Towards a Characterisation of Parallel Functional Applications[*]

Evgenij Belikov Hans-Wolfgang Loidl Greg Michaelson

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

Abstract

To devise novel parallelism control mechanisms further insights into dynamic behaviour of parallel functional programs and run-time systems (RTS) are needed. We use profiling to characterise eight applications on a multi-core and on a multi-core cluster. We focus on thread granularity, memory management and communication. Our results confirm that parallel Haskell implementations cope well with large numbers of potential threads, identify memory management overhead as a key limiting factor in a shared-memory RTS, whilst in the distributed RTS, the amount of sharing determines the dominant communication overhead.

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 with non-uniform memory access (NUMA) and interconnect [2, 5]. 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* [15].

Although functional languages have numerous widely recognised benefits such as high level of abstraction for productivity, higher-order functions for composability, and purity which facilitates parallelisation and allows for sequential debugging [20, 7, 9, 19, 6], among others, tuning of parallelism control remains challenging, since the *operational cost model* is often non-intuitive because low-level coordination decisions are taken implicitly by the RTS [10]. 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. Thus manual tuning appears infeasible due to rapid architectural evolution, mandating automated solutions.

Characterising dynamic behaviour of eight parallel functional programs on a modern 48-core machine and a cluster consisting of 8-core nodes (we use up to 64 cores in total) is a step forward in gathering insight into attributes that can be exploited to improve the effectiveness and efficiency of parallelism management and useful for comparing run-time systems. The results confirm that the parallel Haskell [16] implementations cope well with large numbers of potential threads, quantify the impact of sharing (heap fragmentation due to distributed data structures) on communication overhead in a distributed-memory RTS (GHC-GUM) [18], and identify memory management overhead as a key limiting factor to scalability for a shared-memory RTS (GHC-SMP) [12].

[*]This work is partly supported by the Scottish Informatics and Computer Science Alliance (SICSA).

2 Parallel Applications

Most applications[1] were adopted from [14, 11] and grouped by the used parallelism pattern. We investigate how program characteristics change across different run-time systems and architectures with varying number of PEs.

Five applications use the *Divide-an-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 regular and flat (i.e. not nested) `parfib` computes the number of function calls for computation of the Nth Fibonacci number using arbitrary-length integers (input: $N = 50$, threshold of 23); the naive exponential implementation is primarily aimed at assessing thread subsumption capabilities of the RTS.
- The `worpitzky` program, from the domain of symbolic computation, checks the Worpitzky identity for two given arbitrary-length integers (19 to the exponent of 27, threshold 10).
- The `queens` program determines the number of solutions for placement of N queens on an $N \times N$ board without attacking each other ($N = 16$); a solution is a list of integers generated by discarding unsafe positions, which results in sharing of data structures at RTS level.
- The `coins` program computes ways to pay out a specified amount (5777) from a given set of coins.
- The `minimax` application calculates winning positions for a Noughts-vs-Crosses game on a $N \times N$ board up to a specified depth using alpha-beta search and lazyness to prune unpromising sub-trees ($N = 4$, depth 8).

Three 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 explicit granularity tuning.

- The `sumeuler` program computes the sum over *Euler Totient* numbers in a given integer interval and is fairly irregular ([0..100000], chunk 500); all the parallelism is generated in the beginning of the execution.
- The `mandelbrot` application computes the fairly irregular *Mandelbrot* fractal set for a given range and image size as well as number of iterations (range [-2.0..2.0], 4096x4096 image size, 3046 iterations).
- The `maze` program is a nested data-parallel AI application which searches for a path in a maze (of size 29).

The benchmarks are implemented in Glasgow parallel Haskell [16], a dialect of Haskell [8], which implements semi-explicit annotation-based model of parallelism where most of parallelism management is implicit. Although similar benchmarks have been used in the past, we run measurements on configurations with significantly larger number of PEs which enables us to better assess scalability, in particular on a distributed-memory platform. We show that where previously almost linear scaling has been reported on few cores, scalability often reaches a plateau well before all PEs are fully utilised on modern server-class multi-cores and multi-core clusters.

Additionally, we use larger input sizes and obtain more detailed profiles. We have extended the profiling infrastructure to record *per-lightweight-thread* granularity information as well as to collect more detailed summary statistics on protocol messages and sizes (specific to the distributed RTS). Relevant background information on high-level programming models and RTS-level policies can be found in [3]. Both run-time systems [18, 12] implement a variant of *work-stealing* for load balancing, where idle PEs ask randomly chosen PEs for work [4].

3 Application Characterisation

We report relative speedups and profiles from a median run out of three on a dedicated multi-core and on a lightly loaded cluster of multi-cores as as we are primarily interested in the parallelism behaviour of the 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 (512GB in sum). Average memory latency is 16ns with up to 3x difference depending on the NUMA regions. The `beowulf` cluster comprises 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, connected via Gigabit Ethernet with average latency of 150ns. We use CentOS 6.5, GHC 6.12.3[2], gcc 4.4.7, and PVM 3.4.6.

[1] Application source code can be obtained from `http://www.macs.hw.ac.uk/~eb96/atps15benchmarks.tar.gz` or via email.
[2] Some experiments using GHC 7.6.3 show similar trends for SMP; GUM has not yet been ported to more recent GHC versions.

3.1 Performance and Scalability

We fix the input size and increase the number of PEs to assess scalability. Run time decreases as the applications are able to profitably exploit 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[3] runs on higher numbers of PEs which indicates a scalability issue.

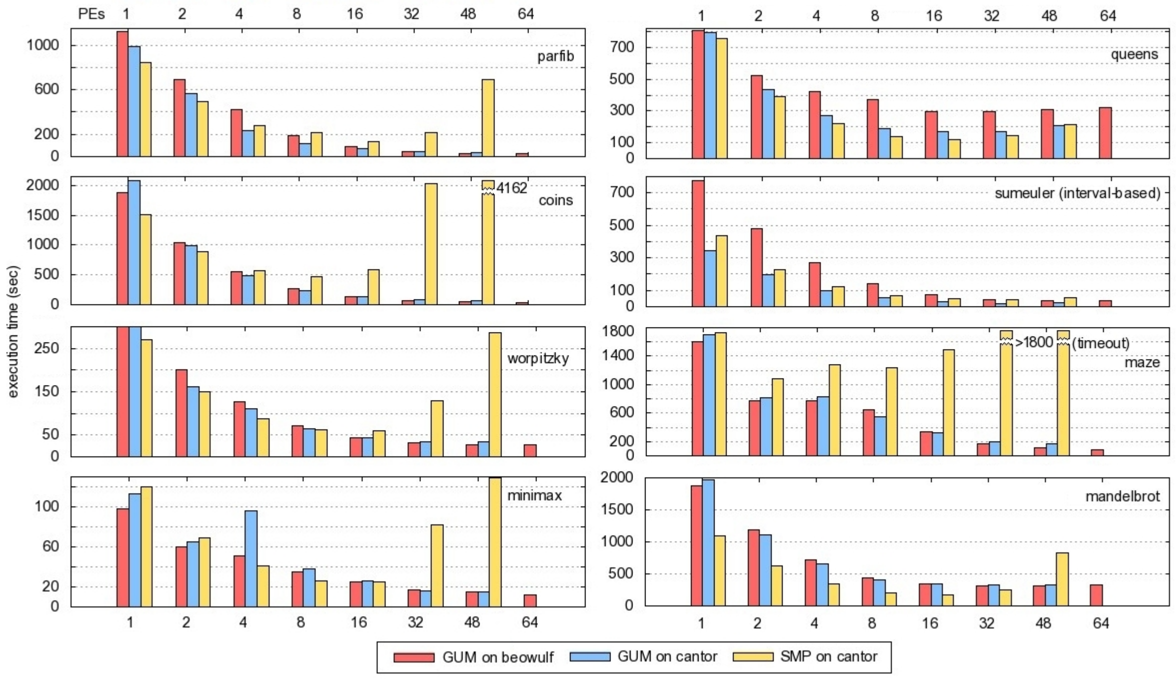

Figure 1: Application Execution Times

In Figure 1 we observe strong scaling for `parfib` and `coins` for GUM and good scaling for `sumeuler` with load balancing issues on high numbers of PEs. 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. SMP shows best performance for relatively low number of PEs, whilst on 48 cores a memory management issue discussed in Section 3.3 leads to a slowdown for 5 out of 8 programs. Moreover, `queens`, `mandelbrot`, and `minimax` exhibit limited scalability due to excessive communication and heap residency, which hints at improvement potential at application level. Increasing granularity for `worpitzky` is deemed likely to improve performance as currently median thread size is very small and the number of threads very high compared to programs that scale better.

3.2 Granularity

We have extended run-time profiling capabilities of GUM and SMP to record thread granularity information[4]. Unlike SMP, GUM RTS instances maintain private heaps and thus avoid GC-related synchronisation overhead. In Figure 2 we present application granularity profiles[5], grouped by the RTS and architecture. A *log-log* scale is used for comparability due to orders of magnitude differences in thread sizes (x-axis) and numbers (y-axis).

For `parfib`, `coins`, and `worpitzky`, we observe an order of magnitude less and larger threads for GUM than for SMP, which demonstrates effectiveness of GUM's *thread subsumption* mechanism and the aggressiveness of SMP's thread creation for D&C applications. Subsumption allows to implement *advisory* parallelism where the RTS decides whether to inline child threads into the parent or execute them in parallel, similar to lazy futures [13]. The shapes of the profiles for GUM on `cantor` are to a large extent similar to the shape of the profile on `beowulf`, but differs distinctively from SMP profiles, suggesting that RTS-level parallelism management policies have a strong influence on granularity, especially if the architectural features are not explicitly taken into account.

[3]We use SMP and GUM as a shorthand for GHC-SMP (shared-memory RTS) and GHC-GUM (distributed RTS), respectively.
[4]Profiling overhead is negligible as it involves mostly counters and is amortised by other dominant overheads (e.g. GC).
[5]Application names are found in the upper right corner of each histogram, so that three rows in a column represent an application.

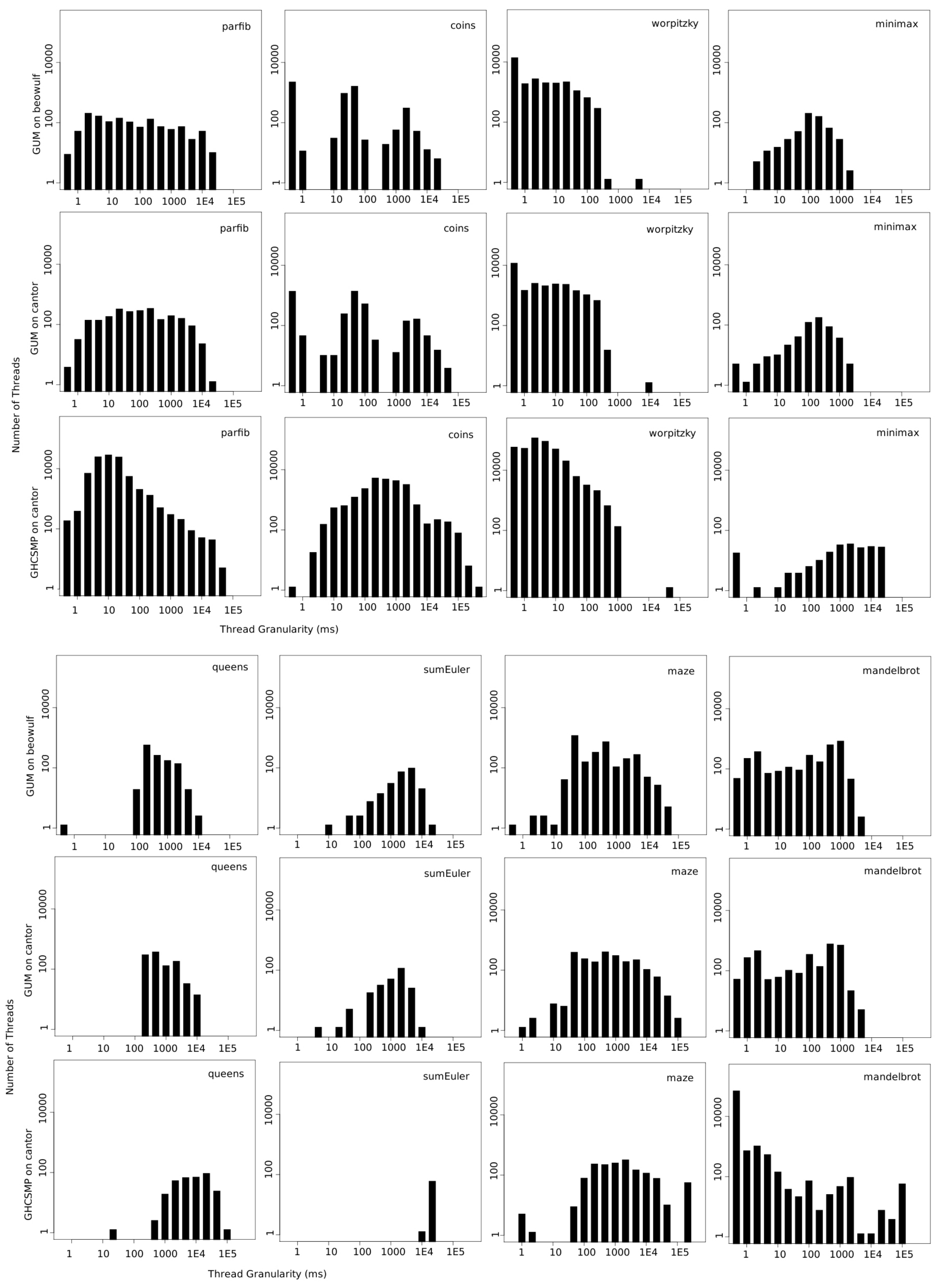

Figure 2: Run Time Granularity (GUM vs SMP on `cantor` using 48 PEs)

Parallelism is often over-abundant and fine-grained in functional programs, leaving considerable *parallel slackness* and requiring an effective thread subsumption mechanism. We observe a wide range of *actual* and *potential* parallelism degrees across applications. For instance, `sumeuler` only has 200 potential threads which is insufficient to keep all PEs busy, whereas for `coins` and `worpitzky` there are four orders of magnitude more potential threads, 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. The systems *automatically adapt* the degree of actual parallelism to the number of the available PEs.

By contrast, thread subsumption is ineffective in flat data-parallel applications, where parallelism is created at the start of the computation. Hence, to exploit the subsumption mechanism, data-parallel applications appear to require nested parallelism. We observe optimisation potential for D&C applications: recognising and inlining smaller threads as well as reducing the spread of the granularity distribution and the number of threads whilst preserving larger threads.

3.3 Memory Use and Garbage Collection

Many parallel functional programs are memory-bound as they perform graph reduction. Figure 3 depicts GC overhead – a reason for scalability issues for SMP. The GC% increases consistently across all applications for SMP and results in severe contention on the first generation heap. By contrast, GUM starts off with higher GC% which then drops or remains constant in most cases. 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 and can be used to compare aggressiveness of work allocation across run-time systems. In Figure 4 we observe initially higher allocation rates for SMP that are then dropping faster than for GUM.

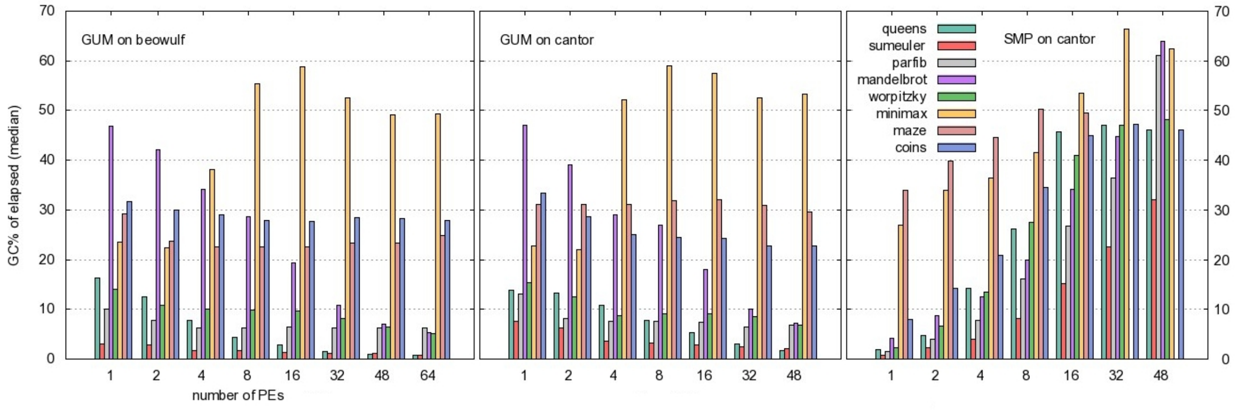

Figure 3: Garbage Collection Overhead

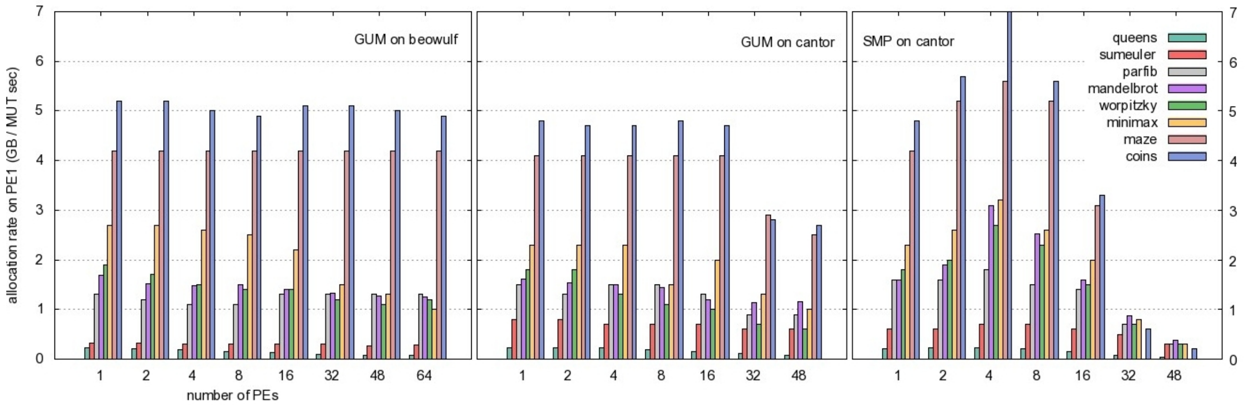

Figure 4: Allocation Rates

Application *working sets* are represented by *heap residency*. We observe roughly constant or decreasing residency for GUM on both distributed- and shared-memory architectures (except for `minimax`), whilst for SMP the residency is growing in most cases, as due to contention some heap-allocated objects are retained for longer.

3.4 Sharing and Communication

GUM uses virtual shared memory, so each RTS instance maintains a Global Address (GA) table of stable inter-processor pointers which are used as roots for GC. *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. Based on this metric, our application set can be partitioned into two classes: most of the applications shown in Figure 5, exhibit a moderate GA residency of at most 600 per PE; in contrast to this behaviour, `worpitzky` reaches a value of 2500 for a large number of PEs, and even worse `mandelbrot` (not shown) reaches a GA residency of 8000, and `queens` (not shown) of over 250000. This points to a high degree of sharing in the program due to poor data distribution and locality, which incurs a lot of communication and becomes a bottleneck for parallel performance, hinting at potential for application-level optimisation.

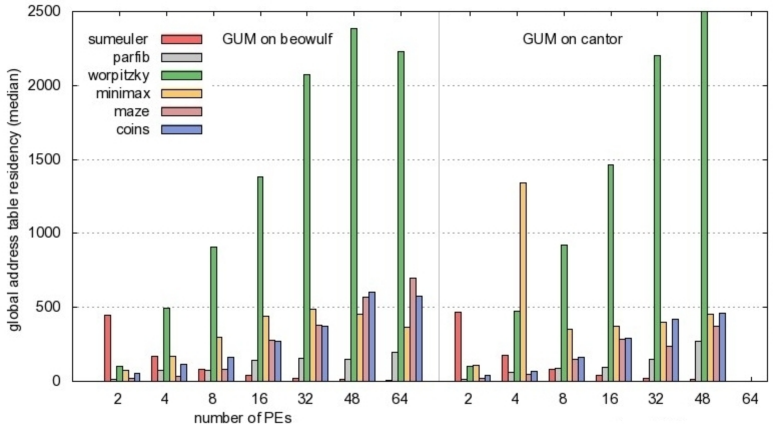

Figure 5: Global Address Table Residency (Heap Fragmentation)

As shown in Figure 6, for `parfib`, `coins`, `maze`, and to lesser extent `minimax` and `sumeuler` we observe modest linear increase in communication rate with less than 40% of work request messages. The median of the graph sent usually increases slightly, but only for `queens` it is excessive with over 14MB median graph sent per mutation second on 48 cores, as communication rate skyrockets (840k messages on 48 cores with frequent very long fetches (when a light-weight thread blocks waiting for another), with only 15% of the messages being work requests).

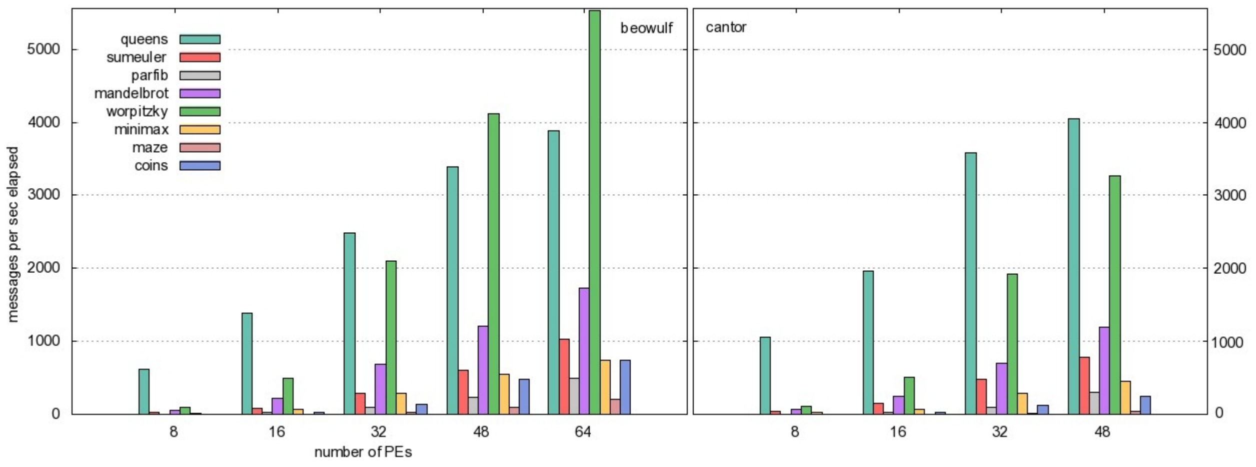

Figure 6: Communication Rate Comparison for GUM

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 are work requests, due to very fine thread granularity. Then `sumeuler` follows, illustrating another issue — lack of inherent parallelism for the given chunk size 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.

Careful parallelisation is required to avoid pitfalls that result in excessive overhead and limit scalability. We have found that semi-explicit parallel functional programs have the potential to scale, provided there is enough work on the one hand, but that the granularity and sharing are adequately controlled, on the other.

4 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 scalability, granularity, GC overhead, global address table residency, allocation rate and communication rate, among other metrics. We have found that profiling reveals diverse bottlenecks and helps gain insight into dynamic application behaviour. In particular, the results indicate that:

- Subsumption works well across architectures and D&C applications, as the RTS is able to handle a large number of potential light-weight threads and prune superfluous parallelism by subsuming computations into the parent thread. For data-parallel programs nesting appears to be necessary to benefit from subsumption.
- Compared to SMP, GUM is less aggressive in instantiating parallelism, i.e. generates fewer threads of larger granularity, adapting to the number of PEs and to system latency (the higher the latency, the lazier the instantiation). Additionally, granularity varies across run-time systems but seems relatively similar for the same RTS across different architectures.
- High GA residency is a good indicator of heap fragmentation due to sharing, which in turn causes a high degree of communication, limiting the scalability of the application.
- Communication rate and GA residency vary considerably across applications and have a high, direct impact on parallel performance.
- System-level information (e.g. granularity profiles, GA residency representing heap fragmentation and the fraction of work requests in relation to the total number of messages) appears promising for improving dynamic policy control decisions at RTS level.
- Increased memory residency and GC-percentage in a shared-memory design point to a scalability issue due to contention on the first generation heap, in contrast to a distributed-memory design, confirming the results from [1]. This suggests higher scalability potential of the RTS design that uses private heaps.

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 *co-location* mechanism for potential threads that leverages *ancestry information* to encourage stealing work from the nearby sources 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 [17].

Ultimately, we envision a growing set of representative parallel functional benchmark applications that could be used (similar to mainstream languages) to compare different novel parallelism management policies and mechanisms, aiming at high performance portability across heterogeneous architectures whilst retaining productivity.

Acknowledgements

We are grateful to Natalia Chechina, Julian Godesa, Rob Stewart, and Prabhat Totoo for inspiring discussions and to the anonymous reviewers who helped improve the presentation and the discussion of the results.

References

[1] M. Aljabri, H.-W. Loidl, and P. Trinder. Distributed vs. shared heap, parallel Haskell implementations on shared memory machines. In *Proc. of Symp. on Trends in Functional Programming*, University of Utrecht, The Netherlands, 2014. To appear.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *CACM*, 52:56–67, October 2009.

[3] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Heriot-Watt University, December 2013.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[5] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

[6] K. Hammond. Why parallel functional programming matters: Panel statement. In A. Romanovsky and T. Vardanega, editors, *Ada-Europe 2011*, volume 6652 of *LNCS*, pages 201–205, 2011.

[7] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

[8] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of Haskell: being lazy with class. In *Proc. of the 3rd ACM SIGPLAN History of Programming Languages Conference*, pages 1–55, June 2007.

[9] J. Hughes. Why functional programming matters. *Research Directions in Functional Programming*, pages 17–42, 1990.

[10] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GpH. *Concurrency: Practice and Experience*, 11:701–752, 1999.

[11] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better Strategies for parallel Haskell. In *Proc. of the 3rd Symposium on Haskell*, Haskell '10, pages 91–102. ACM, 2010.

[12] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ACM SIGPLAN Notices*, volume 44, pages 65–78, 2009.

[13] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[14] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

[15] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[16] P. Trinder, E. Barry Jr., M. Davis, K. Hammond, S. Junaidu, U. Klusik, H.-W. Loidl, and S. Peyton Jones. GpH: An architecture-independent functional language. *IEEE Transactions on Software Engineering*, 1998.

[17] P. Trinder, M. Cole, K. Hammond, H.-W. Loidl, and G. Michaelson. Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*, 25(3):309–348, 2013.

[18] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI'96 Conf.*, 1996.

[19] P. Trinder, K. Hammond, H-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[20] S. Vegdahl. A survey of proposed architectures for the execution of functional languages. *IEEE Transactions on Computers*, 33(12):1050–1071, December 1984.