional
- Modest extension, deterministic programming model

```
par :: a -> b -> b
pseq :: a -> b -> b
pfib n = x 'par' y 'pseq' (x + y)
  where x = pfib (n-1)
        y = pfib (n-2)
```
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging
- High-level, semi-explicit, architecture-independent, functional
- Modest extension, deterministic programming model
- Based on Haskell, strongly typed, purely-functional language

```
par :: a -> b -> b  pseq :: a -> b -> b
pfib n = x 'par' y 'pseq' (x + y)
  where x = pfib (n-1)
       y = pfib (n-2)
```
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging
- High-level, semi-explicit, architecture-independent, functional
- Modest extension, deterministic programming model
- Based on Haskell, strongly typed, purely-functional language
- State-of-the-art compiler (GHC), sophisticated RTS (SMP, GUM)

```haskell
par :: a -> b -> b  pseq :: a -> b -> b
pfib n = x 'par' y 'pseq' (x + y)
  where x = pfib (n-1)
       y = pfib (n-2)
```
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging
- High-level, semi-explicit, architecture-independent, functional
- Modest extension, deterministic programming model
- Based on Haskell, strongly typed, purely-functional language
- State-of-the-art compiler (GHC), sophisticated RTS (SMP, GUM)
- Evaluation Strategies and Algorithmic Skeletons

```
par :: a -> b -> b
pseq :: a -> b -> b

pfib n = x 'par' y 'pseq' (x + y)
  where x = pfib (n-1)
       y = pfib (n-2)
```
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
- Implicit synchronisation and communication (PVM or MPI)
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
- Implicit synchronisation and communication (PVM or MPI)
- Random work stealing (decentralised load balancing)
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
- Implicit synchronisation and communication (PVM or MPI)
- Random work stealing (decentralised load balancing)
- Thread subsumption (akin to lazy futures for granularity control)
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
- Implicit synchronisation and communication (PVM or MPI)
- Random work stealing (decentralised load balancing)
- Thread subsumption (akin to lazy futures for granularity control)
- Isolated heaps (parallel garbage collection)
Graph Reduction on a Unified Machine Model (GUM)

- Supports shared-memory and distributed-memory by using virtual shared memory abstraction
- Implicit synchronisation and communication (PVM or MPI)
- Random work stealing (decentralised load balancing)
- Thread subsumption (akin to lazy futures for granularity control)
- Isolated heaps (parallel garbage collection)
- Geared towards dynamic parallelism management
Application Characterisation

- Sanity check: diverse and representative applications
- Identify bottlenecks
- Discover characteristics to be monitored
- Select parameters to be tuned
- (Motivate the need for Parallel Haskell Benchmark Suite)
Applications

- adopted from the nofib suite and 'Seq no more' paper
- parfib, computes the number of calls for fib 50 23, regular d&c
- coins, d&c, permutation search (input: 5777)
- queens, d&c, nqueens problem (16x16 board, depth 3)
- minimax, d&c, alpha-beta search (4x4 board, depth 8)
- worpitzky, d&c, symbolic computation (input: 19 27 10; arbitrary length integers)
- sumeuler, data parallel, moderately irregular ([0..100k], chunk 500)
- maze, nested data parallel, uses speculative parallelism (size 29)
Architectures

Multi-core (focus in this talk), Beowulf cluster, co-processors

<table>
<thead>
<tr>
<th></th>
<th>levels</th>
<th>cores</th>
<th>speed (GHz)</th>
<th>cache (MB)</th>
<th>RAM (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>cantor</td>
<td>5</td>
<td>48</td>
<td>2.3</td>
<td>2 L2 + 6 L3&lt;sup&gt;2&lt;/sup&gt;</td>
<td>64x8</td>
</tr>
<tr>
<td>bwlf</td>
<td>3</td>
<td>32x8</td>
<td>2.0 or 3.0</td>
<td>(256KB+4) or 6</td>
<td>12 or 16</td>
</tr>
<tr>
<td>phi</td>
<td>3</td>
<td>60</td>
<td>1.1</td>
<td>30</td>
<td>8</td>
</tr>
</tbody>
</table>

Software: CentOS 6.5, Gigabit Ethernet, ghc 6.12.3<sup>3</sup>, gcc 4.4.7, pvm 3.4.6
Run times: median of three; relative speedups

- Run-time systems: GHC-SMP vs GHC-GUM
- Fixed input scaling (several input sizes)
- Preliminary thresholding/chunking experiments (not included)

<sup>2</sup>L2 is shared by 2 cores, L3 by six
<sup>3</sup>using ghc 7.6 improves SMP scaling but shows same overall trends
Attributes aka Characteristics

- Performance (Execution Time)
- Scalability (Speedup, potential parallelism: #sparks)
- Memory Use (Heap residency, Allocation Rate, GC overhead)
- Granularity (actual parallelism: #threads, thread granularity)
- Communication (rate in messages per sec elapsed, #packets, #global addresses, bytes of graph sent, % of work requests of the total; blocking and fetching times and counts on per-thread basis)
Run Times

![Graph showing comparison of GUM vs SMP with different applications and number of PEs](image)

- ssumeuler-GUM
- ssumeuler-SMP
- coins-GUM
- coins-SMP
- parfib-GUM
- parfib-SMP
- worpitzky-GUM
- worpitzky-SMP
- queens-GUM
- queens-SMP
- minimax-GUM
- minimax-SMP
- maze-GUM
- maze-SMP

The graph compares the run times of different applications using Glasgow Parallel Haskell (GpH) on a multi-core processor. The x-axis represents the number of PEs (Processing Elements), and the y-axis shows the run time (in seconds). The graph illustrates how different applications perform under various parallel configurations.
Run Times

- Parallelism is not for free (can exploit some, but not all)
Run Times

- Parallelism is not for free (can exploit some, but not all)
- In some cases order of magnitude decrease in run time
Run Times

- Parallelism is not for free (can exploit some, but not all)
- In some cases order of magnitude decrease in run time
- In other cases slowdown occurs for large numbers of PEs
Speedups
## Degree of Parallelism: Actual vs Potential

<table>
<thead>
<tr>
<th>application</th>
<th>number of threads on N cores</th>
<th>total sparks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>sumeuler-GUM</td>
<td>135</td>
<td>171</td>
</tr>
<tr>
<td>sumeuler-SMP</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>minimax-GUM</td>
<td>12</td>
<td>161</td>
</tr>
<tr>
<td>minimax-SMP</td>
<td>5</td>
<td>31</td>
</tr>
<tr>
<td>queens-GUM</td>
<td>14</td>
<td>57</td>
</tr>
<tr>
<td>queens-SMP</td>
<td>5</td>
<td>69</td>
</tr>
<tr>
<td>parfib-GUM</td>
<td>4</td>
<td>38</td>
</tr>
<tr>
<td>parfib-SMP</td>
<td>20</td>
<td>136</td>
</tr>
<tr>
<td>coins-GUM</td>
<td>13</td>
<td>36</td>
</tr>
<tr>
<td>coins-SMP</td>
<td>3</td>
<td>93</td>
</tr>
<tr>
<td>worpitzky-GUM</td>
<td>55</td>
<td>401</td>
</tr>
<tr>
<td>worpitzky-SMP</td>
<td>162</td>
<td>565</td>
</tr>
<tr>
<td>maze-GUM</td>
<td>4</td>
<td>15</td>
</tr>
<tr>
<td>maze-SMP</td>
<td>3</td>
<td>52</td>
</tr>
</tbody>
</table>
Granularity: parfib (zoom on 90%) (1/2)
Granularity: parfib (zoom on 90%) (2/2)
Granularity: parfib (zoom on 50%)
Granularity: Shape – Same or Different?
Granularity: Relative View

GUM

% of total elapsed RT

SMP

% of total elapsed RT

% of total count

E. Belikov (Heriot-Watt University)  Characterising GpH Applications  March 26, 2014
Memory Use: Garbage Collection (median, % of elapsed)
Memory Use: Heap Residency (median, KB or MB)
Memory Use: Allocation Rate (on PE1, GB / MUT sec)
Communication-to-Computation Ratio

![Communication-to-Computation Ratio Graph]

- **Maze**
- **Parfib**
- **Coins**
- **Minimax**
- **Sumeuler**
- **Worpitzky**
- **Queens**

- **Y-axis**: messages per second elapsed
- **X-axis**: number of PEs

The graph shows the communication rates for different applications as the number of PEs increases.
Application Diversity: on GUM using 48 PEs

The diagram shows the application similarity (GUM on 48 PEs) with the allocation rate on PE1 (GB/MUT sec) on the y-axis and the communication rate (messages/sec elapsed) on the x-axis. The applications tested include coins, maze, minimax, parfib, sumeuler, worpitzky, and queens.
Findings

- Most applications manage to exploit up to 8 cores, only a few scale beyond 16, and only 2 have efficiency close to 50% on 48 PEs.

\(^4\) GC\% consistently increases with PEs for all applications for SMP.
Findings

- Most application manage to exploit up to 8 cores, only few scale beyond 16, only 2 have efficiency close to 50% on 48 PEs
- Main bottlenecks: GC (due to high allocation rate and many threads) and communication (graph sharing)

\[^4\text{GC\% consistently increases with PEs for all applications for SMP}\]
Findings

- Most application manage to exploit up to 8 cores, only few scale beyond 16, only 2 have efficiency close to 50% on 48 PEs
- Main bottlenecks: GC (due to high allocation rate and many threads) and communication (graph sharing)
- SMP is more aggressive than GUM in turning sparks into threads

\(^4\) GC% consistently increases with PEs for all applications for SMP
Findings

- Most applications manage to exploit up to 8 cores, only few scale beyond 16, only 2 have efficiency close to 50% on 48 PEs.
- Main bottlenecks: GC (due to high allocation rate and many threads) and communication (graph sharing).
- SMP is more aggressive than GUM in turning sparks into threads.
- GUM produces less and larger threads in most cases (thread subsumption, in particular for D&C).

\[ \text{GC}\% \text{ consistently increases with PEs for all applications for SMP.} \]
Findings

- Most applications manage to exploit up to 8 cores, only a few scale beyond 16, and only 2 have efficiency close to 50% on 48 PEs.
- Main bottlenecks: GC (due to high allocation rate and many threads) and communication (graph sharing).
- SMP is more aggressive than GUM in turning sparks into threads.
- GUM produces less and larger threads in most cases (thread subsumption, in particular for D&C).
- Due to distributed heaps, GUM GC overhead is roughly constant, unlike SMP\(^4\); this leads to GUM outperforming SMP in most cases (both in term of run time and scalability).

\(^4\)GC% consistently increases with PEs for all applications for SMP.
Related Work

- GUMSMP (Multi-Level RTS for GpH; Aljabri, Loidl, Trinder)
- HdpH (Distributed Haskell in Haskell; Maier, Stewart, Trinder)
- HWSkel (Cost Models for Hybrid Skeletons; Armih et al.)
- Locality-aware PrimOps for GpH (e.g. parDist, Aswad et al.)
- Auto Tuning Skeletons using Machine Learning (CARD group at Edinburgh Uni, O’Boyle et al)
- Atlas, GotoBLAS (Auto-tuning Linear Algebra kernels; Patterson et al, Goto et al)
- hwloc, likwid, etc (Architecture Discovery; e.g. Broquedis et al.)
Limitations

- no large applications used
Limitations

- no large applications used
- seven application is not enough for a classification
Limitations

- no large applications used
- seven application is not enough for a classification
- other important parallelism patterns missing (e.g. pipeline)
Limitations

- no large applications used
- seven application is not enough for a classification
- other important parallelism patterns missing (e.g. pipeline)
- rather homogeneous architectures used
Limitations

- no large applications used
- seven application is not enough for a classification
- other important parallelism patterns missing (e.g. pipeline)
- rather homogeneous architectures used
- results are indicative rather than conclusive
Future Work

- Making the numbers dance!
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
- Add load balance and degree of irregularity as further high-level characteristics
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
- Add load balance and degree of irregularity as further high-level characteristics
- Cost model integration in GUM
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
- Add load balance and degree of irregularity as further high-level characteristics
- Cost model integration in GUM
- Devise and evaluate RTS-level architecture-aware policies
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
- Add *load balance* and *degree of irregularity* as further high-level characteristics
- Cost model integration in GUM
- Devise and evaluate RTS-level architecture-aware policies
- Investigate Haskell compilation to Xeon Phi
Future Work

- Making the numbers dance!
- Add further applications with different characteristics
- Add load balance and degree of irregularity as further high-level characteristics
- Cost model integration in GUM
- Devise and evaluate RTS-level architecture-aware policies
- Investigate Haskell compilation to Xeon Phi
- (Bring GUM RTS in line with GHC HEAD)
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
- Self-Tuning Patterns (offline vs online)
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
- Self-Tuning Patterns (offline vs online)
- Architecture-awareness at RTS level
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
- Self-Tuning Patterns (offline vs online)
- Architecture-awareness at RTS level
- Characteristics to monitor (communication, FISHing and sparking rates, GA residency)
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
- Self-Tuning Patterns (offline vs online)
- Architecture-awareness at RTS level
- Characteristics to monitor (communication, FISHing and sparking rates, GA residency)
- Parameters to tune (packet size, FISH delay, number of outstanding FISHes, watermarks)
Some Ideas

- Detecting parallel pattern in use (regular vs irregular)
- Self-Tuning Patterns (offline vs online)
- Architecture-awareness at RTS level
- Characteristics to monitor (communication, FISHing and sparking rates, GA residency)
- Parameters to tune (packet size, FISH delay, number of outstanding FISHes, watermarks)
- Cost-model integration (analytical vs empirical)
Invitation

Please do visit us at Heriot-Watt
there are plenty of free DSG Seminar slots!
Send an email to S.Scholz@hw.ac.uk or myself