Profiling-Based Characterisation of Glasgow Parallel Haskell Applications

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

Seminar on Parallelism in PGAS and Functional Programming Languages

May 22, 2014

\textsuperscript{1}eb120@hw.ac.uk
1. The Quest for Performance Portability

2. Applications, Architectures, and Attributes

3. Characterisation of Glasgow Parallel Haskell Applications

4. Conclusions and Future Work

5. Discussion
Heterogeneous Architectures and Performance Portability

Motivation
- Parallelism is a key source of performance but hard to exploit
- 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

Application Characterisation
- Improves understanding of behaviour of parallel functional programs
- Compares applications across architectures and run-time systems
- Helps design dynamic adaptive policy control for a high-level parallel functional language (Glasgow Parallel Haskell)
Why Glasgow Parallel Haskell?

- Full auto-parallelisation is exceptionally challenging

```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

```
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

```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

```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
- 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
Applications\textsuperscript{2}

- parfib, regular d\&c, number of calls for fib 50 23
- 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, fairly irregular ([0..100k], chunk 500)
- maze, nested data parallel, uses speculative parallelism (size 29)
- mandelbrot, data parallel, high communication rate

\textsuperscript{2}mostly adopted from the \textit{nofib} suite and \textit{Seq no more} paper
Architectures

Server-class multi-core (cantor), Beowulf-class cluster (beowulf)

<table>
<thead>
<tr>
<th></th>
<th>levels</th>
<th>cores</th>
<th>speed (GHz)</th>
<th>cache (MB)</th>
<th>RAM (GB)</th>
<th>latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>cantor</td>
<td>5</td>
<td>48</td>
<td>2.3</td>
<td>2 L2 + 6 L3 $^3$</td>
<td>64x8</td>
<td>10-22ns</td>
</tr>
<tr>
<td>beowulf</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>
<td>ca. 150ns</td>
</tr>
</tbody>
</table>

Software: CentOS 6.5, Gigabit Ethernet, ghc 6.12.3$^4$, 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

$^3$L2 is shared by 2 cores, L3 by six

$^4$using ghc 7.6 improves SMP scaling but shows same overall trends
Attributes aka Characteristics

- Performance (Execution Time)
- Scalability (Speedup, potential parallelism: \#sparks)
- Granularity (actual parallelism: \#threads, thread granularity)
- Memory Use (Heap and GA residency, Allocation Rate, GC overhead)
- 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

Characterisation of Glasgow Parallel Haskell Applications

E. Belikov (Heriot-Watt University)

May 22, 2014

Characterising GpH Applications

Run Times

PEs 1 2 4 8 16 32 48 64

Execution time (sec)

parfib

queens

coins

sumeuler (interval-based)

worpitzky

maze

minimax

mandelbrot

GUM on beowulf  GUM on cantor  SMP on cantor
Speedups

- partfib
- coins
- semeuler
- worpitzky
- maze
- minimax
- mandelbrot

PEs: 1, 2, 4, 8, 16, 32, 48, 64

Speedup: GUM on beowulf, GUM on cantor, SMP on cantor
## 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>8</td>
<td>16</td>
</tr>
<tr>
<td>sumeuler-GUM</td>
<td>186</td>
<td>193</td>
</tr>
<tr>
<td>sumeuler-SMP</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>minimax-GUM</td>
<td>69</td>
<td>139</td>
</tr>
<tr>
<td>minimax-SMP</td>
<td>92</td>
<td>115</td>
</tr>
<tr>
<td>queens-GUM</td>
<td>146</td>
<td>281</td>
</tr>
<tr>
<td>queens-SMP</td>
<td>153</td>
<td>201</td>
</tr>
<tr>
<td>mandelbrot-GUM</td>
<td>1823</td>
<td>2261</td>
</tr>
<tr>
<td>mandelbrot-SMP</td>
<td>7224</td>
<td>22128</td>
</tr>
<tr>
<td>parfib-GUM</td>
<td>89</td>
<td>231</td>
</tr>
<tr>
<td>parfib-SMP</td>
<td>881</td>
<td>11127</td>
</tr>
<tr>
<td>coins-GUM</td>
<td>170</td>
<td>633</td>
</tr>
<tr>
<td>coins-SMP</td>
<td>3687</td>
<td>7478</td>
</tr>
<tr>
<td>worpitzky-GUM</td>
<td>1430</td>
<td>4508</td>
</tr>
<tr>
<td>worpitzky-SMP</td>
<td>12113</td>
<td>49979</td>
</tr>
<tr>
<td>maze-GUM</td>
<td>116</td>
<td>454</td>
</tr>
<tr>
<td>maze-SMP</td>
<td>1172</td>
<td>3029</td>
</tr>
</tbody>
</table>
Granularity (1/2)

- **parfib**
  - mts = 0.04
  - mts = 0.12
  - mts = 0.015

- **coins**
  - mts = 0.04
  - mts = 0.06
  - mts = 0.64

- **worpitzky**
  - mts = 0.001
  - mts = 0.002
  - mts = 0.004

- **minimax**
  - mts = 0.18
  - mts = 0.26
  - mts = 2.92

* mts \(\equiv\) median thread size

Number of Threads vs. GHCSM per cantor

Thread Granularity (ms)
Granularity (2/2)

* mts := median thread size
Memory Use: Garbage Collection (median, % of elapsed)

![Garbage Collection Graph](image_url)
Memory Use: Heap Residency (median, KB or MB)

GUM on beowulf
- queens (MB)
- sumeuler
- parfib
- mandelbrot (MB)
- worpitzky
- minimax (MB)
- maze
- coins

GUM on cantor

SMP on cantor
Memory Use: Allocation Rate (on PE1, GB / MUT sec)
Global Address Table Residency and Fragmentation
Communication-to-Computation Ratio

The graph shows the communication-to-computation ratio for different applications as a function of the number of processing elements (PEs). The applications include 'queens', 'surrender', 'parfib', 'mandelbrot', 'warpirzky', 'minimax', 'maze', and 'corns'. The x-axis represents the number of PEs ranging from 8 to 64, and the y-axis represents the messages per second.
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is $> 70\%$ on 64 PEs
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is > 70% on 64 PEs
- Profiling reveals bottlenecks: GC (high residency rate, many small threads) and communication (GA residency, many messages)
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is $> 70\%$ on 64 PEs
- Profiling reveals bottlenecks: GC (high residency rate, many small threads) and communication (GA residency, many messages)
- Threads subsumption works, much superfluous parallelism is pruned
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is > 70% on 64 PEs
- Profiling reveals bottlenecks: GC (high residency rate, many small threads) and communication (GA residency, many messages)
- Threads subsumption works, much superfluous parallelism is pruned
- GUM adapts to latency and is significantly less aggressive in instantiating parallelism than SMP
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is $> 70\%$ on 64 PEs
- Profiling reveals bottlenecks: GC (high residency rate, many small threads) and communication (GA residency, many messages)
- Threads subsumption works, much superfluous parallelism is pruned
- GUM adapts to latency and is significantly less aggressive in instantiating parallelism than SMP
- Distributed-memory RTS design (private heaps) appears to scale better than shared-memory design (due to contention)
Findings

- Exploiting parallelism: order of magnitude lower run time for 5 of 8 programs; for two efficiency is $> 70\%$ on 64 PEs
- Profiling reveals bottlenecks: GC (high residency rate, many small threads) and communication (GA residency, many messages)
- Threads subsumption works, much superfluous parallelism is pruned
- GUM adapts to latency and is significantly less aggressive in instantiating parallelism than SMP
- Distributed-memory RTS design (private heaps) appears to scale better than shared-memory design (due to contention)
- System-level and architectural information appears potentially useful for adaptive policy control at run time
Related Work

- GUMSMP (Multi-Level RTS for GpH; Aljabri, Loidl, Trinder)
- HdpH (Distributed Haskell in Haskell; Maier, Stewart, Trinder)
- MetaPar (Foltzer et al.)
- HWSkel (Cost Models for Hybrid Skeletons; Armih et al.)
- Locality-aware PrimOps for GpH (e.g. parBounded, Aswad et al.)
- Auto Tuning Skeletons using Machine Learning (CARD group at Edinburgh, 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-scale applications used
- eight applications 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

- Add larger applications with different characteristics/patterns
- Add *load balance* and *degree of irregularity* as further high-level characteristics
- Run experiments on more heterogeneous architectures (e.g. by adding co-processors)
- Cost model integration in GUM
- Devise and evaluate RTS-level architecture-aware policies
Some Policy Control Ideas

- Spark co-location (place sparks of the same source together)
- Throttling of stealing (if FISHing failure rate is high)
- Distinguish generators (high sparking rate) from workers (more stealing than sparking) and use different policies for load balancing (e.g. temporary work pushing for generators)
- Architecture-awareness at RTS level (using cost models)