Towards a Characterisation of Parallel Functional Applications

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

Arbeitstagung Programmiersprachen (ATPS’15), Dresden, Germany

March 18, 2015

\textsuperscript{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 are promising in balancing productivity and performance across diverse architectures

Application Characterisation

- Increases understanding of parallel program behaviour
- 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

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

```
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)
```
Applications

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)

\[\text{mostly adopted from the nofib suite and Seq no more paper}\]
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</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-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 and Scalability
- Granularity (Thread size)
- Memory Use (Heap residency, allocation rate, GC%)
- Communication and Sharing
  (Global Address (GA) table residency, messages per sec)
Run Times

- parfib
- queens
- coins
- suneuler (interval-based)
- worpitzky
- maze
- minimax
- mandelbrot

PEs 1 2 4 8 16 32 48 64

Execution time (sec)

GUM on beowulf  GUM on cantor  SMP on cantor
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; \texttt{thread subsumption})

\textsuperscript{5} also depends on the application-level thresholding/chunking
Selected Granularity Profiles on 48PEs
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)

Divide and Conquer Applications

Data-Parallel Applications

median GC% of elapsed

number of PEs
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 descreasing 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)
Global Address Table Residency (Fragmentation)
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 ⇒ match thread creation policy and parallelism pattern
Conclusions

- Thread subsumption works best for D&C and nested data parallelism ⇒ 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
  ⇒ 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 ⇒ 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 (private heaps) RTS design scales better than shared-memory design
Conclusions

- Thread subsumption works best for D&C and nested data parallelism ⇒ 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 (private heaps) RTS design scales better than shared-memory design
- Next step: use additional architectural and system-level information to tune adaptive parallelism management
Speedups

E. Belikov (Heriot-Watt University)  Characterising GpH Applications  March 18, 2015

Graphs showing speedups for various applications:
- **partfib**
- **coins**
- **srumeluer**
- **worpitzky**
- **maze**
- **minimax**
- **mandelbrot**

The graphs compare performance across different numbers of processing elements (PEs): 1, 2, 4, 8, 16, 32, 48, and 64. The x-axis represents the number of PEs, while the y-axis shows the speedup. The graphs depict performance for GUM on Beowulf, GUM on Cantor, and SMP on Cantor.
Granularity Profiles on 48PEs

[Graphs showing thread granularity profiles for different applications: queens, sumEuler, maze, mandelbrot. Each graph compares the number of threads against thread granularity (ms) on two different systems (beowulf and cantor).]
Memory Use: Allocation Rate (on PE1, GB / MUT sec)
Communication Rate

Divide and Conquer Applications

Data-Parallel Applications