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 eight small and medium-sized semi-explicitly parallel functional divide-and-conquer and data parallel applications on a server-class multi-core and on a cluster of multi-cores focusing on thread granularity, communication, and memory management profiles, 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. The characterisation improves our understanding of the behaviour of parallel functional programs and hints at attributes that can be exploited to optimise application performance by improving parallelism management policies. In particular, thread subsumption mechanism appears key to controlling thread granularity whilst too aggressive thread creation increases garbage collection overhead. Additionally in the distributed-memory implementation, the amount of sharing determines the size of the global address table and hence the communication overhead associated with fragmentation of the virtual shared heap.

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, 26]. 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 [29, 30].

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 [34, 16]. Moreover, higher-order functions increase modularity and allow to naturally express composable patterns that exploit parallelism whilst *separating computation from coordination* [18, 7, 33, 12]. 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 [19]. Further advantages include sequential debugging and iterative parallelisation, since parallel and sequential programs are guaranteed to deliver identical results [13].

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 [20]. 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 and limiting scalability. 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 advisory optimisation hints that the RTS could ignore.

This paper presents a profiling-based characterisation of eight 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 a Beowulf-class cluster consisting of multiple 8-core nodes (we use up to 64 cores in total).

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. Surprisingly, a distributed-memory implementation outperforms a shared-memory implementation on a shared-memory architecture for most applications despite the added communication overhead.
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. High-level parallel programming models and languages are discussed in detail in a recent survey [3], whilst lower-level models are reviewed in [9].

2.1 Parallel Functional Programming in Haskell

We briefly introduce parallel and distributed dialects of Haskell [17] 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 [15], hence we focus on explicit and semi-explicit programming models.

Shared-Memory Programming Models

Concurrent Haskell [28] is a Haskell extension to support explicit shared-memory concurrency via a primitive \texttt{forkIO} that mandatorily spawns a light-weight thread to evaluate its argument, a potentially side-effecting action, but also makes the programming model \textit{non-deterministic}. Mutable reference cells (\texttt{MVar}s) are used for synchronisation.

The \texttt{Par} monad provides a \textit{deterministic} shared-memory programming model with explicit thread creation and granularity control using the \texttt{fork} function [24]. It targets applications with coarse-grained irregular parallelism and is influenced by the dataflow model. The API is similar to Concurrent Haskell but uses single-assignment reference cells (\texttt{IVars}) for communication between threads and lifts scheduling to Haskell level.

Software Transactional Memory (STM) [14] aims to alleviate the issues associated with explicit shared-memory programming by providing transactional variables (\texttt{TVar}s) and composable constructs to \textit{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) [10] 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) [22] is a monadic EDSL for distributed parallel programming and is based on CH and supports fault tolerance, polymorphic closures, and dynamic load balancing using random work stealing. On a node \texttt{fork} is used to create threads which communicate using \texttt{IVars}. Analogously, across the nodes, global \texttt{GIVars} are used for synchronisation, whilst tasks are created using \texttt{spark} with explicit placement using \texttt{pushTo}.
**Eden** [21] 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) [31] extends Haskell with two language constructs. The \texttt{par} :: \texttt{a -> b -> b} combinator annotates an expression that could be profitable to evaluate in parallel and \texttt{pseq} :: \texttt{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** [33, 23] have been introduced to address this issue and provide lazy polymorphic higher-order functions that separate computation from coordination and implement common evaluation patterns at a level of abstraction similar to the Eden skeletons.

**MetaPar** [11] 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 \texttt{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 closure in the shared 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 task creation, 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 sparks are available, idle PEs attempt 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 that implements parallel graph reduction and is available with the state-of-the-art Glasgow Haskell Compiler (GHC), and are geared towards adaptive dynamic management of parallelism.

*GHC-SMP* [25] 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 so-called 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.

*GHC-GUM* (Graph Reduction for a Unified Machine Model) [32], by contrast, uses a 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 until notified once the result has become available. 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 *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 the characterisation 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 and the amount of graph sent to assess sharing and locality.

3 Parallel Applications

Most of the applications we use have been adopted from the parallel part of the nofib suite [27] and from a recent study of Evaluation Strategies [23]. Applications using simple yet powerful patterns are deemed representative of large class of task and data parallel applications [8]. We group the application by the exploited parallelism pattern and investigate how program behaviour changes across different run-time systems and architectures with varying number of PEs.

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.

− The parfib program computes the number of function calls for the recursive computation of the Nth Fibonacci number 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 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 worpitzky application 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 queens program determines the number of solutions for placements of N queens on a square board of NxN 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.
The *coins* program computes possible ways to pay out a specified amount from a given set of coins; in our case the value is 5777; the program is similar to *parfib* as the split and the combine phases require one arithmetic operation each, whilst sequential solution requires finding suitable permutations of coins.

The *minimax* application calculates winning positions for a noughts-vs-crosses game on a NxN board up to a specified depth using alpha-beta search and exploits laziness 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.

### 3.2 Data Parallelism

Three of the applications are *data parallel*, i.e. the parallelism is exploited by simultaneously applying a function to the elements of a data structure. Explicit *chunking* can be used for advisory granularity tuning at application level.

- The *sumeuler* program computes the sum over euler totient numbers in a given integer interval, uses chunking for granularity control and is fairly irregular; we use interval from 0 to 100000 with a chunk size of 500; all the parallelism is generated in the beginning of the execution.
- The *mandelbrot* application computes the mandelbrot fractal set for a given range and image size as well as number of iterations; we use -2.0 to 2.0 range, 4096x4096 image size, and 3046 iterations; the application is fairly irregular.
- The *maze* program is a nested data parallel AI search application which searches for a path in a maze (of size 29); speculation is used to prune some of the unpromising solution candidates.

### 4 Application Characterisation

This section presents experimental design and evaluates the 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), virtual shared heap fragmentation, and communication rate.

#### 4.1 Experimental Design

We report application performance and profiles from a median run out of three on a multi-core and on a cluster of multi-cores. We report relative speedup as we are primarily interested in the behaviour of the parallel applications.

The 48-core machine (*cantor*) 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). Memory access latency differs by up to a factor of 3 depending on the NUMA regions involved with average latency of 16ns.
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 with average latency of 150ns.

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. Application Execution Times](image)

### 4.2 Performance and Scalability

For each application we fix the input size and increase the number of PEs to assess scalability. Figure 1 presents the run times on up to 48 cores on cantor and up to 64 cores on beowulf (note the different scales). We observe that for most applications the run time decreases as the applications are able to profitably exploit some parallelism resulting in an order of magnitude reduction.
in execution time for 5 programs. The exceptions are *queens* due to excessive memory use, *maze* which generates more work with increasing PE numbers, and GHC-SMP\(^1\) runs on higher numbers of PEs which indicates a scalability issue.

![Fig. 2. Application Scalability](image)

We observe strong scaling for *parfib* and *coins* for GUM with efficiency of over 70% on *beowulf* and over 50% on *cantor* and good scaling for *sumeuler*, which however seems to have load balancing issues for high numbers of PEs. Surprisingly, *maze* doesn’t scale on SMP although it creates work proportional to the number of PEs. SMP shows best performance for low to medium number of PEs, whilst on 48 cores results in a slowdown for 5 programs due to a memory management issue discussed in Section 4.4. By contrast, GUM scales up to 64 PEs in most cases, although often the benefit of adding PEs decreases with PE number due to increasing overhead and reduced work per PE. In particular, *queens*, *mandelbrot*, and to lesser extent *minimax* exhibit limited scalability due to excessive communication and heap residency, which hints at improvement potential at application level. Likewise, we believe that increasing granularity would also improve performance of *worpitzky*, since currently median thread size for this application is very small and the number of threads very high.

\(^1\) we use SMP and GUM as a shorthand for GHC-SMP and GHC-GUM, respectively
We observe a wide range of actual and potential parallelism degrees in Table 1. For instance, *sumeuler* only has 200 sparks which appears insufficient to keep all the PEs busy, whereas for *worpitzky* there are of four orders of magnitude more sparks available, 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 which creates threads more aggressively. Parallelism is often over-abundant and fine-grained in functional programs, leaving considerable parallel slackness and requiring an effective thread subsumption mechanism.

### 4.3 Granularity

We have extended run-time profiling capabilities of GUM and SMP to record thread granularity information as well as time threads spend fetching and blocking. Profiling overhead is negligible as it merely involves counters, and increases with the number of threads but so do other more significant overheads such as communication and garbage collection. In contrast to GHC, GUM RTS instances maintain private heaps and thus avoid GC-related synchronisation overhead.

---

**Table 1. Parallelism Degree: Actual vs Potential**

<table>
<thead>
<tr>
<th>application</th>
<th>number of threads on N cores</th>
<th>total</th>
<th>sparks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td><em>sumeuler</em>-GUM-beowulf</td>
<td>128</td>
<td>165</td>
<td>184</td>
</tr>
<tr>
<td><em>sumeuler</em>-GUM-cantor</td>
<td>135</td>
<td>171</td>
<td>186</td>
</tr>
<tr>
<td><em>sumeuler</em>-SMP-cantor</td>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td><em>minimax</em>-GUM-beowulf</td>
<td>10</td>
<td>30</td>
<td>62</td>
</tr>
<tr>
<td><em>minimax</em>-GUM-cantor</td>
<td>12</td>
<td>161</td>
<td>69</td>
</tr>
<tr>
<td><em>minimax</em>-SMP-cantor</td>
<td>5</td>
<td>31</td>
<td>92</td>
</tr>
<tr>
<td><em>queens</em>-GUM-beowulf</td>
<td>10</td>
<td>66</td>
<td>135</td>
</tr>
<tr>
<td><em>queens</em>-GUM-cantor</td>
<td>14</td>
<td>57</td>
<td>146</td>
</tr>
<tr>
<td><em>queens</em>-SMP-cantor</td>
<td>5</td>
<td>69</td>
<td>153</td>
</tr>
<tr>
<td><em>mandelbrot</em>-GUM-beowulf</td>
<td>763</td>
<td>1259</td>
<td>1772</td>
</tr>
<tr>
<td><em>mandelbrot</em>-GUM-cantor</td>
<td>734</td>
<td>1245</td>
<td>1823</td>
</tr>
<tr>
<td><em>mandelbrot</em>-SMP-cantor</td>
<td>1537</td>
<td>782</td>
<td>7224</td>
</tr>
<tr>
<td><em>parfib</em>-GUM-beowulf</td>
<td>4</td>
<td>34</td>
<td>82</td>
</tr>
<tr>
<td><em>parfib</em>-GUM-cantor</td>
<td>4</td>
<td>38</td>
<td>89</td>
</tr>
<tr>
<td><em>parfib</em>-SMP-cantor</td>
<td>20</td>
<td>136</td>
<td>881</td>
</tr>
<tr>
<td><em>coins</em>-GUM-beowulf</td>
<td>16</td>
<td>62</td>
<td>170</td>
</tr>
<tr>
<td><em>coins</em>-GUM-cantor</td>
<td>13</td>
<td>36</td>
<td>170</td>
</tr>
<tr>
<td><em>coins</em>-SMP-cantor</td>
<td>3</td>
<td>93</td>
<td>3687</td>
</tr>
<tr>
<td><em>worpitzky</em>-GUM-beowulf</td>
<td>57</td>
<td>403</td>
<td>1449</td>
</tr>
<tr>
<td><em>worpitzky</em>-GUM-cantor</td>
<td>55</td>
<td>401</td>
<td>1430</td>
</tr>
<tr>
<td><em>worpitzky</em>-SMP-cantor</td>
<td>162</td>
<td>565</td>
<td>12113</td>
</tr>
<tr>
<td><em>maze</em>-GUM-beowulf</td>
<td>4</td>
<td>13</td>
<td>56</td>
</tr>
<tr>
<td><em>maze</em>-GUM-cantor</td>
<td>4</td>
<td>15</td>
<td>116</td>
</tr>
<tr>
<td><em>maze</em>-SMP-cantor</td>
<td>3</td>
<td>52</td>
<td>1172</td>
</tr>
</tbody>
</table>
Fig. 3. Application Granularity (GUM vs SMP on 48 PEs)

Fig. 4. Application Granularity (GUM vs SMP on 48 PEs) (contd.)
In Figures 3 and 4 we present applications' granularity profiles. Logarithmic scales are used across both dimensions for comparability, because of orders of magnitude differences in thread sizes and numbers. For \textit{parfib}, \textit{coins}, and \textit{worpitzky}, we observe an order of magnitude less and larger threads for GUM than for SMP, which demonstrates the effectiveness of GUM’s thread subsumption mechanism and aggressiveness of SMP’s thread creation for D&C applications. An interesting case is \textit{sumeuler} for which we see the opposite picture as all of the work is created at the beginning of execution and no thread subsumption can take place. Similarly, for \textit{minimax}, \textit{queens}, and \textit{maze}, SMP has less and larger threads. However the performance is rather poor for these programs, due to effects of memory use, sharing, and communication discussed in Sections 4.4 and 4.5. Moreover, \textit{queens} and \textit{mandelbrot} are the only applications where SMP outperforms GUM by a small margin, due differences in communication costs.

In general, the shapes of the profiles for GUM on shared-memory architectures are to a large extent similar to the shape of the profile on a distributed-memory architecture, but remains distinctively different for SMP profiles, suggesting that RTS characteristics have a strong influence on the granularity profile, especially if the architectural features are not explicitly taken into account. For the more scalable applications we observe less and larger threads for GUM than for SMP. In other cases performance is relatively poor and influence of memory residency and communication confound the granularity profile.

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 computational intensity, report the median percentage of elapsed time used for garbage collection (GC) and the number of global references as virtual shared heap fragmentation.

Figure 5 depicts the percentage GC takes and reveals a reason for scalability issues observed with SMP. The GC\% increases consistently across all applications for SMP and results in severe contention on the first generation heap. By contrast, GUM initially starts off with higher GC\% which then reduces or at least remains roughly constant in most cases (with exception of \textit{minimax} where heap residency is very high). This highlights the benefit of a distributed-memory design on shared-memory architectures by avoiding some of the synchronisation, which pays off particularly for applications with low communication rate.

In addition to GC\%, allocation rate signifies computational intensity of each application as shown in Figure 6. We find GUM maintains allocation rate with growing PE numbers for most applications on \textit{beowulf}, whereas on \textit{cantor} allocation rate drops at 32 PEs indicating reduced relative computational intensity due to overhead. After an initial raise in allocation rate on SMP, which confirms the benefit of more aggressive thread creation on lower number of cores on \textit{cantor}, we observe a rapid drop for higher PE numbers, which correlates with a large increase in GC\% pointing to a scalability issue as computational intensity drops far beyond the sequential level.
Fig. 5. Garbage Collection Overhead

Fig. 6. Allocation Rates

Fig. 7. Heap Residency
Application *working sets* are represented by *heap residency* in Figure 7 (note the different units). We observe roughly constant or decreasing residency for GUM on both distributed-memory and shared-memory architectures (except for *minimax*), whilst for SMP the residency is growing in most cases. This corresponds to increased GC%, as due to contention some of the heap-allocated objects are retained for longer, an effect most pronounced for *queens* and *coins*. The jump in residency from one to two PEs for GUM is the result of sharing and the need to maintain global addresses and reflects the potential for optimising *queens* by reducing the amount of sharing.

Moreover, as GUM uses virtual shared memory, each instance of the RTS maintains a Global Address (GA) table of stable inter-processor pointers which are used as roots for garbage collection. Fragmentation of the shared heap can lead to decreased performance since excessive sharing results in higher GA residency and reduced locality, which leads to additional communication overhead. Thus GA residency can be used as an indicator of the degree of virtual shared heap fragmentation. From our application set, *worpitzky* (see Figure 8, increasing with PEs), *mandelbrot* (not shown, residency of 8000 and decreasing), and *queens* (not shown, residency of 250000 and decreasing) appear to suffer most from heap fragmentation, which explains, along with very fine granularity, the limited scaling on larger number of PEs. The GA residency grows for *worpitzky*, since there are two sources of parallelism and the RTS is not able to keep sparks from the same source of parallelism together to improve locality.

### 4.5 Communication

GUM-specific characteristics, such as communication-to-computation ratio, 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 9, 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 increases 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, as communication rate skyrockets (840k messages on 48 cores with frequent very long fetches, 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, which also coincides with decreasing memory residency and low allocation rate. For most applications the number of packets sent increases linearly and reflects the size of shared graph, whilst packet size is mostly very small and constant (in the range between 5 and 50 bytes), except for queens (4k) and mandelbrot (ca. 9k). Due to space limitation we present no graphs on these metrics. We find that in general packets are smallest for integer-based programs and data-parallel programs often have larger packets than D&C programs. Communication rate appears to correlate with heap fragmentation (GA residency) and the percentage of work requests of the total number of messages seems to indicate the degree of load imbalance.

Having assessed application characteristics in relation to scaling and performance of the applications, we observe that careful parallelisation is required to avoid pitfalls that result in excessive overhead. 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.
5 Conclusions

We have characterised a set of small and medium-sized parallel functional applications run on a multi-core and on a cluster of multi-cores in terms of communication rate, heap and GA residency, allocation rate, and thread granularity. We found that profiling reveals diverse bottlenecks and helps gain insight into dynamic application behaviour. In particular, the results indicate that:

- Thread subsumption works well across applications and architectures, as RTS is able to handle a large number of light-weight threads and prune superfluous parallelism by merging computations into a single thread.
- Compared to SMP, GUM is less aggressive in instantiating parallelism, i.e. generates fewer threads of larger granularity, adapting to system latency (the higher the latency, the lazier the instantiation).
- System-level information (e.g. GA residency representing heap fragmentation and the fraction of FISH messages in relation to total number of messages) appears promising for improving policy control decisions.
- Communication rate and GA residency vary considerably across applications and have a high impact on parallel performance.
- Increased memory residency and GC-percentage in a shared-memory design limit scalability due to contention on the first generation heap, in contrast to a distributed-memory design (confirming the results from [1]).

The insights from this characterisation inform the design of a dynamic adaptation mechanism based on monitoring a set of relevant parameters and dynamically tuning related policies. For instance, we are currently implementing a spark co-location mechanism that leverages information on spark source sites to attempt to steal work from the same source to improve locality and reduce fragmentation of the shared heap for applications with multiple sources of parallelism. Additionally, architectural information on communication latency and computational power of different PEs could be used to further improve co-location decisions. Another adaptation we plan to investigate uses stealing failure information to throttle FISHing if the overall load is low, alongside sparking rate to distinguish workers from parallelism generators that could temporarily switch to work pushing to speed up the dissemination of tasks across the PEs.

Finally, we envision a flexible RTS that uses an architecture-aware cost-model to dynamically adapt parallelism management policies at run time to systematically achieve high performance portability across heterogeneous architectures whilst retaining productivity by using a uniform high-level programming model.

Acknowledgements

We are grateful to Rob Stewart, Natalia Chechina, and Julian Godesa for inspiring discussions and helpful comments, and to the Scottish Informatics and Computer Science Alliance (SICSA) for support.
References