Profiling-Based Characterisation of Glasgow Parallel Haskell Applications

Evgenij Belikov\(^1\), Hans-Wolfgang Loidl, Greg Michaelson

Symposium on Trends in Functional Programming (TFP14), Soesterberg, NL

May 27, 2014

\(^1\)eb120@hw.ac.uk
1. Performance Portability

2. Applications, Architectures, Characteristics

3. Characterisation of Parallel Haskell Applications

4. Conclusions and Future Work
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

- Increases understanding of behaviour of parallel functional programs
- Compares dynamic characteristics across parallelism patterns, architectures, and run-time systems (RTS)
- Helps improve adaptive parallelism management for a high-level semi-explicit parallel programming language
Glasgow Parallel Haskell

- Based on Haskell: State-of-the-art compiler and RTS (GHC)
- High-level, **semi-explicit**, **architecture-independent**, functional
- **Deterministic** programming model
- **Advisory** parallelism (sparking akin to lazy task creation)
- Composable abstractions: Skeletons, Evaluation Strategies

```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)
```
Applications²

Divide and Conquer

- parfib: regular, number of calls for fib 50 23
- coins: permutation search (input: 5777)
- queens: nqueens problem (16x16 board, depth 3)
- minimax: alpha-beta search (4x4 board, depth 8)
- worpitzky: symbolic computation, multiple sources of parallelism (input: 19 27 10; arbitrary length integers)

Data Parallel

- sumeuler: irregular ([0..100k], chunk 500)
- mandelbrot: irregular (4096 x 4096 image)
- maze: nested, uses speculative parallelism (size 29)

²mostly adopted from the nofib suite and Seq no more paper
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 6.12.3⁴, gcc 4.4.7, pvm 3.4.6

- Run-time systems: GHC-SMP vs GHC-GUM
- Run times: median of three; relative speedups
- 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
Characteristics

- Performance (Execution time)
- Scalability (Speedup, actual vs potential parallelism)
- Granularity (Thread size)
- Memory Use (Heap and GA residency, allocation rate, GC%)
- Communication (Messages per sec, fetching time)
Run Times

![Graphs showing run times for different applications and configurations.](image-url)
Characterisation of Parallel Haskell Applications

Speedups

- parfib
- coins
- smergeur
- worpitzky
- maze
- minimax
- mandelbrot

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

Graphs showing speedups for different applications with varying number of processors.

Legend:
- Red: GUM on beowulf
- Blue: GUM on cantor
- Yellow: SMP on cantor

* < 1.0 (timeout)
Findings: Performance and Scalability

- Order of magnitude reduction in run time (5 out of 8 programs)
- Speedup flattens out for GUM, slowdown for SMP for higher PE numbers (due to lack of work or excessive overhead)
- Mostly abundant potential parallelism available (10e6 sparks) (D&C often has more sparks than data-parallel\textsuperscript{5})
- Actual parallelism order(s) of magnitude lower (10e4 lightweight threads; \textit{thread subsumption})

\textsuperscript{5} also depends on the application-level thresholding/chunking
Granularity on 48PEs (1/2)

* mts := median thread size

- parfib
- coins
- worpitzky
- minimax
Granularity on 48PEs (2/2)

* mts := median thread size
Findings: Granularity

- Thread subsumption more effective for D&C and nested than for flat data-parallel applications
- GUM on beowulf has similar profile to GUM on cantor
- GUM on cantor significantly differs from SMP on cantor
- In most cases fewer and larger threads for GUM than for SMP
  ⇒ optimisation potential: reducing the number of small threads
Memory Use: Garbage Collection (median, % of elapsed)
Memory Use: Allocation Rate (on PE1, GB / MUT sec)

Graphs showing allocation rates for different algorithms and numbers of PEs in both Divide and Conquer and Data-Parallel applications.

- GUM on beowulf
- GUM on cantor
- SMP on cantor

E. Belikov (Heriot-Watt University)
Findings: Memory Use

- High behavioural diversity across programs and parallelism patterns
- GC% and heap residency:
  - constant or decreasing for GUM
  - increasing for SMP
- Allocation rate:
  - constant for GUM on beowulf
  - constant, then decreasing for high numbers of PEs for GUM on cantor
  - increasing for small PE number, then dropping rapidly for SMP (due to contention on the first generation heap)
Characterisation of Parallel Haskell Applications

Global Address Table Residency (Fragmentation)

Divide and Conquer Applications

Data-Parallel Applications

E. Belikov (Heriot-Watt University)

May 27, 2014 17 / 20
Communication Rate

GUM on beowulf

GUM on cantor

Divide and Conquer Applications

Data-Parallel Applications
Findings: Communication

- High communication rate limits scalability for both D&C and data parallel applications
- For most application we have small packets and linearly increasing communication rate
- **High GA residency increases communication overhead, indicating reduced locality due to fragmentation**
- Parallelism is often instantiated in the beginning of execution for data-parallel programs
Conclusions

- Thread subsumption works best for D&C and nested data parallelism, much superfluous parallelism is pruned
  ⇒ match thread creation policy and parallelism pattern
Conclusions

- Thread subsumption works best for D&C and nested data parallelism, much superfluous parallelism is pruned
  ⇒ match thread creation policy and parallelism pattern
- GC overhead for SMP due to contention
  ⇒ throttle aggressiveness of work stealing and thread creation
Conclusions

- Thread subsumption works best for D&C and nested data parallelism, much superfluous parallelism is pruned
  ⇒ match thread creation policy and parallelism pattern
- GC overhead for SMP due to contention
  ⇒ throttle aggressiveness of work stealing and thread creation
- Communication overhead due to fragmentation for GUM
  ⇒ co-locate sparks from the same spark site
Conclusions

- Thread subsumption works best for D&C and nested data parallelism, much superfluous parallelism is pruned
  ⇒ match thread creation policy and parallelism pattern
- GC overhead for SMP due to contention
  ⇒ *throttle aggressiveness of work stealing and thread creation*
- Communication overhead due to fragmentation for GUM
  ⇒ *co-locate sparks from the same spark site*
- Distributed-memory RTS design scales better than shared-memory design
Conclusions

- Thread subsumption works best for D&C and nested data parallelism, much superfluous parallelism is pruned ⇒ match thread creation policy and parallelism pattern
- GC overhead for SMP due to contention ⇒ throttle aggressiveness of work stealing and thread creation
- Communication overhead due to fragmentation for GUM ⇒ co-locate sparks from the same spark site
- Distributed-memory RTS design scales better than shared-memory design
- Next step: use additional architectural and system-level information to tune adaptive parallelism management