Characterisation of Parallel Functional Applications

Student Research Paper

Evgenij Belikov, Hans-Wolfgang Loidl, and Greg Michaelson

School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, EH14 4AS, Edinburgh, UK
{eb120|H.W.Loidl|G.Michaelson}@hw.ac.uk

Abstract. This paper presents a profiling-based characterisation of seven small and medium-sized semi-explicitly parallel functional divide-and-conquer and data parallel applications with respect to thread granularity, communication, and memory management profile, which appear highly relevant for dynamic and adaptive parallelism control at run-time system level. The results confirm that the parallel Haskell implementations cope well with large numbers of potential threads, quantify the impact of the communication rate on performance, and identify memory management overhead as a major limiting factor to scalability on shared-memory machines. We find that a message-passing-based implementation outperforms a shared-memory-based implementation in terms of run time and scalability up to a factor of 6.3. This characterisation helps to improve our understanding of the behaviour of parallel functional programs and discover program attributes that can be exploited to optimise application performance by improving parallelism management policies. In particular, threads subsumption mechanism appears key to controlling thread granularity and too aggressive thread creation results in higher garbage collection overheads thus limiting scalability.

1 Introduction

Currently, parallelism is a key source of gains in application performance as mainstream architectures feature a growing number of processing elements (PEs) since single-core performance stopped increasing due to physical limitations. However, exploiting parallelism remains a major challenge, exacerbated by the diversity of architectures that rapidly evolve towards heterogeneous and hierarchical designs that use processors with different instruction sets, functionalities, and power-performance characteristics, alongside non-uniform memory access (NUMA) and interconnect [2, 5, 25]. A key issue for effective parallel programming is the increased complexity of specifying parallelism management and coordination in addition to a correct and efficient algorithm to solve a particular problem in a given domain. Hence, a desirable solution should achieve performance portability across a wide range of architectures without sacrificing programmer productivity [28, 29].
Functional languages provide a high level of abstraction thus enhancing productivity and are well-suited to exploit parallelism due to purity and referential transparency, as evaluation order is not fixed and expressions can be evaluated in parallel [33, 15]. Moreover, higher-order functions increase modularity and allow to naturally express composable patterns that exploit parallelism whilst separating computation from coordination [17, 7, 32, 11]. Productivity can be further increased, by transferring the responsibility for communication, synchronisation, as well as thread and memory management to the compiler and the run-time system (RTS). Notably, a deterministic programming model can be supported that by design rules out the occurrence of common errors that are notoriously difficult to detect and correct such as deadlocks and race conditions [18]. Further advantages include sequential debugging and iterative parallelisation, since parallel and sequential programs are guaranteed to deliver identical results [12].

Unfortunately, despite the relative ease of parallelisation, tuning parallel functional applications across diverse architectures is challenging, since operational cost model is often non-intuitive because low-level coordination decisions are taken implicitly by the RTS [19]. For instance, exploiting all the fine-grained parallelism inherent in functional programs would result in prohibitive thread management overhead, whilst too coarse granularity leads to load imbalance, thus reducing performance. Consequently, manual tuning appears infeasible due to rapid architectural evolution in general, and in particular for input-dependent and irregular applications common in large-scale data analysis, computational finance, and symbolic computation domains. Hence, performance portability has to be obtained automatically, whilst programmers at best would provide non-mandatory optimisation hints that the RTS could ignore.

This paper presents a profiling-based characterisation of seven small and medium-sized parallel functional applications from several domains with respect to characteristics relevant for dynamic and adaptive management of parallelism, such as the available and exploited parallelism, scalability, communication, granularity, and memory usage on a modern 48-core machine and we are currently consolidating results from experiments run on a Beowulf cluster of multi-cores.

The results confirm that the parallel Haskell implementation copes well with large numbers of potential threads, quantify the impact of the communication rate on parallel performance, and identify memory management overhead as a major limiting factor to scalability on shared-memory machines. This characterisation helps understanding the behaviour of parallel functional programs and discover program attributes that can be exploited to improve the effectiveness and efficiency of parallelism management.

2 Background

This section describes parallel Haskell dialects and two run-time systems that implement Glasgow parallel Haskell (GpH) with a focus on parallelism management. A detailed survey of high-level parallel programming models and languages, including imperative and object-oriented, can be found elsewhere [3].
2.1 Parallel Functional Programming in Haskell

We briefly introduce parallel and distributed dialects of Haskell [16] that are not restricted to arrays and group them by programming model and by the level of explicitness of coordination to be managed by the programmer. There have been attempts to auto-parallelise Haskell reporting limited success [14], hence we focus on explicit and semi-explicit programming models.

Shared-Memory Programming Models  
Concurrent Haskell [27] is a Haskell extension to support explicit shared-memory concurrency via a primitive `forkIO` that mandatorily spawns a light-weight thread to evaluate its argument, a potentially side-effecting action, but also makes the programming model non-deterministic. Synchronisation is achieved by using mutable reference cells (`MVar`s).

The `Par` monad provides a deterministic shared-memory programming model with explicit thread creation using `fork` and explicit granularity control [23]. It targets large-grain irregular applications and is influenced by the dataflow model. The API is similar to Concurrent Haskell but uses `IVars` for communication between threads and lifts scheduling to Haskell level.

Software Transactional Memory (STM) [13] aims to alleviate the issues associated with explicit shared-memory programming by providing transactional variables (`TVar`s) and composable constructs to atomically execute a block of expressions. If no conflicts occur, the transaction is committed, whilst otherwise execution is rolled back and transaction needs to be retried. This way all threads maintain a consistent view of memory. The implementation requires changes to the RTS and uses lock-free data structures instead of locks.

Distributed-Memory Programming Models  
Cloud Haskell (CH) [9] is an embedded domain-specific language (EDSL) for distributed programming inspired by Erlang. CH provides constructs for explicit communication, closure serialisation, and (light-weight) process creation and placement lifted to the Haskell level and supports fault tolerance.

Haskell distributed parallel Haskell (HdpH) [21] is a monadic EDSL for distributed parallel programming and is based on CH but supports fault tolerance, polymorphic closures, and dynamic load balancing using random work stealing. On a node `fork` is used to create threads which communicate using single-assignment reference cells (`IVars`). Analogously, across the nodes, global `GIVars` are used for synchronisation, whilst tasks are created using `spark` with explicit placement using `pushTo`.

Eden [20] is a Haskell extension for shared-nothing distributed parallel programming and provides constructs for explicit process creation and static placement using work pushing. Granularity is controlled by the programmer, whereas communication is implicit and relies on a custom RTS that supports different communication libraries such as Parallel Virtual Machine (PVM) and Message Passing Interface (MPI), as well as a custom zero-copy implementation for shared-memory machines. A set of algorithmic skeletons is provided to raise the level of abstraction (e.g. parallel map, static farm, dynamic workpool).
Glasgow parallel Haskell (GpH) [30] extends Haskell with two language constructs. The `par :: a -> b -> b` combinator annotates an expression that could be profitable to evaluate in parallel and `pseq :: a -> b -> b` fixes the evaluation order by evaluating the first argument and then the second. Using these constructs directly proved difficult since it requires operational knowledge otherwise parallelism may be lost due to unexpected strictness or laziness. Evaluation Strategies [32, 22] have been introduced to address this issue and provide lazy polymorphic higher-order functions that separate computation from coordination and implement common evaluation patterns similar to the level of abstraction of Eden skeletons.

MetaPar [10] is a continuation-based meta-scheduler for the Par monad that aims at facilitating scheduling and load balancing for heterogeneous architectures including GPUs and Clouds based on a Resource abstraction. Inspired by composable scheduler activations, MetaPar bases its functionality on Accelerate [6] for GPU execution and on CH ideas for distributed execution.

2.2 Run-Time System Support for Glasgow parallel Haskell

We focus on semi-explicit GpH as it provides a unified high-level programming model that supports both distributed-memory as well as shared-memory architectures and comes close to implicit parallelism yet has larger potential for annotation-based optimisations. GpH relies on a RTS for policy control.

Run-Time System Policies  
Thread Management and Granularity Control constitute two key policies. At every encountered `par`, a spark is created and added to the spark pool. Once a PE is idle it will turn a local spark into a light-weight thread to execute. Sparking is cheap since a spark is a pointer to an unevaluated part of the graph (a thunk), whereas light-weight threads are more expensive to manage as they store execution state and a stack. A thread can subsume a potentially parallel thread by evaluating the spark it has created sequentially to avoid thread creation costs. This mechanism, akin to lazy futures, helps throttle too fine-grained parallelism. In contrast to the fully implicit approach, programmer annotations may help the RTS to further increase granularity by creating less sparks. If no local spark is available, the PE attempts to steal work from other PEs.

Scheduling and Load Balancing are controlled by the RTS to ensure that work is evenly distributed across the PEs. Random work stealing is used for load balancing but there are differences in implementations, as discussed below. The key decisions are which PE to ask for work and which spark to donate. In general it is impossible to predict the amount of work associated with a spark which makes the decision difficult. For regular divide and conquer applications the breadth-first until saturation then depth-first heuristic works well and matches the intuition that older sparks are larger, so a PE would tend to send older sparks away and keep younger ones for local subsumption. Work stealing is scalable due to its decentralised nature and costs of stealing are amortised since idle PEs initiate the work search.
Glasgow Parallel Haskell Implementations

The two RTS variants, which implement GpH, share large portions of the RTS available with the Glasgow Haskell Compiler (GHC), which implements parallel graph reduction, and are geared towards adaptive dynamic management of parallelism.

\textit{GHC-SMP} [24] is designed for use on shared-memory architectures and allows idle PEs to access and steal from other PEs’ spark pools (implemented using lock free deques), which is efficient but also a rather aggressive mechanism since it leaves no flexibility for those PEs to retain sparks for themselves. On each core, the RTS runs a Capability, which is an OS thread that holds all the information necessary to evaluate closures (or thunks). A number of light-weight threads (or tasks) created from sparks are multiplexed across the Capabilities. Work pushing to idle PEs is also supported.

\textit{GHC-GUM} (Graph Reduction for a Unified Machine Model) [31], by contrast, uses a \textit{virtual shared graph} and runs a single-threaded instance of the RTS on each core. Thus communication is required to distribute work as spark pools are private. If a thread attempts to evaluate a thunk already under evaluation, it will block and will be later notified once the value has become available, hence avoiding work duplication. Otherwise a thread will evaluate the thunk by rewriting the shared structure with a so-called black hole on which other threads would block and sends fetch-me messages to awake the blocking threads once the result has been computed. GUM’s scheduling mechanism is designed for flat and homogeneous networks of PEs and is less aggressive by favouring sending less but larger messages and attempting to overlap communication and computation where possible to hide latency.

2.3 Application Characterisation Studies

Most application characterisation studies are used to assess and inform the design of computer architectures. Another common use case is the comparison of the coverage of parameter space by several benchmark suites to assess their similarity. We aim to discover architecture and application specific parameters that could be monitored and dynamically tuned within an RTS.

The influential Berkeley Report [2] introduces twelve \textit{motifs} that describe common computational kernels (e.g. graph algorithms, structured grid, dense and sparse matrix operations) and reviews their use in different application domains to justify their importance. However, this view appears rather high-level as multiple motifs can overlap in terms of run-time behaviour and parallelism patterns used.

Based on characterisations of SPLASH-2 and PARSEC benchmarks [4] we decided to use several common characteristics such as the working set size (heap residency), communication-to-computation ratio, numbers of threads, but we also use system-specific information: memory allocation rate, detailed information on granularity of the light-weight threads and on related blocking and fetching times and counts, as well as size of the global address table residency and the amount of graph sent to assess sharing and locality.
3 Parallel Applications

We describe the applications used grouped by the exploited parallelism pattern. Most of the applications have been adopted from the parallel part of the nofib suite [26] and from a recent study of Evaluation Strategies [22]. Applications using simple yet powerful patterns are deemed representative of large class of task and data parallel applications [8]. We investigate how program behaviour varies across the patterns and whether a pattern could be detected from characteristics observable by the RTS which may prove useful for dynamic optimisation.

3.1 Divide and Conquer

Five of the applications use the divide and conquer (D&C) pattern, where a problem is recursively split into sub-problems that are solved and the results combined to form the final result. A threshold value can be used to restrict the depth of a tree to a certain level from which on the problem is solved sequentially.

– parfib is a D&C program that computes the number of function calls for the recursive computation of the Nth Fibonacci number which adds (N-1)th and (N-2)th numbers using arbitrary-length integers; we use \( N = 50 \) and threshold of 23; this benchmark aims at assessing thread subsumption capabilities of the RTS and is representative of a regular and flat D&C applications with a single source of parallelism; both the splitting and the combining phases require two arithmetic operations on integers of arbitrary length, the sequential work is exponential; the number of packets sent for the GUM version increases linearly whilst average packet size remains constant

– worpitzky checks the Worpitzky identity for two given arbitrary-length integers and is representative of the domain of symbolic computations; we take 19 to the exponent of 27 and use a threshold of 10; at the top level this requires one exponentiation, one equality comparison, and a fold (sum) over a list of \( n \) intermediate results, which are computed in part in parallel and for the other part require two arithmetic operations and binomial computation using three factorial and three arithmetic operations; parallel computations include a single source of parallelism and 3 arithmetic operation for both the combine and the split phase; the number of packets sent increases linearly and is substantially larger than for parfib and coins with a roughly constant packet size; the activity profile shows periodic phases of low activity and significant number of fetching threads

– queens determines the number of solutions for placements of \( N \) queens on a square board of \( N \times N \) so that no two queens are attacking each other, with \( N = 16 \); the positions are represented by a list of integers and generated by discarding unsafe positions; in contrast to all other applications queens sends an excessive amount of packets although there is no clear trend in the median number of packets sent per PE; although decreasing, average packet size is larger than packet buffer and leads to additional packets being sent in fragments, thus leaving some room for optimisation
- *coins* computes possible ways to pay out a specified amount from a given set of coins; in our case the value is 5777; the application is similar to *parfib* with linearly growing number of messages of slightly decreasing size; both the split and the combine phases require one arithmetic operation, whilst sequential solution requires finding suitable permutations of coins

- *minimax* calculates winning positions for a noughts-vs-crosses game on a NxN board up to a specified depth using alpha-beta search and exploits lazyness to prune unpromising sub-trees and *parList* strategy to introduce parallelism; we use $N = 4$ and a depth of 8; the board is represented by a list of rows of cells containing either *Empty*, *X* or *O*; the number of packets sent increases linearly and is similar to *parfib*, whilst packet size is constant and second largest although two orders of magnitude smaller than for *queens*

### 3.2 Data Parallelism

Two of the applications are *data parallel*, i.e. the parallelism is exploited by simultaneously applying a function to the elements of a data structure. Although GpH is not the best-suited programming model for regular arrays, it is worth considering for nested and irregular data parallelism on lazy data structures.

- *sumeuler* computes the sum over euler totient numbers in a given integer interval, uses *chunking* for granularity control and is moderately irregular; we use interval from 0 to 100000 with a chunk size of 500; parallelism is generated in the beginning of the execution which explains the roughly constant number of packets sent and decreasing median number of messages sent per PE, whilst average packet size remains constant and very small

- *maze* is a nested data parallel AI search application that finds a path in a maze (of size 29); communication rate is relatively low, whilst average packet size remains constant; the overall number of available sparks depends on the number of PEs as some speculative parallelism is used

### 4 Application Characterisation

This section presents experimental design, and evaluates obtained results to characterise the dynamic behaviour of the aforementioned applications. The key characteristics are execution time, speedup, available and actual degree of parallelism, granularity profiles, working set sizes, allocation rates, percentage of garbage collection (GC) and communication rate.

#### 4.1 Experimental Design

We report application performance and profiles on a multi-core and are currently consolidating results obtained on a cluster of multi-cores. The data presented is taken from a median run out of three. We report relative speedup as we are primarily interested in the behaviour of the parallel applications.
The 48-core machine consists of four AMD Opteron processors with two
NUMA nodes with six 2.8GHz cores each. Every two cores share 2MB L2 cache
and all six cores on a NUMA-node share 6MB L3 cache and 64GB RAM (a total
of 512GB). The Beowulf cluster comprises a mix of 8-core Xeon 5504 nodes
with two sockets with four 2GHz cores each, using 256 KB L2 cache, and 4MB
shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets
with four 3GHz cores each, using 6MB shared L2 cache and 16GB RAM. The
machines are connected via a commodity Gigabit Ethernet. On all the machines
we run CentOS 6.5 and use run-time systems based on GHC 6.12.3, gcc 4.4.7,
and PVM 3.4.6. We use the somewhat dated GHC version since we have not
yet ported the GUM RTS and profiling support to a newer version, however we
did run a set of experiments using GHC 7.6 which showed improved scaling for
SMP but the overall trends remained unchanged.

Fig. 1. Overview of Application Runtimes

4.2 Scalability

For each application we fix the input size and increase the number of PEs to
assess the scalability of the programs. Figure 1 presents the execution times on
up to 48 cores. We observe that for most applications the run time decreases
monotonically, that is the applications are able to profitably exploit some par-
allelism. The exceptions are GHC-SMP\(^1\) versions of maze, coins, parfib and
to lesser extent worpitzky since overhead at some point negates the benefits of
adding further cores. We will discuss the reasons in more detail below.

\(^1\) we use SMP and GUM as a shorthand for GHC-SMP and GHC-GUM, respectively
We observe poor performance for queens with no scaling beyond 16 cores on which SMP beats GUM with 6.16 against 4.72 speedup. The run time of minimax decreases for GUM (except for an anomaly on 4 cores due to external load interference), however, best speedup is rather low with 7.46 on 48 cores, where SMP has 0.93 (best on 16 cores with 4.88). The performance of worpitzky also leaves room for improvement: best GUM speedup is on 32 cores with 9.10 (9.01 on 48). For SMP speedup is best on 16 cores with 4.42 and 0.94 on 48 cores. Notably, there is a very large overall number of threads for both GUM (20k) and SMP (320k) on 48 cores. The run time of the GUM version of parfib is reduced in all cases: on 48 cores run time is 37.25 sec, whilst SMP is 18x slower with 688.98 sec. GUM version scales linearly up to 16 cores, and best speedup is 26.42 on 48 cores, whereas SMP version reaches best speedup of 6.12 on 16 cores, whilst speedup on 48 cores is merely 1.22. We observe a similar picture for coins: GUM run times always decrease to 73.77 sec on 48 cores vs SMP’s 4181 sec. The best runtime is achieved on 8 cores corresponding to the runtime of GUM on 4, around 400 sec. The GUM version shows the best speedup out of the programs used: 28.22 on 48 cores, whilst SMP version scales poorly with best speedup of 3.27 on 8 cores and slowdown to 0.36 on 48 cores.

As shown in Figure 2, most applications scale well up to 8 cores, but then most are overwhelmed by the overheads and only GUM versions of parfib and coins continue scaling up to 48 cores whilst maintaining efficiency of over 50%. Super-linear speedup is due to parallel overhead when run on one PE and it is not exceeding 10% thus resulting in slightly sub-linear absolute speedup. We discuss working sets in Section 4.4. We argue that parfib and coins start running out of work as allocation rate substantially decreases.

**Fig. 2.** Overview of Application Scalability
Table 1. Parallelism Degree: 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>suomeuler-GUM</td>
<td>135</td>
<td>171</td>
</tr>
<tr>
<td>suomeuler-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>

In general, parallelism is abundant and fine-grained leaving considerable parallel slackness and requiring thread subsumption. For instance, for worpitzky there are 7340000 sparks, most of which are pruned at run time. GUM appears well-suited for D&C applications and is able to subsume threads to a larger extent than SMP. On the other hand, if potential parallelism is low, then SMP manages to have less threads overall.

4.3 Granularity

We have extended RTS profiling capabilities to record thread granularity information, blocking time, fetching time and the number of times a thread blocks and fetches. Profiling overhead is negligible as it merely involves counters, the overhead increases with the number of threads but so do other more significant overheads such as communication and garbage collection. In contrast to GHC, GUM maintains private heaps between RTS instances and thus avoids locking.

In Figure 3 we present the granularity profiles (note the different scales). GUM version of parfib has an order of magnitude less threads, and the threads are up to two orders of magnitude larger, which indicates the effectiveness of thread subsumption. The reason why GUM version of coins has less but smaller threads is due to larger absolute run times of the SMP version. For minimax, SMP has less and larger threads and yet performance is worse than GUM, an issue we discuss in Section 4.4. For worpitzky GUM has an order of magnitude (32x) less and larger threads, alas thread numbers are prohibitively large for both versions. For queens SMP has smaller number of larger threads and SMP outperforms GUM, although overall performance is poor in both cases.
We observe distinctive shape similarity for SMP versions of parfib, coins, and to lesser extent worpitzky, which suggests that it may be possible to detect the use of the D&C pattern at run time. For GUM, shape similarity is apparent for parfib and worpitzky, whilst coins appears rather different.
4.4 Memory Use and Garbage Collection

Many parallel functional programs are memory-bound as they perform graph reduction. We measure heap residency to represent a program's working set, observe allocation rate as a characteristic representing memory access, and we record the median percent of the elapsed time used for garbage collection (GC).

For `parfib` median GC% for GUM decreases from 13% on one core to 6.75% on 48 cores, whilst for SMP the GC% steadily increases from 1.6% on one core to 61% on 48 cores. Heap residency is small and drops for GUM, small but increases by factor of 10 for SMP. Allocation rate is constant for GUM on up to two 16 cores then roughly halved; for SMP it grows first but then decreases to values lower than for GUM.

<table>
<thead>
<tr>
<th>PEs</th>
<th>parfib</th>
<th>worpitzky</th>
<th>queens</th>
<th>coins</th>
<th>minimax</th>
<th>sumeuler</th>
<th>maze</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39</td>
<td>32</td>
<td>39</td>
<td>33</td>
<td>296</td>
<td>275</td>
<td>70</td>
</tr>
<tr>
<td>2</td>
<td>24</td>
<td>53</td>
<td>24</td>
<td>38</td>
<td>1601</td>
<td>1608</td>
<td>62</td>
</tr>
<tr>
<td>4</td>
<td>23</td>
<td>75</td>
<td>18</td>
<td>60</td>
<td>922</td>
<td>2308</td>
<td>71</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
<td>106</td>
<td>24</td>
<td>125</td>
<td>402</td>
<td>2740</td>
<td>66</td>
</tr>
<tr>
<td>(median)</td>
<td>16</td>
<td>23</td>
<td>170</td>
<td>27</td>
<td>232</td>
<td>328</td>
<td>2978</td>
</tr>
<tr>
<td>32</td>
<td>19</td>
<td>245</td>
<td>29</td>
<td>326</td>
<td>133</td>
<td>2982</td>
<td>58</td>
</tr>
<tr>
<td>48</td>
<td>21</td>
<td>349</td>
<td>29</td>
<td>433</td>
<td>92</td>
<td>2987</td>
<td>58</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>unit</th>
<th>KB</th>
<th>KB</th>
<th>KB</th>
<th>MB</th>
<th>MB</th>
<th>KB</th>
<th>KB</th>
<th>MB</th>
<th>KB</th>
<th>KB</th>
<th>KB</th>
<th>KB</th>
<th>KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.5</td>
<td>1.6</td>
<td>1.8</td>
<td>1.8</td>
<td>0.23</td>
<td>0.21</td>
<td>4.8</td>
<td>4.8</td>
<td>2.3</td>
<td>2.3</td>
<td>0.8</td>
<td>0.6</td>
<td>4.1</td>
</tr>
<tr>
<td>2</td>
<td>1.3</td>
<td>1.6</td>
<td>1.8</td>
<td>2.0</td>
<td>0.23</td>
<td>0.22</td>
<td>4.7</td>
<td>5.7</td>
<td>2.3</td>
<td>2.6</td>
<td>0.8</td>
<td>0.6</td>
<td>4.1</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
<td>1.8</td>
<td>1.3</td>
<td>2.7</td>
<td>0.22</td>
<td>0.23</td>
<td>4.7</td>
<td>7.0</td>
<td>2.3</td>
<td>3.2</td>
<td>0.7</td>
<td>0.7</td>
<td>4.1</td>
</tr>
<tr>
<td>8</td>
<td>1.5</td>
<td>1.5</td>
<td>1.1</td>
<td>2.3</td>
<td>0.19</td>
<td>0.21</td>
<td>4.8</td>
<td>5.6</td>
<td>1.5</td>
<td>2.6</td>
<td>0.7</td>
<td>0.7</td>
<td>4.1</td>
</tr>
<tr>
<td>(PE1)</td>
<td>16</td>
<td>1.3</td>
<td>1.4</td>
<td>1.0</td>
<td>1.5</td>
<td>0.16</td>
<td>0.16</td>
<td>4.7</td>
<td>3.3</td>
<td>2.0</td>
<td>2.0</td>
<td>0.7</td>
<td>0.6</td>
</tr>
<tr>
<td>GB/</td>
<td>32</td>
<td>0.9</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
<td>0.11</td>
<td>0.07</td>
<td>2.8</td>
<td>0.6</td>
<td>1.3</td>
<td>0.8</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>MUTsec</td>
<td>48</td>
<td>0.9</td>
<td>0.3</td>
<td>0.6</td>
<td>0.3</td>
<td>0.08</td>
<td>0.03</td>
<td>2.7</td>
<td>0.2</td>
<td>1.0</td>
<td>0.3</td>
<td>0.6</td>
<td>0.3</td>
</tr>
</tbody>
</table>

| 1 | 13.0| 1.6| 15.4| 2.2| 13.8| 1.9| 33.4| 8.0| 22.8| 27.0| 7.5| 0.8| 31.1| 34.0 |
| 2 | 8.1 | 3.9| 12.5| 6.7| 13.3| 4.8| 28.7| 14.2| 22.0| 33.9| 6.3| 2.2| 31.1| 39.9 |
| 4 | 7.5 | 7.7| 8.8 | 13.4| 10.8| 14.2| 25.0| 20.9| 52.2| 36.5| 3.6| 4.0| 31.2| 44.6 |
| (median) | 8 | 7.6| 16.2| 9.1| 27.5| 7.8| 26.1| 24.4| 34.5| 59.0| 41.6| 3.2| 8.2| 31.8| 50.2 |
| 16 | 7.4 | 26.7| 9.1| 41.0| 5.3| 45.8| 24.3| 45.0| 57.5| 53.5| 2.8| 15.1| 32.1| 49.6 |
| 32 | 6.4 | 36.4| 8.6| 47.0| 3.1| 47.0| 22.8| 47.2| 52.5| 66.4| 2.5| 22.5| 30.9| -   |
| 48 | 6.8 | 61.0| 6.9| 48.2| 1.8| 46.1| 22.7| 46.1| 53.4| 62.4| 2.0| 32.1| 29.5| -   |

GUM version of `coins` exhibits GC drop from 33.4% on one core to 22.7% on 48, whilst SMP GC increases from 0.8% to 46.1%. Residency is roughly constant for GUM but grows fo SMP. Allocation rate is roughly constant for GUM up to 16 cores and then is halved. These observations indicate that regular and moderately irregular D&C workloads scale well on GUM.
The GC percentage for **minimax** is large for either RTS — on 48 cores GUM shows 53.4% and SMP 62.4%. GUM has larger residency: 241MB vs 35MB for SMP on 48 cores, which might be linked to communication overheads resulting from graph sharing. The allocation rate for SMP slightly increases but then drops faster than for GUM.

We see a similar picture for **worpitzky** — GC drops from 15.4% to 6.9% for GUM and it increases from 2.2% to 48.2% for SMP. GUM’s residency is constant, whilst the allocation rate drops. By contrast, SMP memory residency increases from 33 on one to 433 on 48 cores along with first increasing and then rapidly dropping allocation rate. For **sumeuler** GUM residency drops almost instantly signifying some load imbalance due to insufficient parallelism indicated by low relatively allocation. The trend regarding GC% is visible for **sumeuler** and **maze**. GC% decreases for GUM and increases for SMP consistently across all applications, suggesting a memory management bottleneck on SMP that is responsible for limited scalability on higher numbers of PEs.

![Communication-to-Computation Ratio on a Multi-Core](image)

**Fig. 4.** Communication-to-Computation Ratio on a Multi-Core

### 4.5 Communication

GUM-specific communication characteristics, such as communication-to-computation ratio, amount of sharing by observing the amount of graph sent, locality and heap fragmentation by measuring the residency of global address (GA) tables, and percentage of steal requests (FISH messages) as indicator of load imbalance or lack of work, provide additional insight into operational behaviour of parallel functional programs. Surprisingly, in most cases GUM outperforms SMP on a multi-core although designed for a distributed-memory architecture.
As shown in Figure 4, for parfib, coins, maze, and to lesser extent minimax and sumeuler we observe modest linear increase in communication rate with less than 40% of FISH messages. The median of the graph sent and GA residency usually increase slightly, but only for queens they are excessive with over 19k GA residency and around 14MB median graph sent per mutation second on 48 cores (even more on smaller number of cores!), similarly the communication rate for queens skyrockets (840k messages on 48 cores with very long fetch times and counts, but only 15% of messages are FISHes). We are currently investigating ways to eliminate these overheads. Next highest communication rate is for worpitzky with over 100k messages sent on 48 cores, almost 50% of which as requests for work, due to very fine thread granularity. Following suit is sumeuler, which exemplifies another issue — lack of inherent parallelism for the given threshold leads to load imbalance on higher number of PEs demonstrated by over 95% of sent messages being requests for work. By contrast, the behaviour of minimax is unsuspicious: communication rate grows linearly, the percentage of FISHes in the total messages are mostly under 50%, whilst median graph sent is roughly constant accompanied by a slight increase of GA residency.

Having assessed the scaling of the applications, we observe that careful parallelisation is required to avoid pitfalls that result in excessive overhead and poor performance. Semi-explicit programs have the potential to scale, provided there is enough work on the one hand, but that the granularity is adequately controlled, on the other. Our profiling results indicate that it may be possible to detect an application of a pattern such as D&C and exploit this knowledge for optimisation, or it may be possible to detect a pathological case (e.g. high percentage of FISH messages or increasing fetch counts). We are currently working on ways to adapt to such situations.

5 Conclusions

We have presented a characterisation of a set of small and medium-sized parallel functional programs in terms of communication rate, heap residency, allocation rate, and thread granularity. Our findings also demonstrate how profiling can reveal diverse bottlenecks and help gain insight into dynamic application behaviour. The results indicate that both GHC-GUM and GHC-SMP can handle a large number of light-weight threads, and in the case of divide-and-conquer parallelism, effectively merge potential parallel computations into a single thread, thus reducing overhead. However, SMP’s mechanism is more aggressive and leads to a larger number of threads.

One challenge in modern NUMA architectures is to deal with different memory latencies based on the NUMA region in which a memory bank is located. In our characterisation, we find that allocation rate and GC overhead tend to be significantly lower in GUM, in particular for compute-intensive divide-and-conquer applications — an aspect studied in more detail in the accompanying paper [1]. The virtual shared heap GUM implementation mostly generates fewer and larger threads and outperforms SMP up to a factor of 6.3 for all but one
applications. The communication rate of the applications varies considerably, and has a high impact on the parallel performance.

We are currently extending our results to a cluster of multi-cores, focusing in particular at the communication profile on this platform, and adding further applications to the set of benchmarks. The data from this characterisation will inform the design of a dynamic adaptation mechanism based on monitoring — for instance by detecting a pattern of parallelism and dynamically tuning a related policy or using the rate of sparking and fishing to separate workers from generator of parallelism. We also plan to add an architecture-aware cost-model to the RTS to guide parallelism management at run time.

Acknowledgements

We are grateful to Rob Stewart for helpful comments and to SICSA, the Scottish Informatics and Computer Science Alliance, for support.

References