GUMSMP: A SCALABLE PARALLEL HASKELL IMPLEMENTATION

by

Malak Saleh Aljabri

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

University of Glasgow
College of Science and Engineering
School of Computing Science

November 2015

Abstract

The most widely available high performance platforms today are hierarchical, with shared memory leaves, e.g. clusters of multi-cores, or NUMA with multiple regions. The Glasgow Haskell Compiler (GHC) provides a number of parallel Haskell implementations targeting different parallel architectures. In particular, GHC-SMP supports shared memory architectures, and GHC-GUM supports distributed memory machines. Both implementations use different, but related, runtime system (RTS) mechanisms and achieve good performance. A specialised RTS for the ubiquitous hierarchical architectures is lacking.

This thesis presents the design, implementation, and evaluation of a new parallel Haskell RTS, GUMSMP, that combines shared and distributed memory mechanisms to exploit hierarchical architectures more effectively. The design evaluates a variety of design choices and aims to efficiently combine scalable distributed memory parallelism, using a virtual shared heap over a hierarchical architecture, with low-overhead shared memory parallelism on shared memory nodes. Key design objectives in realising this system are to prefer local work, and to exploit mostly passive load distribution with pre-fetching.

Systematic performance evaluation shows that the automatic hierarchical load distribution policies must be carefully tuned to obtain good performance. We investigate the impact of several policies including work pre-fetching, favouring inter-node work distribution, and spark segregation with different export and select policies. We present the performance results for GUMSMP, demonstrating good scalability for a set of benchmarks on up to 300 cores. Moreover, our policies provide performance improvements of up to a factor of 1.5 compared to GHC-GUM.

The thesis provides a performance evaluation of distributed and shared heap implementations of parallel Haskell on a state-of-the-art physical shared memory NUMA machine. The evaluation exposes bottlenecks in memory management, which limit scalability beyond 25 cores. We demonstrate that GUMSMP, that combines both distributed and shared heap abstractions, consistently outperforms the shared memory GHC-SMP on seven benchmarks by a factor of 3.3 on average. Specifically, we show that the best results are obtained when sharing memory only within a single NUMA region, and using distributed memory system abstractions across the regions.

In The Name Of Allah, The Most Gracious, The Most Merciful.

"Allah brought you forth from the wombs of your mothers when you knew nothing, and He gave you hearing, sight and intelligence so that you may give thanks to Him."(Quran, 16:78)

Acknowledgements

First and foremost, all praises and thanks be to Allah, the Almighty, for giving me the health, strength and ability to complete this research.
The work presented in this thesis would not have been possible without the encouragement and support of numerous people, who I would like to thank from the bottom of my heart.

Professionally, I would like to offer my unreserved appreciation and gratitude to my supervisor, Prof. Phil Trinder, for his brilliance and amazing insight, reinforcement throughout the course of the research, and his great efficiency when checking and correcting my thesis. I am truly grateful to him for always believing in me and encouraging me to do my best. His willingness to listen, and constant support helped me greatly in overcoming the difficulties I faced during the research, both academically and personally. Without his reassurance and support, this work would not have been completed. One simply could not wish for a better, more supportive supervisor.

I also owe my eternal gratitude to my supervisor Dr.Hans-Wolfgang Loidl, who taught me much during the course of this thesis. He provided me with technical support, invaluable guidance, and a wealth of knowledge. What I learned from him is immeasurable. I have highly profited from our weekly hacking sessions, regular meetings, stimulating discussions, and productive ideas sharing, which added considerably to the positivity of my experience and made this work possible.

Special thanks go to Prof. Greg Michaelson for his willingness to always help, listen, and give thoughtful suggestions and advice, which assisted me greatly. I am deeply grateful to him for his time, support, and encouragement.

I would also like to thank the people in the computer science department at Heriot-Watt University, and at the University of Glasgow, for their support and

assistance.

Special thanks are also due to the embassy of Saudi Arabia, and Umm-Alqura university (which I am very keen to work at on my return) for giving me this opportunity, believing in my ability, and providing me with the financial support to complete my PhD.

Thanks also goes to all my relatives and friends here in the UK, and in Saudi Arabia, for their reassurance and backing.

Personally, I wish to extend special thanks and love to my parents, and my brothers and sisters, for their steadfast support during my stay in the UK. The continual positivity and prayers of my parents have been extremely instrumental in motivating me to pursue the research that led to this achievement.

Words cannot begin to express the debt of gratitude, love, and admiration I feel for my husband Naif, and my son Abdulaziz, and my little daughter Danah, for their care and sacrifice during my studies. I doubt that I will ever be able to convey my appreciation to them fully, but I owe them my sincere thanks. Their impact on me has been greater than I could say. I am truly grateful. This thesis is, therefore, dedicated to them.

Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Malak Saleh Aljabri

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Multi and many-core architectures have become the dominant general purpose hardware. Moreover, the current trend in parallel architectures has shifted towards hierarchical architectures, in which several shared memory units (i.e. multi-cores, or NUMA regions) are connected via a network. In high-performance computing, a hybrid parallel programming model is frequently used to best exploit such architectures. This combination requires multi-level parallel programming in both a shared memory model and a distributed memory model. For example, a directive-based parallelism through OpenMP [36] on a physical shared memory multi-core node[1], combined with message passing coordination, through MPI [125] across the cluster is often used on large scale clusters. While such a model can effectively exploit both shared and distributed memory compute resources, managing two abstractions is a burden for the programmer and increases the cost of porting to a new platform.

In contrast, our new runtime system (RTS) for parallel Haskell, GUMSMP, provides a *uniform, semi-explicit, high-level parallel programming model*, with adaptive, automatic policies at both levels of the hierarchy. The model relieves the programmer from the burden of explicitly controlling coordination in a multi-level hierarchy, delegating control almost entirely to the RTS.

Glasgow Parallel Haskell (GpH) [162] is a widely-used parallel extension of Haskell, a lazy functional language. GpH was developed to facilitate parallel programming by limiting the programmer's work to specifying a few key aspects

[1]The terms: node, core, PE, and HEC are defined in the Glossary

of coordination. The remaining low-level coordination aspects, such as communication, synchronisation, distributed garbage collection are managed by a sophisticated language implementation.

There are two different implementations of the same semi-explicit programming model GpH, namely: GHC-SMP [119]: a low-overhead physical shared memory implementation integrated in GHC, and GHC-GUM [163, 164]: a virtual shared memory implementation on clusters built on top of explicit message passing. A major difference between these two implementations lies in the work distribution models that are supported. Work distribution in GHC-GUM is achieved by message passing. In contrast, in GHC-SMP tasks can exploit shared memory to directly access a shared work pool.

In this thesis, we present the design, implementation, and performance evaluation of GUMSMP, a multi-level GpH implementation. GUMSMP smoothly integrates the work distribution policies of GHC-SMP and GHC-GUM, thereby providing a platform for *scalable parallelism* not bounded by the limitations of physical shared memory. GUMSMP combines the work distribution of GHC-SMP, using a shared memory model within a single multi-core, and the work distribution of GHC-GUM, using a distributed memory model across a hierarchy of multi-cores. GUMSMP is mainly designed to target cluster of multi-cores hierarchical architectures, but the trends of the modern architectures are driving more and more architectures into this space such as large NUMA.

The heterogeneity that the thesis deals with is that GUMSMP is designed for systems with hierarchical memory, e.g. clusters of multi-cores or NUMA architectures. It does not, however, deal with accelerators like GPUs or FPGAs.

1.1 Thesis Statement

The most widely available high performance platforms today are hierarchical, with shared memory leaves, e.g. clusters of multi-cores, or NUMA with multiple regions. These are often programmed using a hybrid parallel programming model e.g. combining a message-passing and a shared-memory model. This thesis *investigates whether a multi-level parallel implementation can effectively exploit*

hierarchical architectures and achieve scalability, while still using a single, high-level programming model. Moreover, can a hierarchical implementation balance the use of NUMA shared memory without additional operating system support? We investigate the hypothesis by providing a new parallel Haskell implementation GUMSMP that combines the load balancing mechanisms of the distributed memory implementation GHC-GUM, and the shared memory implementation GHC-SMP. The performance of GUMSMP is evaluated on clusters of multi-cores, and on NUMA machines using an established set of parallel Haskell benchmarks.

1.2 Contributions

We have designed, implemented, and experimented with a new implementation for the parallel Haskell dialect GpH, targeting hierarchical parallel architectures e.g. clusters of multi-cores. Our implementation GUMSMP smoothly integrates the work distribution policies of GHC-SMP (for shared memory) and GHC-GUM (for distributed memory), thereby providing a hierarchy-aware platform for scalable parallelism not bounded by the limitations of a physical shared memory. In particular, the GUMSMP implementation provides scalability by using distributed memory techniques to exploit multiple shared memory nodes, while still providing a single programming model. The thesis makes the following contributions:

- The design and implementation of a new and sophisticated shared and distributed memory parallel runtime system for a production functional language (GUMSMP based on GHC 6.12.3). Accounting for the hierarchical nature of modern architectures like clusters of multi-cores, we provide a design for an improved load distribution mechanism [6] (Sections 3.5.3, 3.5.4).

- The development and the evaluation of the effectiveness of different policies to improve the automatic hierarchical load distribution. In particular, the *low-watermark* mechanism for work pre-fetching, showing improvements of up to a factor of 3 comparing GUMSMP with and without *low-watermark*

(Section 4.3.1), and favouring inter-node work distribution, showing a further improvement of up to 19% over the *low-watermark* (Section 4.3.2). Moreover, a novel spark segregation mechanism is studied to separate local and global sparks, identifying different policies to export sparks remotely and select sparks for local evaluation (Section 4.3.3). As well as the effect of using dedicated gateways, thus restricting one core for communication work (Section 4.3.4). In terms of reducing the bottlenecks of memory management overhead, we discuss further tuning by optimising the number of cores per PE (Sections 4.3.5, 5.3.3), as well as adjusting the heap settings by providing larger allocation area which consistently improve performance by a factor of up to 1.4 over the default heap setting (Sections 4.3.6, 5.3.4). Furthermore, we show that combining active and passive load distribution for sparks at the intra-node level delivers improvement of up to 22% over passive load distribution for sparks (Sections 4.3.7, 5.3.6).

- The thesis reports a systematic performance evaluation of GUMSMP in comparison to GHC-SMP and GHC-GUM using a set of benchmarks on both cluster and NUMA architectures. Compared with GHC-SMP, GUMSMP delivers performance improvement of a factor of 3.3 on average on a NUMA machine with 40 cores by balancing the shared and distributed heaps. Investigation of the scalability limits of GHC-SMP reveals that the garbage collection represents a main source of overhead [8] (Section 5.2). Compared with GHC-GUM, GUMSMP provides an improvement of up to a factor of 1.5 on average on a cluster of multi-core architecture with up to 128 cores by exploiting the specifics of shared memory (Section 5.3). We show that GUMSMP scales to deliver a speedup of up to 175 on a cluster with 100 nodes, comprised of 300 cores (Section 5.4).

1.3 Publications

This thesis is closely based on the work reported in the following papers:

Refereed Papers: The author is the primary author of all papers (the Saudi Students Conference papers received light-weight refereeing).

- M. Aljabri, H.-W. Loidl, and P. Trinder, "The Design and Implementation of GUMSMP: A Multilevel Parallel Haskell Implementation", in Proceedings of the 25th Symposium on Implementation and Application of Functional Languages, ser. IFL '13. New York, NY, USA: ACM, 2014, pp. 37–48. Available: http://doi.acm.org/10.1145/2620678.2620682 [6].

- M. Aljabri, H-W. Loidl, and P.W. Trinder. "Balancing Shared and Distributed Heaps on NUMA Architectures", in Trends in Functional Programming, ser. Lecture Notes in Computer Science, J. Hage and J. McCarthy, Eds. Springer International Publishing, 2015, vol. 8843, pp. 1–17. Available: http://dx.doi.org/10.1007/978-3-319-14675-1 [8].

- M. Aljabri, P. Trinder, and H-W.Loidl, Overview of the Design of GUMSMP: a Multilevel Parallel Haskell Implementation. In: Proceedings of the Saudi Scientific International Conference 2012, London, UK, 11-14 Oct 2012. Saudi Scientific International Conference, London, UK, p. 25 [5].

- M. Aljabri, H-W. Loidl, and P.W. Trinder. "Assessing the Scalability Issues on Many-Core NUMA machines". In: Proceedings of the Saudi Student Conference 2015, London, UK, 1 Feb 2015. The 8th Saudi Students Conference, London, UK, 1037 [7].

Technical Reports: Additionally, the author contribute to the following technical report.

- 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, 16.12.2013. Available: http://www.macs.hw.ac.uk/cs/techreps/doc0103.html [18].

Chapter 2

Literature Survey

2.1 Introduction

A fundamental change in processor architecture means that the number of cores (processors) is increasing, with heterogeneous and hierarchical parallel architectures becoming the dominant general purpose hardware platforms. Such architectures drastically increase the computing resources and enable multiple parallel tasks to be computed faster than they could be on single-core machines. One of the major challenges facing the computing world today is how to get the most benefit from these architectures and use them efficiently. While traditional imperative programming languages are used for parallel programming e.g. C or Fortran with MPI on High Performance Computing (HPC) platforms, these approaches require additional programmer effort as the coordination specification is low level.

This chapter surveys the different parallel architectures, and discusses the challenges that have arisen from the recent trends toward hierarchical machines with different organisations of cores and memory. Moreover, we review parallel programming technologies, and demonstrate their advantages and disadvantages on parallel architectures. In particular, we discuss different parallel programming approaches, languages, and implementations with a greater focus on the high-level parallelism supported by parallel functional languages, which are related closely to our work on GUMSMP.

We separate the content of this survey into a discussion of parallel architec-

tures in Section 2.2, and the different parallel programming models, approaches, and patterns in Section 2.3. We then cover parallel languages in Section 2.4 offering an overview of parallel imperative languages, as well as parallel functional languages. We also discuss hybrid parallel programming models and different parallel systems. In Section 2.5 we focus on parallel Haskell implementations, and conclude by comparing different shared and distributed memory parallel Haskell implementations. Finally, in Section 2.6 we discuss load balancing as a central mechanism affecting the performance of parallel applications.

2.2 Parallel Architectures

Most parallel computers comprise three main building blocks: the cores, the memory modules, and the interconnection network. Over recent decades, there has been a gradual development in the degree of sophistication of each of these building blocks, but what makes one parallel computer different from another is the manner in which they are arranged. Parallel computing cores themselves are now essentially similar to those used in single-core systems with new technologies increasingly embedded with the cores for the parallel architectures with the aim to add more improvement or new functionalities of different parallel architectures, such as Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs) [152].

Parallel computers have been classified in different ways, but in all probability the earliest and the most widely used classification is *Flynn's Taxonomy* [61]. Flynn classified computing architectures into four categories, based on the number of instruction streams and data streams available: Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD), as demonstrated in Figure 2.1.

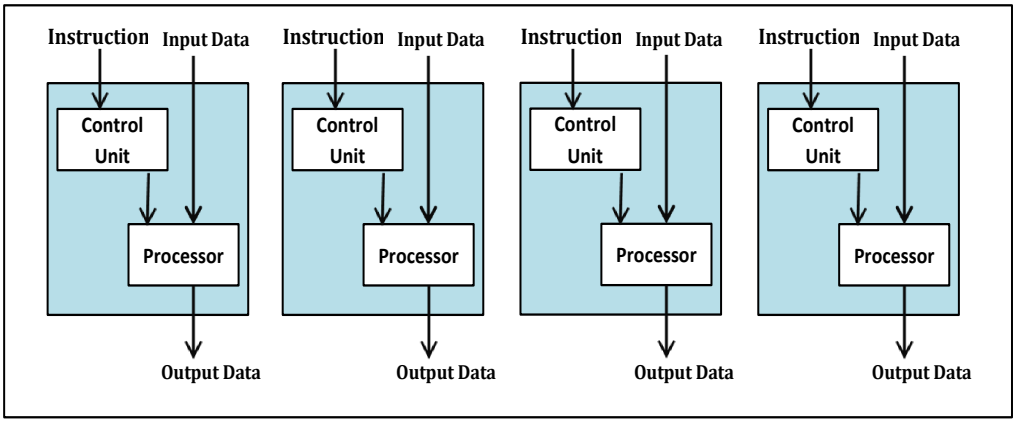

Figure 2.1: Multiple Instruction Multiple Data architecture [121]

This classification of parallel architectures represents a coarse model, as most parallel cores are hybrids from different categories [81]. Flynn taxonomy is broadly employed as initial classification of computer architectures. Nonethe-

less, it has various obvious disadvantages, the clearest of which is the fact that the MIMD class is overcrowded. The majority of multi-processing systems and multiple computer systems fall within this classification, including modern personal machines characterised by x-86 based multi-core processors [169]. A more recent classification of parallel architectures is based on the type of parallelism supported, i.e. function-parallel or data-parallel architectures [86, 65].

The MIMD category represents most contemporary parallel architectures, which are increasingly hierarchical and heterogeneous. Computers in this category consist of multiple cores; that is, they have different cores that perform different instruction streams on different data streams. Each core is able to work independently, as it represents an independent hardware unit for computation.

A number of related factors promote the development of MIMD architectures. Importantly, computers in this category offer a support for wider range of parallel patterns compared with other parallel architectures. Therefore, they are applicable to a wider range of parallel applications. Moreover, MIMD is extremely cost effective and this has undoubtedly been a great incentive for its promotion [52, 81].

Computers with MIMD architecture are further sub-divided into two main categories, based on the organisation of memory as Shared Memory and Distributed Memory. The main difference between the two is the organisation of the memory. In the case of shared memory MIMD, all cores share access to the same memory. In the second category, distributed memory MIMD, each core has its own local memory, and accesses values in other memories using the network. This means that different parts, or sub-tasks, of a computational task are distributed to multiple cores, each with its own memory space; and then, the results from each core are reassembled into one solution.

2.2.1 Shared Memory

In shared memory architectures, there is a single shared memory, which is available for all cores to access via a direct interconnected network. Communication among these cores can be achieved through a reading and writing to the shared memory, and occurs implicitly as a result of conventional memory access instructions. Shared memory systems [100] can be classified as: Symmetric

Multi-Processor systems (SMP), which, as this name implies, means all processors (cores) share memory and Input/Output (IO) equally, and can access the same memory location at the same speed, thus providing Uniform Memory Access (UMA) [37]. The second category is Non-Uniform Memory Access (NUMA) machines, which have recently gained popularity, and are shared memory machines, wherein the memory is closer to some cores, and therefore can access some memory locations faster than others [97].

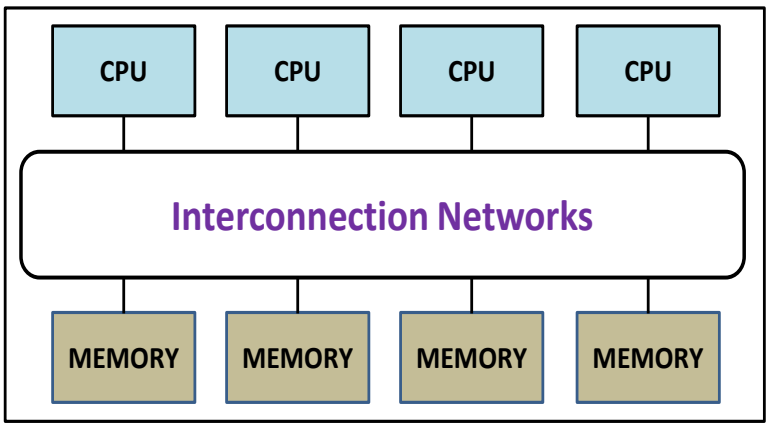

Figure 2.2: Generic shared memory architecture [121]

The current trend is towards many-core, shared memory machines; two or four cores are commonly utilised in devices used daily, including laptops and mobile phones. Moreover, by using hundreds or thousands of cores, it is now possible to deliver far greater computing performance than was previously possible, with affordable expenditure of energy, such as Intel's TeraScale [122] and Tilera's TILE-Gx100 [74], which use 80 and 100 cores respectively, whereas NVIDIA's GT300 GPUs uses 512 scalar cores [29].

Parallel applications on homogeneous many-core architectures are generally easier to implement than those on heterogeneous (e.g. a multi-core and GPUs) or distributed memory architectures, which require more low level coordination. However, the scalability of current shared memory machines is limited due to the increasing overheads resulting from maintaining cache coherency and providing efficient data access synchronisation [57].

2.2.1.1 NUMA

The use of a NUMA model for physical shared memory machines is one of the key trends in hardware design [97]. NUMA designs are now utilised by many multi-core servers, and it is anticipated that new and emerging many-core servers will adopt a NUMA design. The main aim of this design is to provide performance scalability for many-core machines with a large main memory.

In this model, the main memory is partitioned into several NUMA regions, each of which is associated with several cores. Access to the memory within the local region is fast; whereas, remote non-local access must pass through an on-chip network to access a different memory bank, and is, therefore, much slower. Moreover, this performance asymmetry intensifies as the number of cores in a single region increases, thus negatively affecting uniformity, especially for languages with automatic memory management [160].

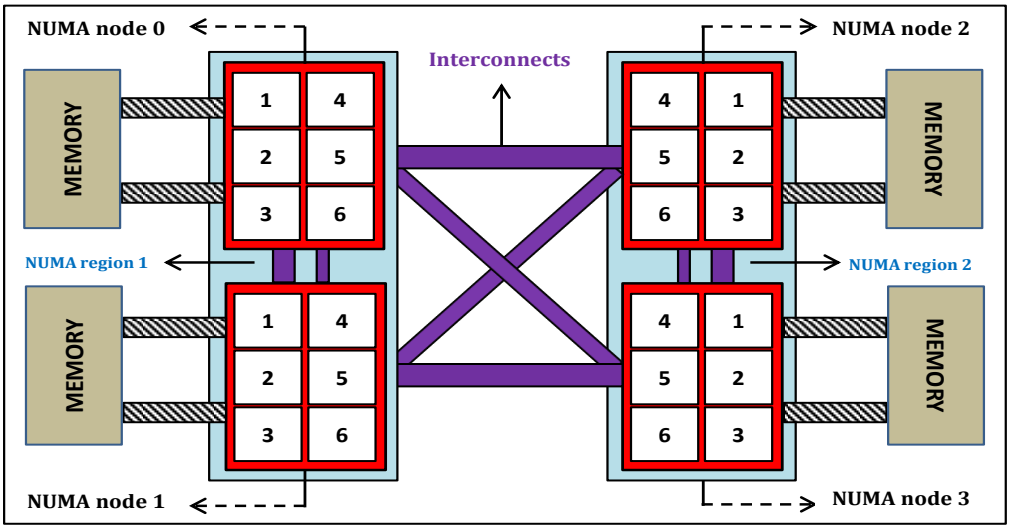

Figure 2.3: A typical NUMA architecture with 2 NUMA regions [47]

In the case of many-core, the NUMA design of the memory sub-system requires awareness of the differences in latency by the system or the algorithm, in order to avoid scaling issues. Furthermore, both effective memory bandwidth and latency to different regions can be negatively impacted by hardware problems [19]. Traditionally, the term NUMA is mainly used to characterise the structure of the memory sub-system. However, other resources, such as IO, are also generally

impacted by the asymmetry of the NUMA architectures, which can result in a substantial fluctuation in IO performance relative to latency and bandwidth, where remote IO access generates a higher latency and usually a lower bandwidth for data transfer [160]. The GUMSMP system we have designed allows us to explore the performance implications of different combinations of shared and distributed memory on NUMA architectures (Section 5.2).

2.2.2 Distributed Memory

In distributed memory MIMD machines, each core has its own private memory. This means no core can access the memories of the other cores in the machine; therefore, to be shared, data must be passed from one core to another as a message.

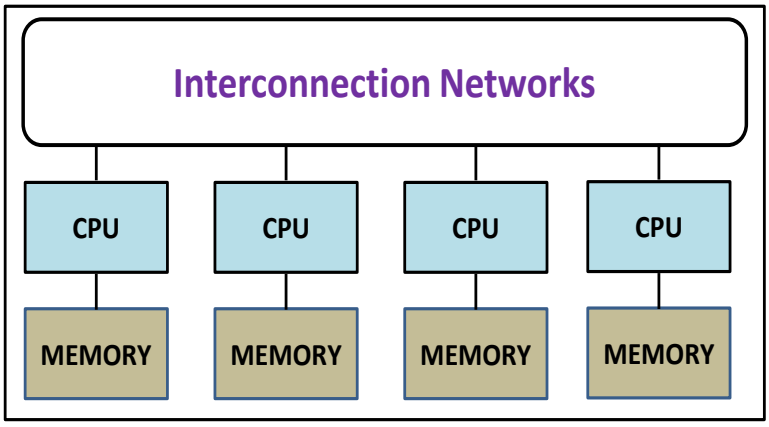

Figure 2.4: Generic distributed memory architecture [121]

There are two main classes of distributed memory architectures: Massively Parallel Processors (MPP) and clusters. With MPP [139], the cores and the network infrastructure are specifically designed to work closely in a single parallel computer. These systems are extremely scalable and thousands of cores can be supported by a single system that represents the principal architectures of super computing [92]. Clusters [44] on the other hand, are comprised of state-of-the-art computers connected by a network. Systems of this kind includes Grids [64], and Clouds [12] are becoming more widespread and powerful. Drivers for these changes are the improvements in network technology. These architectures are

suitable for large scale, complex, and HPC applications.

2.2.2.1 Heterogeneity

In the era of multi-cores, traditional parallel programming models based on homogeneous cores are often unable to meet the requirements of HPC applications. Consequently, parallel architectures are moving toward heterogeneity at different levels, including combining cores with different capabilities, as well as different hierarchies of memories and networks [30, 40, 149, 13].

Examples of heterogeneity are the use of accelerators such as GPUs and FPGAs that are commonly integrated in architectures [103] as depicted in Figure 2.5. GPUs excel at regular data-parallel applications using floating point operations; these have become increasingly important as powerful computing resources, not only for graphics, computer games, or films, but also in science, engineering and financial modelling [131]. FPGAs on the other hand, are suitable for computing intensive part of the applications such as matrix multiplications, as well as streaming applications using integer or logic operations [95].

This thesis deals with heterogeneity, in the sense that it is designed for clusters of multi-cores, so some cores share memory, where others are elsewhere in the cluster. It does not, however, deal with accelerators like GPUs or FPGAs.

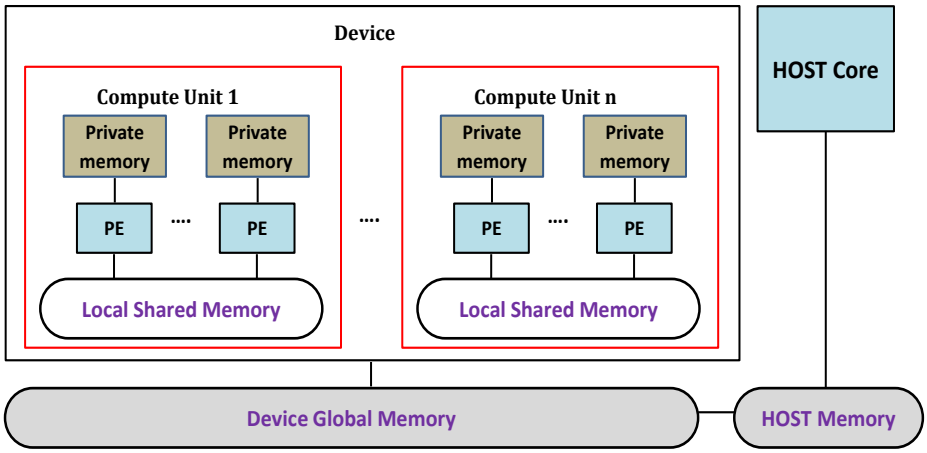

Figure 2.5: A typical heterogeneous architecture core(HOST) + GPU(Device)[18]

2.3 Parallel Programming

The aim of parallel computing is to increase an application's performance by executing the application on multiple cores, thereby increasing the speed of the execution. Using a set of cores that work cooperatively means that we can concentrate our computational resources cores, memory, or IO bandwidth on a given problem and achieve our aims at greater speed [173]. The other advantage offered by parallel computing is that it deals with problems that are too large or too complex to be handled by a single computer, especially computers with a limited memory [152]. A single computer resource can only perform a single task at a time, with the sequence of steps running in a specific order. With multiple computing resources, on the other hand, different tasks can be divided among a number of cores, which then makes it possible to execute the tasks simultaneously. In addition to this, non-local computer resources can be used on a wide area network [16]. All this increases not only the speed, but also the scope of what can be achieved.

However, the majority of current software was originally developed to work on a single-core, and will not easily benefit from the rapid parallel architectural evolution without a major redesign [82, 157]. The problem is even worse, as the vast majority of mainstream programming languages were not created with parallelism as a primary consideration, meaning that they are not particularly appropriate for exploiting parallel architectures [18]. In particular, the main key challenge facing the programmer is the creating of a parallel algorithm, which requires a number of steps not required in a sequential program. These steps correspond with coordinating parallelism across multiple processing elements (PEs).[1] These work cooperatively to exploit the underlying parallel resources. In particular, handling task decomposition to partition computation among PEs, mapping tasks to PEs, and handling communication and synchronisation issues [63], and as yet there is limited support in the form of tools, or standardised language level mechanisms to comprehensively deal with this need.

[1] PE is an abstraction of a core, see (Section 3.4.1, and Glossary). We use PE in the programming language context, and core in hardware level discussions.

When PEs are cooperating to perform a computation, they might require a method of communication; this can be achieved in various ways based on the underlying parallel machine. In shared memory computers, only a single process can be engaged in the shared memory at any given time [148]. To prevent, for example, a PE trying to access shared data before another PE has completed an updating task at the same data, low-level synchronisation mechanisms should be used, e.g. *locks*, or *barrier synchronisation*. Another key problem with all shared memory architectures, which permit the caching of shared variables, is cache coherence. When a PE wants to access a shared variable in its cache, as this variable has been accessible to other PEs, there is no way for the PE to know whether the data it is accessing has been modified by another PE. These issues have been exacerbated with the recent trend toward having multi-level caches. Hardware, cache coherence protocols are often used to provide coherency, by maintaining the invariant of a single writer, multiple readers for the shared memory [120]. On the other hand, programming for distributed memory machines involves different kinds of overheads to coordinate communications between the PEs in the form of message passing, which involves serialisation/de-serialisation of large data structures.

Coordination often results in extra overheads, and may lead to losing the potential performance gain completely if the coordination time (time spent in synchronising access to shared memory, or communicating large data structure over the network) exceeds the time spent doing useful independent computations. Moreover, care must be taken by the programmer in order to use synchronisation mechanisms, to ensure correctness, as the misuse of such mechanisms often leads to hard-to-detect and resolve problems, such as corrupted data, race conditions, or deadlocks [98, 18]. Furthermore, another challenge is the need to strike a balance between the communication cost when balancing the load to best exploit the underlying parallel architectures and preserving data locality by reducing communication cost [18].

2.3.1 Approaches

There is a continuum of parallel programming approaches from: implicit parallelism to explicit parallelism [142, 91, 83, 151]

1. **Implicit parallelism:** This approach, which is also called auto parallelisation or parallel compiler maintains distance between programmers and the coordination aspects of parallelism. Exploitation of parallelism is the responsibility of a sophisticated compiler and runtime system (RTS); although, completely implicit parallelism remains a difficult goal to achieve. There are several high-level parallel functional programming languages like Glasgow Parallel Haskell (GpH) [162] (Section 2.4.5.1), but with these the programmer still has to carry out some work specifying the coordination aspects; thus, this is considered semi-explicit parallelism. Our work of GUMSMP represents one implementation of GpH.

2. **Explicit parallelism:** This approach requires the programmer to explicitly specify the majority of the details of parallelisation; the extent to which this is the case depends on the level of the parallel programming languages being used. Parallelisation involves several tasks, such as the decomposition of tasks, mapping tasks to processing elements, and managing the communication of data between processing elements.

In principle, any combination of implicit and explicit is possible. Modern parallel languages tend to be implicit in terms of communication and synchronisation, and explicit in terms of decomposition and mapping. The implicit approach is more suitable for problems with a specific pattern or regular parallelism. In contrast, the explicit approach reduces productivity by assigning parallelism management to the programmer [18].

2.3.2 Levels of Abstractions

Parallel programming languages can be classified by the level of abstraction they provide for parallel programming, and the level of explicit parallelism control required by the programmer as low, mid, or high.

1. **Low level:** The low-level parallel programming languages require the programmer to specify all the details of parallel coordination such as task decomposition, communication, and synchronisation. An example of this is Java Socket, which is a very low-level interface for specifying the inter-process communication details [159]. It has been widely used to provide efficient and flexible distributed applications; however, it involves the writing of long and complex details [48]. Another example of a flexible, but low-level model is the Portable Operating System Interface (POSIX) Threads, which is used with shared memory and provides the programmer with explicit parallelism control at the threads level [32, 49].

2. **Mid level:** The mid-level languages abstract over some of the parallel coordination details, such as mapping work to PEs, but still require the programmer to manage others explicitly. The goal of this level is to create an equilibrium between programmer productivity, by increasing the level of abstraction; thus facilitating the programming task, and the parallel performance by increasing tuning approaches [18]. Primitives or libraries can be used efficiently with several mid-level computation languages, i.e. Fortran, C, and C++ [151]. An example of this approach is the Message Passing Interface (MPI) [125, 77], Parallel Virtual Machine (PVM) [71] libraries (Section 2.4.1.1), and Open Multi-Processing (OpenMP) [36] (Section 2.4.1.2), which are widely known and routinely used to support parallel programming.

3. **High level:** The main purpose of high-level parallel libraries and languages is to provide as much implicit parallelism as possible, by abstracting over most of the parallel coordination details. In such a model, the programmer will have a minimal level of parallelism control, often provided in the form of advisory parallelism specification. Examples of this include GpH [162] (Section 2.4.5.1), which is explicit about decomposition, represented by the primitive `par`, and is implicit about mapping, communication, and synchronisation. High-level parallelism is also supported by Partitioned Global Address Space (PGAS) languages, such as Chapel [35], X10 [38], and Fortress [9] (Section 2.4.3).

2.3.3 Patterns

To exploit parallel architectures efficiently, computation needs to be partitioned into smaller sub-tasks, which are mapped to PEs in a way that sets a suitable balance between computational load in order to exploit the parallel cores, and the preservation of data locality in order to reduce the overheads associated with the communication [18]. A general trend in the parallel languages community is to define the parallel programming paradigms in which applications can be classified. These patterns are defined to manage the load balancing component of parallelism.

The most popular parallel programming patterns used for most typical applications are presented in this section, and are summarised from [151, 121, 69, 154].

1. **Master/Workers (Task Farming):** The master/workers, or task farming, parallel programming paradigm is a problem decomposition model. The program comprises two entities: the master, and multiple workers or "slaves". The master's job is to decompose the problem into a number of tasks, and distribute them amongst the workers. Subsequently, when the computation has been completed, the master collects the partial results, so that the final computation result can be produced. The second entity, the "slaves" processes (or workers), execute tasks in a very simple cycle: they receive a message with the task, then process the task before returning the results to the master. The equal division of tasks among the workers, known as load balancing, is essential to improve the performance of any application. This can be done either statically or dynamically, as described below:

    - With static load-balancing, the task distribution is carried out at the start of the computation process. This means, as soon as each worker has been allocated their portion of the work, the master can participate in the computation. Task allocation can then be performed either once or cyclically.
    - Dynamic load-balancing, is a more suitable option when the number of tasks is greater than the number of PEs available, when the num-

ber of tasks is unknown at the beginning of an execution, or when the problem is unbalanced. The inbuilt flexibility of dynamic load-balancing enables the application to adapt to the changing conditions of the system. This feature of the dynamic load balancing paradigm allows it to respond if one or more PEs fail, thereby simplifying the creation of applications that are sufficiently robust to survive the loss of workers or even the master. High computational speed increases and good scalability can be achieved with this paradigm. However, with many PEs, the master PE becomes a bottleneck, compromising the application's scalability. This can be overcome by extending the single master to a set of masters, each of which is then responsible for controlling a different group of process workers, thereby improving the scalability of the paradigm.

2. **Data Parallelism:** Is the most frequently used paradigm; where parallelism is exploited by dividing a data structure into chunks which are assigned to PEs to perform specific task on those chunks. The approach taken here is to seek fine-grained inner loops inside the code and parallelising those inner loops, resulted in large number of chunks suitable to be worked on by large number of parallel PEs [28]. On distributed memory architectures, the chunks of the data structure exist in the local memory of the PE, and on shared memory architectures, all PEs have access to the data structures reside on the shared memory [16]. This type of parallelism is also referred to as geometric parallelism, or domain decomposition. Since in this design neighbouring PEs communicate with each other, the communication load will be determined by the size of the boundary of the element; and the volume of the data will determine the computation load. Provided the data is evenly distributed by the PEs, and the system is homogeneous, data parallel applications are extremely efficient, and highly scalable [151]. Data parallelism was initially first introduced to exploit parallelism in SIMD parallel architectures in the 1980s. Today, data parallelism is widely used to exploit parallelism in the recent massively parallel architectures such as General-Purpose computing on Graphics Processing Units (GPGPUs) [132].

3. **Data Pipelining:** Is a type of parallelism based on functional decomposition, and is one of the simplest pattern that is most commonly applied. The different tasks of the algorithm, which operate concurrently, are identified and each PE then executes its small portion of the total algorithm. The processes are arranged as different stages of the Pipeline, each one of which is responsible for executing a particular task. This type of parallelism, which is commonly used in applications for data reduction or image processing, is sometimes called "data flow parallelism" because the data flows between one stage of the pipeline and the next. The form of communication is very simple, and may be totally asynchronous.

4. **Divide and Conquer:** Is a frequently used approach in the design and development of sequential algorithms where a task is broken down recursively into a number of smaller sub-tasks, which sometimes represent smaller samples of the original task, each of which is computed independently. The results are then combined to produce the final, overall result. In parallel divide-and-conquer, the division of a task into smaller sub-tasks, and a combination of results can be carried out in parallel. The divide-and-conquer paradigm consists of three main phases: divide, compute, and combine. These are designed in the form of a virtual tree, whereby some of the PEs create sub-tasks, and the results then combined to produce an overall result. The computation of sub-tasks is carried out by the leaf nodes of the virtual tree.

2.3.4 Mechanisms

Parallelism may be provided by a number of different language mechanisms as follows.

1. **Language Primitives:** In this case, the entire language can be applied to compose the parallel execution. In this case, parallelism is specified as first-class primitive constructs, which are part of the language definition; thereby, increasing flexibility. An example is the Occam programming language [126].

2. **Language Extension:** In this case, an existing language is extended with parallelism support. Therefore, parallelism is expressed as constructs in the program. An example of this would be in GpH [162] (Section 2.4.5.1), where parallelism is specified as a specific construct `par` judiciously added by the programmer to the Haskell program to indicate those parts of the program that could be evaluated in parallel, and leave the management of parallelism to the underlying RTS.

3. **Coordination Language:** In this case, a new language is dedicated for parallelism coordination; thereby, separating computational activities from coordination. An example is Linda [72, 130], which serves as a framework that can be applied to a computational language such as C, in order to support parallelism coordination.

4. **Libraries:** In this case, parallelism is expressed in specific libraries such as MPI [125, 77], or PVM [71] (Section 2.4.1.1). The functions of those libraries can be called from a program written in computational languages such as C, C++, or Fortran.

5. **Compiler Directives (Pragmas, Annotations):** In this case, parallelism is expressed as comments for the compiler in the main program. Semantically, the comments do not change any aspect of the meaning of the main program, and can be ignored by the compiler. An example of this approach would be OpenMP [36] (Section 2.4.1.2).

2.3.5 Skeletons

Algorithmic skeletons [41, 76, 141] are a model of structured parallel programming that aim to simplify the task of developing parallel applications by abstracting generic and recurring computations and communication patterns within parallel programs as predefined application independent components, thus achieving a trade-off between the performance of the application, its portability and the programmer's productivity [43]. There are several advantages to using algorithmic skeletons [58]: they facilitate the development of parallel programs by abstracting over the low-level details of coordination aspects, and restricting the

programmer's job to writing the sequential fragments of the program and instantiate skeletons. Thus, it both provides efficient application and increases programmer productivity. Skeletons can be nested; thus implementing complex patterns [45, 123], taking another skeleton as an argument, or returning it as a result. In terms of functional programming, skeletons are higher-order functions, which can take other functions as argument. Algorithmic skeletons can be classified according to the type of parallelism represented [94, 138], as:

- Data parallel skeletons: in which a function is applied to every element of the data structure in parallel.
- Task parallel skeletons: such as pipelines or task farms in which a set of functions are applied in parallel to different elements of the data stream.

Some skeletons were recently developed for certain kinds of application domains, such as Google's MapReduce [46], and Apache Hadoop [150] for distributed data mining.

2.4 Parallel Languages

2.4.1 Imperative Languages

In imperative languages, variables represent memory locations with assignments used to manipulate those memory locations. Unlike functional languages, imperative languages are characterised by concepts such as pointers, loops, control statements, and side effects. Moreover, they establish specific steps to be taken by the machine to perform a given algorithm; whereby, instructions are executed in a low-level sequential manner closely matching the single-core architecture [110].

In fact, when supporting parallelism, there is a trade-off between the efficiency of the parallel program and the simplicity of writing it. Writing a parallel program in imperative languages such as C, C++, and Fortran with parallel libraries, can deliver an efficient program with high exploitation of parallelism. However, the programmer must still manage the low-level details of coordination aspects, such as process management, and explicitly manage communication and synchronisation. Consequently, the programmer's productivity is reduced,

the resulting program is complex, long, and more prone to errors. On the other hand, Functional languages (Section 2.4.5) provide higher level support for parallelism, better programmer productivity, and less code with the price of lower performance compared with parallelism support in imperative languages.

This section outlines the most common parallel programming libraries, such as MPI [125, 77], and PVM [71] as message passing models, which target distributed memory architectures. It also discusses OpenMP [36] as a mainstream model for shared memory architectures.

2.4.1.1 Message Passing

Message passing has emerged as a standard model for exploiting parallelism on large-scale homogeneous distributed memory architectures. In particular, it provides scalable, portable, and efficient parallel applications. Moreover, it supports different computational models; i.e. "Functional Parallelism", in which the main application is divided into different tasks, and can be parallelised by implementing those tasks in parallel. Another form of supported parallelism is "Data Parallelism", in which the same computation can be carried out in parallel on different data sets. The message passing model is supported by different languages, including C, C++, and Fortran. The message passing model, as opposed to shared memory model, can be used to program both shared and distributed memory parallel machines.

In the 1990s, a committee of vendors, government laboratories and universities worked together to produce a standard specification for developers and users of message passing libraries. At that time, many different message passing libraries often produced by computer vendors, existed. The result of their standardisation efforts was MPI [125, 77]. Since it is a standard interface, code written for one system can easily be ported to another system. MPI is the only message passing library that can be considered standard, and which lends itself to virtually all distributed memory programming models.

PVM [71] is an integrated set of software tools and libraries, which provides a unified framework for the development of parallel programs. PVM was an earlier

version of a portable library for message passing than MPI. However, nowadays MPI is more widely used, because it is supported by many computer vendors and has similar functionality to PVM.

Message passing models using PVM or MPI have gained acceptance as they represent mainstream parallel programming models and are likely to remain useful and widely used in the future, regardless of their low-level, and requirement for combinations of different parallel programming models to support heterogeneous or hierarchical parallel architectures. On the other hand, parallelising an application using MPI or PVM is not easy. The programmer must be involved in each aspect of the coordination, i.e. specifying how the work is to be divided among the PEs, balancing the work to be done and managing the PEs communications. Moreover, there is no method in MPI or PVM that can be employed to separate communication and computation elements of a code, as there is with other high-level parallel programming languages, such as GpH.

2.4.1.2 Shared Memory

Parallelism approaches, that is based on threads, like pthreads [32] or JavaThreads [128], represent the main model of parallel programming on shared memory architectures. Nonetheless, the explicit thread-based model is low-level, as the programmer is required to manage synchronisation and communication in a manner that sidesteps race conditions and ensures that deadlocks do not occur [98, 18]. Moreover, the use of fine-grained locking to manage synchronised access to shared memory can lead to limited scalability.

By contrast, OpenMP is a de facto standard Application Programming Interface (API) used mainly with shared memory architectures to provide a higher-level model that abstracts over the thread management details. OpenMP is not a new programming language, rather, it is a specification that can be added to some programming languages such as Fortran, C, and C++ to specify the coordination aspects of a parallel program [36, 140]. It consists of a set of compiler directives, supporting library routines, and environmental variables to specify parallelism, and program runtime characteristics [140]. The compiler directives inform the compiler which regions of the code require parallel implementation.

The main model of parallel programming supported by OpenMP is called the "Fork/Join model". The master thread will work sequentially until it encounters parallel directives, then a team of new threads will be prompted to fork off. This team then works in parallel until the end of the parallelised section, where it joins together so that the master thread can continue working as before, until it encounters another parallel directive, and so on.

OpenMP is very active community, with emerging standards for language independent shared memory computations. It has been successfully used to develop high performance parallel applications, as a consequence of its several advantages. First, it is simple to learn and use, requiring little programming effort. Moreover, it provides high performance applications that are able to run on different shared memory platforms with a different numbers of threads. Furthermore, as a result of its being a directive-based approach, the same code can be developed on single-core, as well as multi-cores platforms; in the former, the directives are simply considered to be comments and therefore ignored by the compiler and successfully implemented sequentially [36, 37]. Another important advantage of OpenMP is that it allows incremental parallelisation to be carried out. By starting with a sequential program, the programmer needs to simply add those directives which express parallelism [137, 37]. OpenMP is identified as a high-level parallel programming model, although it is explicit about threads. The parallelisation task is not always straightforward, as the programmer still needs to consider carefully how to exploit parallelism efficiently.

2.4.2 Parallel Object Oriented Programming

Object-orientated programming facilitates encapsulation of low-level mechanisms, and additionally clears a path for the programmer to manage the complexity of parallelism. Further, it enables modularity and code reuse, the importance of which is maximised for parallel programmes, as they have greater development overheads [88].

The object-oriented class of parallel programming supports a high-level approach to parallelism with different communication models. In particular, the Charm++ system [89] supports explicit parallelism with a message passing model,

and builds on top of C++ to provide an asynchronous message-driven orchestration, together with an adaptive runtime system. It has been used for numerous, large-scale, portable applications; for example, biomolecular simulations from the domain of molecular dynamics. In contrast, A Parallel Object-oriented Environment for Multi-computer Systems (POEMS) [88] provides implicit support for parallelism, by hiding the coordination of parallelism management and assign it to the underlying runtime execution model, which is based on object replication.

As an alternative to message passing, method invocation represents another communication mechanism adopted by POEMS, as well as another set of parallel object oriented programming libraries such as JavaParty [136], which extends Java by providing locality support for non-uniform parallel machines such as NUMA, as well as supporting the data-parallel pattern, ParoC++ [167], which extends C++ with the support of parallel objects, and adaptive utilisation of heterogeneous architectures resources, such as Grids.

2.4.3 Hybrid Parallel Programming Model

Current state-of-the-art practice in high-performance computing to exploit the increasingly hierarchical architectures, is using different programming models at different levels of the hierarchy, such as at the cluster and at the node (a multi-core) levels of a cluster of multi-cores. A typical hybrid model might be MPI message passing at the distributed memory (cluster) level, together with OpenMP on each shared memory (multi-core) level [174]. A more complex model offers support for massive data parallelism at the third level by using Compute Unified Device Architecture (CUDA) for example, to exploit GPUs cluster [175]. The complexity associated with managing two or more different parallel programming abstractions restricts this approach to parallel programming to areas of HPC.

An early attempt to unify programming models in this setting involved the implementation of an early definition of OpenMP on clusters of workstations [85], which was built on the TreadMarks distributed shared memory implementation [11]. More recently, ScaleMP [145] provided a commercial virtualisation solution, in the form of the vSMP infrastructure. It provides a "virtual symmetric multi-processing" over a cluster of multi-cores, through OpenMP or pthreads

as programming abstractions. Its focus is on memory intensive applications, rather than classical high-performance applications, and it is aimed at businesses rather than computing centres; reflecting the transition of parallel programming towards the mainstream. Using an InfiniBand network, performance measurements in [146] report a remote memory latency of 20 times local memory access. However, through the aggregation of several remote memory accesses on the application level, a bandwidth of 96GB/s was achieved, resulting in a good speedup of up to 80 on 104 cores. From a programming model perspective, problems with load balancing in OpenMP applications have been identified, underlining the importance of load balancing policies in virtual shared memory implementations.

The OpenCL and OmpSs frameworks are driven towards providing a unified model for multilevel parallel programming over a hierarchical architectures of cores and GPUs [171, 53, 96]. Regrettably, such a model is still lower-level than would be desired, which results in increasing the responsibility of the programmer to manage parallelism.

PGAS languages take a data-centric view. They provide primitives for mapping distributed data structures across nodes, and expresses computation at named locations as the main mechanism for controlling parallelism. PGAS languages provide a higher level of abstraction compared to OpenMP, in that they hide the details of the coordination between parallel activities, as well as managing communications automatically inside the RTS.

Prominent examples of the PGAS languages are Chapel [35], X10 [38], and Fortress [9]. PGAS concepts have been integrated into mainstream languages in the form of Unified Parallel C (UPC) [54] and Co-Array Fortran [127]. A library-based implementation of asynchronous PGAS, in the form of Global Futures [39] provides implicit synchronisation based on future abstraction.

Our work in GUMSMP represents an implementation for GpH, targeting a cluster of multi-cores architectures, where the model of parallelism is unified.

2.4.4 Parallel Systems

There are many parallel systems targeting different classes of parallel machines. An early parallel Lisp implementation was MulT [93], which introduced the notion

of lazy task creation [124]. This concept was also employed in the design of GHC-GUM. It allows threads to subsume the evaluation of the data, for which the potential parallelism has been generated.

The work on Lazy Threads [75] explores different ways for encoding and distributing potential parallelism. The representation of parallelism in GHC-GUM and GHC-SMP, in the form of sparks, represents one point in the spectrum, with a focus on low overheads.

These most closely related languages to our work on GUMSMP are all dialects of Haskell, and discussed in more detail in Sections 2.4.5.1, 2.4.5.2, and 2.5. There are other parallel systems with a shared design or implementation concerns to our work, and are discussed below.

2.4.4.1 Manticore

Manticore is a heterogeneous parallel implementation for ML and provides implicitly threaded parallelism using nested data-parallel constructs, drawn from NESL [25, 24] and Nepal [33, 34], which can be combined with concurrent and coarse-grained parallelism support of ML's [143] to provide explicit synchronisation and coordination, based on message passing on a large scale; thus supporting parallelism at multiple levels [59, 60]. Different degrees of parallelism are supported ranging from implicit parallelism: where the compiler and runtime system are responsible for managing parallelism to explicit threading, where the programmer has the entire control to manage parallelism. These mechanisms are constructed on top of a sequential language that is based on the features of functional programming. By supporting two levels of abstraction, the implementation is sufficiently flexible to support hierarchical networks. Futures and data-parallel constructs are key abstractions to manage local parallelism [59, 60].

2.4.4.2 Filaments

Filaments [111, 112] was an early system implemented as a portable package with a focus on light-weight threads, potentially combined with distributed shared memory, encouraging an approach to parallelisation that exposes massive amounts of parallelism, determining at runtime whether or not to exploit specific paral-

lelism, rather than to restrict it at application level.

This approach benefits from the advantages of fine-grain parallelism, such as its simplicity, efficient and dynamic load balancing, simplifying code generation, and portability, as it depends on the application and the problem size rather than on the executing cores. Filaments has been implemented as a package for shared memory multi-cores and termed Shared Filaments (SF) and used as a system-call library with performance within 10% of the equivalent hand-coded coarse-grained programs for several parallel applications. SF was then applied to the modified Sisal compiler as a back-end, to produce Filaments code and therefore achieve efficient parallelism in the functional data flow language [67].

This was then extended to Distributed Filaments (DF) to target a cluster of workstations by combining it with a distributed shared memory customised for use with fine-grain threads. It achieves efficient performance by providing automatic load balancing, overlapping communication with computation, and implementing a reliable and fast datagram communication protocol. For a variety of parallel applications, DF achieves good speedup on a cluster of workstations, e.g. a speedup of up to 5.7 on 8 cores [68].

A filament is a lightweight thread with size ranges from small, as in the computation of an average in a Jacobi iteration; medium size, as in the computation of an inner product in a matrix multiplication, or large with one process per core as in a coarse-grain program, with one to one correspondence between a process and a core. Server threads are created on each core to execute filaments one at a time. The Filaments system achieves its efficiency by employing a combination of key important techniques: Stateless threads; so there is no private stack for the threads, and threads are treated equally, so there is no pre-emption or context switches, but only one stack per server thread (core). It also provides small thread descriptors, therefore, allowing for more room for the program data in the cache; thus enhancing data locality. Moreover, it provides less contention, due to the use of local ready queues for all types of threads. More optimisations are offered such as inlining, pruning, and pattern recognition to reduce the different sources of overheads associated with creating and executing filaments [111, 112].

2.4.4.3 Task Parallel Library

Task Parallel Library (TPL) is a library for .NET framework that does not require language extension and provides efficient performance by taking advantage of potential parallelism in the program, e.g. a speedup between 5 to 7.5 on 8 cores [99]. Parallelism is expressed in terms of tasks, which represent the units of work, mapped to the worker threads. The key elements for defining custom control structures based on TPL, are parametric polymorphism (generic) and first class anonymous functions (delegate). All types of parallelism can be built based on those two elements, using just the two primitives provided by the library, i.e. task, and replicable task [99].

Work-stealing queue is the main data structure used for load balancing, where each thread has its own local task queue from which others can steal tasks. Since the performance of the work-stealing queue is critical to overall performance, an alternative duplicating queue has been implemented to trade memory with performance, as work can sometimes be duplicated to avoid locking [99]. Even though TPL provides automatic parallelism management in terms of mapping the parallel tasks to worker threads, its patterns of parallelism are considered to be a semi-explicit approach (Section 2.3.1) as the programmer is still required to manage the task decomposition and synchronisation [18].

2.4.5 Functional Languages

The following paragraphs are closely based on [166]. Functional languages as computation languages for parallelism have many attractive properties with regards to parallelism, which has gained worldwide recognition over the past three decades. They provide a high-level programming approach by trading some performance for ease of programming, thus increasing the programmer's productivity. They offer sophisticated methods of abstraction, in particular they feature an absence of side effects which simplifies parallel programming. The other primary benefit of a pure computation language is the referential transparency which ensures that the execution is afforded significant freedom of execution order without altering the semantics of the program. Specifically, independent expressions can be evaluated in any order and are guaranteed to deliver the same result. In the

case of parallel execution, this is also known as deterministic parallelism [116]. Therefore, the potential exists for the independent expressions to be evaluated in parallel.

For these reasons, such high-level, purely functional languages represent a good host for parallelism support in the form extended coordination (sub) languages, or standalone parallel functional languages. In line with the corresponding high-level pure computation language, the parallel functional languages offer high-level parallelism support in the form of automatic management of a number of coordination aspects. Therefore, the programmer is freed from explicit low-level management of parallelism. On the other hand, the high-level management of parallelism often reduces performance as it is less effective than hand-crafted coordination.

Examples of such languages include: Single Assignment C (SAC) [147], which has syntax based on C, but uses single assignment semantics, making it referentially transparent. It also utilises program transformations heavily to improve the efficiency of the data parallelism generated. It mainly targets numerical applications and achieves excellent speedups on the NAS benchmark suite. Another widely used parallel functional language is Erlang [172] which represents a high-level distributed memory parallel language with explicit message passing as a primary way of communication between the light-weight processes based on the "Actor Model" [84, 1], which is a model for concurrency, based on the usage of actors as lightweight processes, which differs from threads in its ability to exchange messages as well as not maintaining shared states. Erlang has been widely used to provide scalable industrial real-time systems. Other groups of languages follow the multi-paradigm programming which combines some features of functional languages with object oriented languages, such as the traditional Ocaml [153], or the new generation languages such as F# [158], and Scala [129]. There are substantial surveys of parallel functional languages in [166, 165].

Haskell as a purely functional language has been put in to use as a host computation language for a broad range of parallel languages. Beside the aforementioned characteristics of functional languages, Haskell belongs to the *lazy* (non-strict) sub-set of functional languages. The following discussion of lazy

evaluation is closely based on [166, 104]. Lazy evaluation aims to perform the least possible amount of reduction. That is evaluating an expression only when the computation needs its value. In contrast with the strict languages, in lazy languages the evaluation is demand-driven as the consumer of the evaluation result is in charge of the evaluation amount and order. Lazy evaluation of non-strict evaluation has both challenges and opportunities for parallel execution.

The main challenges are:

1. **Laziness vs. eagerness required for parallelism:** Lazy evaluation is sequential and is based on achieving the least possible amount of reduction. That is reducing the expression only when its value is needed. To the contrary, parallel programs perform computations on multiple PEs, meaning that eager evaluation is preferable. A balance between lazy and eager evaluation is needed to provide sufficient parallelism. In the context of speculative parallelism, laziness limits the amount of speculative parallelism, because the demand in the program determines the evaluation degree.

2. **Must specify the degree of evaluation:** In contrast to strict languages, the consumer of the evaluation result is in charge of the evaluation amount and order, which has to be specified by the programmer.

3. **Hard to build cost models:** In lazy languages, the behaviour of the computation is demand-driven and depends on the amount and order of evaluation required by the consumer which make it challenging to construct cost models. This means that reasoning about computational costs is more complicated than in a language with eager evaluation.

It could be considered as unusual to find a lazy language such as Haskell as a widely used functional computation language with many coordination sub languages e.g. par Monad, HdpH, GpH. Many characteristics making Haskell an appropriate host computation language for parallelism do not differ from those characteristics making it a good sequential language: referential transparency, advanced type system and high-level abstraction in addition to a range of other features. Moreover, the following opportunities provided by the lazy evaluation highly support high-level parallelism.

1. **On-demand reductions:** In a parallel setting, on demand reduction assures that only the required data items of a large data structure are communicated from the producer to the consumer. To tackle the piece-wise communication, the consumer can explicitly specify the evaluation degree of the required data structure.

2. **Clean separation of computations and coordination:** By detaching the evaluation mechanism from the language semantics, the programmer can separate the specification of the result value from operational behaviours of the program. This represents an important feature for parallel programming as evaluation can be separately specified without altering the original computation code. For example, in the parallel Haskell extension, GpH, this extra coordination is supported by evaluation strategies discussed in the following Section 2.4.5.1.

There is a diversity of languages and implementations for parallel Haskell. At the language level, diversity is based on the different abstractions supported. These vary in terms of how explicitly they control parallelism, e.g. implicit, semi-explicit, and fully explicit approaches. At the implementation level, diversity is based on different classes of architectures with different characteristics, e.g. clusters, multi-cores, etc. In the following sub-sections, we focus on parallel Haskell dialects that are closely related to our work. In particular, we discuss GpH, Eden, and HdpH as examples of parallel Haskell languages supporting semi-explicit approach. We then discuss Cloud Haskell and Par Monad as examples of parallel Haskell languages supporting explicit approach. The discussion of the different implementations for parallel Haskell is provided in Section 2.5.

2.4.5.1 Semi-explicit Parallelism

Glasgow Parallel Haskell (GpH): The following description outlines the main features of GpH and is summarised from [162, 119, 115, 164]. GpH is one of the many extensions of Haskell lazy functional language, which was developed to facilitate parallel programming, by supporting high-level parallelism. Since GpH is mostly implicit, the majority of the coordination aspects of the parallel program are managed by the RTS. These coordination aspects involve threads

and memory management, communications and synchronisation. In GpH, the programmer specifies the part of the program to be evaluated in parallel, leaving the decision to the RTS. In particular, the parallelism in GpH is achieved by adding two primitives to Haskell: `par` and `pseq` .

- The intuition of x `par` e is: evaluate e and return it but, at the same time, attempt to evaluate x (on a separate core). Therefore, `par` does not enforce the evaluation of x in parallel with e, but it is rather a request for parallelising e, if possible.

- The notation $e1$ `pseq` $e2$ is: evaluate $e1$ to the top level constructor of the data structure: Weak Head Normal Form (WHNF), and then return $e2$.

For example, x ‘`par`‘ y ‘`pseq`‘ (x + y)
In this code, x is sparked for parallel evaluation and the current thread evaluates y, and then adds the results of x and y. To complete the addition operation, both arguments must be evaluated. Typically, x and y are local variables, bound to a sizeable computations.

Depending only on using `par` and `pseq` can obscure the algorithm by engaging a lot of program text to describe the dynamic behaviour code. In order to separate the algorithm and the dynamic behaviour codes, Evaluation Strategies have been developed [162].

```
par  :: a -> b -> b   -- parallel composition
pseq :: a -> b -> b   -- sequential composition
```

Figure 2.6: GpH coordination primitives

Evaluation Strategies use lazy higher order functions to separate the computations (algorithm) from the coordination (dynamic behaviour), and therefore raise the abstraction level, to allow the parallel function to be developed in two separate sections: the algorithm and the strategy. This is a great advantage of GpH as compared to other low-level parallel libraries like MPI in which it is much more difficult to differentiate between the computation and coordination aspects of the program; since a huge amount of the program text is engaged in code, specifying the coordination aspects.

A strategy is a function responsible for specifying the dynamic behaviour needed by the algorithm to solve a problem in a parallel way, without modifying the computation code. It returns a nullary value (), since it is executed purely for effect. A strategy can be applied with "using" construct to apply a strategy to a value. It takes a value of specified type and a strategy of the same type and applies a strategy to the value, as illustrated in Figure 2.7.

```
type Strategy a = a -> () -- evaluation strategy
using :: a -> Strategy a -> a -- applying strategy to value
using x s = s x `seq` x
```

Figure 2.7: Evaluation strategies

Since strategies are functions, they can be combined together, passed as argument, or composed. A new strategy can be defined simply for specific applications. The evaluation degree strategies are presented in Figure 2.8, and defined as follows:

- **r0** : which involves on reduction at all, and can be used to exclude some elements from a list from reduction.
- **rwhnf** : reduces to Weak Head Normal Form (WHNF), which is the default in Haskell.
- **rnf** : fully evaluates its argument, which means reduces it to Normal Form (NF).

```
r0 :: Strategy a
r0 _ = ()

rwhnf :: Strategy a
rwhnf x = x `seq` ()

class NFData a where
  rnf :: Strategy a
  rnf = rwhnf
```

Figure 2.8: Evaluation degree strategies

"Strategies specifying data oriented parallelism describe the dynamic behaviour in terms of data structure" [162]. For example, parList is a function

that applies a strategy to each element in a list in parallel. Another example is parMap, which takes a function and a list of data, and maps the function over the list in parallel. More information about these strategies can be found in [162].

New Strategies: Evaluation strategies have been redesigned as shown in Figure 2.9 to provide new benefits, while preserving the original strategies features of modularity and compositionality [115]. The new benefits are:

1. Introducing evaluation-order monad to allow the specification and ordering of a set of evaluations in a compositional way, thereby providing a clearer, more efficient, and more generic specification of the parallel evaluation.

2. Resolving the space management issues for retaining heap unnecessarily with the original strategies. With the new formulation, sparks representing parallelism are preserved, while the heap associated with superfluous parallelism is reclaimed. Speculative parallelism is better supported with the new formulation, as unnecessary speculation will be pruned by the garbage collection.

3. The new formulation makes it possible to directly express the class of parallel coordination abstractions that cannot be expressed with the original strategies. Those parallel coordination abstractions are embedded within the lazy components of the data structure, and with the original strategies, they were defined as functions rather than expressed as strategies. Thereby, the new strategies produce more compositional strategies, and facilitate a richer set of parallelism combinators.

Eden: The following description of Eden is summarised from [109, 21]. Eden is a semi-explicit approach to functional parallel programming, which extends Haskell. Processes are explicitly defined in Eden, and the communication supported combines explicit and implicit models [109, 107]. Distributed memory parallelism is supported by Eden; thus, there are no shared values among the processes, as it supports the message passing model for communication (Section 2.4.1.1).

```
data Eval a = Done a

instance Monad Eval where
return x = Done x
Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

type Strategy a = a -> Eval a
using :: a -> Strategy a -> a
x `using` s = runEval (s x)

dot :: Strategy a -> Strategy a -> Strategy a
s2 `dot` s1 = s2 . runEval . s1

r0 :: Strategy a
r0 x = return x

rseq :: Strategy a
rseq x = x `pseq` return x

rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x

rpar :: Strategy a
rpar x = x `par` return x

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do x' <- s x

xs' <- evalList s xs
return (x':xs')

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar `dot` s)
```

Figure 2.9: New evaluation strategies

The programmer only has to specify the data on which the process depends, and the underlying runtime system automatically manages the other control issues, such as communication action and process placement [107]. The programmer is provided with some control over the load balancing, as well as the granularity in order to specify expressions to be evaluated as parallel processes. This can be achieved by using the high-level parallelism abstractions (libraries of skeletons) provided by Eden to simplify the task of parallelising a program substantially.

For coordination, process abstractions and process instantiations are provided in Eden as shown in Figure 2.10. The process abstraction: process ($\backslash x \rightarrow e$) of a predefined polymorphic type Process a b defines the behaviour of a process having the parameter x, with type a as input and the expression e with type b as output. When processes are instantiated, they are executed in parallel if resources are available.

```
-- process abstractions and instantiations
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # ) :: (Trans a, Trans b) => Process a b -> a -> b
```

Figure 2.10: Basic coordination constructs in Eden

A predefined infix instantiation operator (#) is used to create processes. When the expression: $e1$ # $e2$ is evaluated, a new process is created dynamically with its interconnected communication channels, to evaluate the application of $e1$ to $e2$. The evaluation of $e2$ is executed by the parent process (that owns the instantiation expression), and the resulted full normal form data is sent via an implicitly generated channel.

The child process evaluates the application of $e1$ to $e2$, and returns the result via another implicitly generated channel. After creating the processes, the only data objects communicated are fully evaluated, except for lists, which are transmitted element by element as a stream. Values are communicated automatically in Eden without a prior request from the receiver, until a notification is sent from the receiver, indicating that input values are no longer needed.

Common parallel evaluation patterns are abstracted by algorithmic skeletons into higher order functions, which simplify parallel program development, as well as abstracting over the coordination details. Eden supports a range of skeletons, such as the master-worker skeleton [21].

HdpH: The following description of HdpH is summarised from [113, 155]. A High-level Distributed Memory parallel Haskell in Haskell (HdpH) is a high-level, semi-explicit distributed memory parallel Haskell extension, influenced by Cloud Haskell, but providing higher-level coordination and targeting hierarchical architectures. Unlike Cloud Haskell, which is designed for distributed computing,

HdpH supports parallel computing and provides evaluation strategies and algorithmic skeletons influenced by GpH and Eden. Furthermore, it supports parallelism at two levels, shared memory level and distributed memory level. For shared memory parallelism support, it employs the primitives supported by Par Monad [116]. In particular, it uses a *fork* primitive to generate a parallel thread, and Ivar as a communication abstraction to communicate the computation results of the parallel tasks. For distributed memory parallelism, the *spark* primitive is used to generate parallel tasks that can be executed in a remote host, similar to GpH. Serialisation is achieved by extending the Cloud Haskell closure serialisation to support polymorphic closure transformations; thus offering high-level abstractions. Fault tolerance is implemented in HdpH as high-level skeletons-based on the supervised work pool approach.

```
-- Shared Memory Primitives (same as Par Monad)
data Par a -- par comp yielding an    a
fork :: Par () -> Par ()
data IVar a -- IVar expecting an    a
new :: Par (IVar a)
get :: IVar a -> Par a
put :: NFData a => IVar a -> a -> Par ()

-- Distributed Memory Primitives
data Closure a -- closure yielding an    a
spark :: Closure (Par ()) -> Par ()
data GIVar a -- global IVar expecting an    a
glob :: (NFData a, Binary a) => IVar a -> Par (GIVar a)
rput :: (NFData a, Binary a) => GIVar a -> a -> Par ()
```

Figure 2.11: Shared and distributed memory primitives for HdpH

2.4.5.2 Explicit Parallelism

Cloud Haskell: The following description of Cloud Haskell is summarised from [56, 55]. The term "Cloud" was used to indicate that the language targets a large number of distributed memory machines. Cloud Haskell is a domain-specific language for distributed memory systems, implemented entirely in Haskell. It emulates the Erlang style of explicit message passing, as a primary means of communication and a fault tolerance mechanism. Moreover, it provides a novel approach for function closures serialisation, to allow higher order functions to be

communicated in a distributed computing environment. It also provides additional features inherited from the use of Haskell, e.g. purity, strong type system, and monads. The programming model supported is the "Actor Model", similar to Erlang, where the inter-processes communication is achieved through explicit message passing, with the support of all types of messages. Processes are created on nodes which represent the machines. Within each process, the Haskell thread-based model of concurrency is still supported. Thus, there are three levels of abstraction: node, process, and thread.

To build an application with the Cloud Haskell, the programmer is provided with a set of primitives for processes management and monitoring, serialisation, and messages communication with hidden underlying implementation details. Examples of such primitives are presented in Figure 2.12. However, efficiently building an application with Cloud Haskell can sometimes involve a considerable learning curve [17].

```
 -- Basic messaging
instance Monad ProcessM
instance MonadIO ProcessM
send :: Serializable a => ProcessId -> a -> ProcessM ()
expect :: Serializable a => ProcessM a

-- Process management
spawn :: NodeId -> Closure (ProcessM ()) -> ProcessM ProcessId
call  :: Serializable a => NodeId -> Closure (ProcessM a) ->
   ProcessM a
terminate :: ProcessM a
getSelfPid :: ProcessM ProcessId
getSelfNode :: ProcessM NodeId

--Process monitoring
linkProcess :: ProcessId => ProcessM ()
monitorProcess :: ProcessId -> ProcessId -> MonitorAction ->
   ProcessM ()
```

Figure 2.12: Some of the interface functions of Cloud Haskell

Par Monad: The following description of Par Monad is summarised from [116]. It is a parallel Haskell programming model for pure deterministic parallel computations, providing explicit monadic control of concurrency, and it targets a single shared memory multi-core machine. As demonstrated in the shared memory

primitives for HdpH in Figure 2.11, *Par* is used to introduce parallelism based on the use of a *fork* primitive to generate parallel tasks. Deterministic result in the par Monad is produced by the *runPar*, a mechanism that is combined later with Cloud Haskell in HdpH, to support distributed memory parallelism (Section 2.4.5.1).

There are other parallel Haskell languages such as Data Parallel Haskell (DPH) [135], which adopts key insights, and evolved out of earlier work on Nepal [33, 34], which in itself was heavily influenced by the NESL [25, 24] system, to support nested data parallelism. With the focus on data-parallel applications, hierarchical networks are less of a concern in its design, as it focuses on utilising a single multi-core. In fact, an important aspect of the DPH implementation is flattening transformations, and distributing equal workload to processing units. This aims to bring parallelism over nested data structures into a flat format, so that manipulating only flat arrays, which can be more efficiently exploited by massively parallel hardware, opens up a wider range of applications including sparse or irregular problems.

2.5 Parallel Haskell Implementations (RTS)

Parallel Haskell implementations are classified as distributed memory or shared memory implementations.

2.5.1 Distributed Memory Implementation

GpH on GHC-GUM: A Graph Reduction for a Unified Machine Model (GHC-GUM) [163, 164] is the virtual machine for the parallel Haskell functional language, which has been released as an extension to the GHC [134, 118]. It is based on the parallel reduction of the graph representing the program, the parallelism being exploited by the reduction of independent sub-graphs being carried out in parallel [133]. Some parallel/distributed Haskell extensions such as GpH, and GdH, use GHC-GUM as the core of their implementation. GHC-GUM is portable and available on different architectures (shared memory, distributed memory, or network of workstations). It is a message-based system, implemented

by different generic communication libraries: PVM, MPI, and MPICH-G (an implementation of MPI on top of the Globus grid middle ware) [66]. We provide a detailed description of GHC-GUM, and its components in Section 3.3.

Eden: The details of the Eden implementation are summarised from [109]. The main feature of Eden is its support for the distributed heap; it is strict regarding transference of data to the core and the management of this transfer is organised into three tables.

The Eden implementation extends GHC [134, 118] functionality by defining eight primitive operations, as shown in Figure 2.13, for explicit remote task creation and channel-based communication mechanisms. The language level constructs are implemented as Eden Module over Eden-specific primitive operation.

```
1. createProcess# request process instantiation on another core
2. createDC#      create communication channel
3. setChan#       connect communication channel in the proper way
4. sendHead#      send head element of a list on a communication channel
5. sendVal#       send single value on a communication channel
6. noPE#          determine number of processing elements in current setup
7. selfPE#        determine own core identifier
8. merge#         nondeterministic merge of a list of outputs into a single
                  input
```

Figure 2.13: Primitive operations for Eden

As a result of using primitive operations and Eden Module, no changes are applied to the GHC's front-end and major modifications concern alterations in the RTS and the compiler's back-end.

Greater flexibility and sustainability are achieved in Eden through its layered implementation, as shown in Figure 2.14; therefore, runtime system aspects are lifted into the Eden Module, i.e. defining basic work-flows as a high-level of abstraction.

Eden's run time system is an implementation of the Distributed Eden Abstract Machine (DREAM). It is an instance of the extended STG-machine, and executes each Eden process, which has one or more concurrent thread independently evaluating different output expressions. These concurrent threads share the input to the process, as well as the information on the heap and the inport table. Each thread is represented in the heap as a TSO (thread state object).

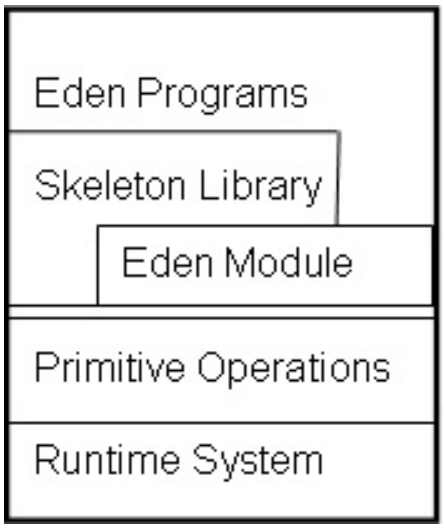

Figure 2.14: Layer structure of the Eden system [109]

Inport and outport represent the channels' end on the receiver side and on the sender side respectively. Any thread demanding unavailable data will be blocked on a Queue-Me closure.

When data is sent, it is known to the receiving inport, the sending thread (referred to by its outport) and the location of the incoming data, as specified by the inport. The result of each thread's computation is sent through its own outport.

The inport table and the outport tables are used to maintain the necessary information, such as mapping inport ids to Queue-Me closures addresses in the heap, and mapping outport ids to the destination outports respectively.

Multiple Eden processes are evaluated in the same PE in an interleaved manner and, in order to reduce a process creation overhead, each PE runs one instance of DREAM, to execute several Eden processes concurrently.

One of the PEs is nominated as the Main PE and starts the execution by evaluating the expression "main". A scheduler and the runtime tables are shared among all the Eden processes executing in the same PE (one inport table, one outport table and one process table).

- Inport table is used to map the unique identifier of the inport to the heap addresses of the corresponding Queue-Me closures, and to the global references to the connected outports.

- Outport table is used to maintain the mapping of outports identifiers to

the corresponding address of the thread state object. It is used for garbage collection and other system management purposes.

- Process table provides the number of inports and the number of threads for each process, which is equal to the number of outports.

Cloud Haskell: The details of the Cloud Haskell implementation are summarised from [56, 55]. Cloud Haskell was implemented entirely at the Haskell level, and tested with a recent version of GHC. Processes represent the basic units of concurrency, and are implemented using the Concurrent Haskell's threads supported by the standard GHC system, which offers low creation and termination overheads as a single node supports hundreds of processes, which may start and end frequently. Any process can send and receive messages which are asynchronous, reliable, and buffered. Each process maintains a process identifier, comprising the host name, TCP port, and a unique process number.

To exchange a message, the sending process connects to the given port, identifies the target process, and sends the serialised message. At the other side, when the message is received, it is placed into the message queue belonging to the destination process. The implementation of message queues was based on the Haskell's Software Transactional Memory (STM) which offers a mechanism to receive atomic transactions on individual message queues.

HdpH: The details of HdpH implementation are summarised from [113]. HdpH is implemented entirely in concurrent Haskell as a library on top of the standard GHC system. HdpH implementation is layered and moduler coded in Vanilla GHC Concurrent Haskell with independent modules for different coordination aspects, e.g. thread management, communication, scheduling, global references, spark management etc.; thus it preserves maintainability and facilitates development. The communication layer abstracts over the inter-process communication, which is based on MPI. The management of sparks in HdpH follows a similar work stealing approach of GHC-GUM, but at the Haskell level. In particular, each PE maintains a single spark pool, where sparked computations that are suitable for remote execution are placed. Within a PE, if there is no thread in

the runnable threads pool, then a spark will be turned to local threads. Sparks are sent for remote executions upon receipt of a work request message. When the spark pool is running low, a work request message is sent to a random PE, which is forwarded if no spark is available. A No-work message is returned to the originator PE if no sparks are available in the visited PEs, which delays for sometimes before sending another work request. The selection of a spark is age-based: the youngest sparks are turned to threads and the oldest ones are exported for remote executions. A Push message can also be sent which require an immediate execution of the computation and is suitable for very short and urgent actions, like writing to an IVar or forking a thread.

A global references mechanism is used in HdpH to access remotely hosted objects, with a registry table maintained to keep a link between the global references and the referred objects, much like the Global Indirection Table (GIT) in GHC-GUM (Section 3.6.1).

2.5.2 Shared Memory Implementation

GpH on GHC-SMP: It is an optimised shared memory implementation for GpH integrated in GHC [119]. It assumes a physical shared memory, and uses mutexes for synchronisation between local threads. GHC-SMP excels at the efficient handling of lightweight threads. Millions of lightweight threads are supported by the GHC-SMP runtime system. To achieve this, the threads are multiplexed onto a handful of operating system threads, approximately one for each physical core. We provide a detailed description of GHC-SMP in Section 3.3.

Par Monad: The details of Par Monad implementation are summarised from [116]. It provides implementation of the system-level functionality (work-stealing scheduler) as a Haskell library. It also separates the implementation of Par Monad, and the scheduler, thus, facilitates the modification of the implementation, such as implementing different scheduling policies. Work-stealing scheduling is implemented, where one worker thread is created for each physical core. Each Haskell Execution Context (HEC) has its own work pool, to keep the runnable threads, and runs its own instance of the scheduler, which aims to run a thread

from its own work pool, if no thread is available, it steals threads from others work pools. If no thread is available, then the worker thread becomes idle, and will be woken by another worker thread whenever the latter has new runnable threads available.

There are other parallel Haskell implementations, such as Meta-Par [62], which is built on the Par Monad, takes Haskell-level programmability and abstractions of RTS functionality further, especially for scheduling on heterogeneous architectures (with a focus on CPU and GPUs interaction).

2.5.3 Parallel Haskell Implementations Comparison

Many of the parallel Haskell language implementations depend on sophisticated runtime systems implemented in a low-level language, to automatically manage parallelism, i.e. synchronisation, communications, work scheduling etc. Examples include GHC-GUM, GHC-SMP, and Dream/EDI. The implementation of GUMSMP follows this approach.

Having a runtime system implemented in a low-level language gives high performance, but maintenance is challenging and the RTS needs to be continuously updated.

The main difference between Eden, GHC-GUM, and GHC-SMP are:

1. Heap Model: Completely independent sub-heaps are maintained in Eden, whilst a virtual shared heap is maintained in GHC-GUM, and a shared heap is maintained in GHC-SMP.

2. Granularity Control: Process creation is mandatory in Eden, whilst other implementations support dynamic mechanisms, such as sparking, speculative or optional thread creation and thread subsumption.

3. Work Distribution: Work distribution is Eager in Eden, as the process is moved to other PEs; whilst in GHC-GUM and GHC-SMP, the work distribution is lazy, in which case idle PEs steal work.

4. Eden and GHC-GUM implementations share the advantages of supporting a distributed heap, such as reducing synchronisation overheads.

The current trend for parallel Haskells is to use a concurrent Haskell to implement all functionality, instead of modifying the GHC runtime system, thereby trading performance for maintainability and ease of development. Examples include Cloud Haskell, Par Monad and HdpH.

Table 2.1: Parallel Haskell implementations comparison (+:property is supported, −:property is not supported, ++:property is improved)

	Distributed Memory					Shared Memory	
Property / Language	GUMSMP	GHC-GUM	Eden	Cloud Haskell	HdpH	GHC-SMP	Par Monad
Scalable (distributed memory)	+	+	+	+	+	-	-
Fault Tolerance (isolated heaps)	-	-	+	+	+		
Polymorphic Closures	+	+	+	-	+	+	+
Pure, i.e. Non-Monadic API	+	+	+	-	-	+	-
Determinism	+	+	-	-	-	+	+
Implicit Task Placement	++	+	+	-	+	+	+
Automatic Load Balancing	++	+	+	-	+	+	+
Hierarchical Load Balancing	+	-	-	-	+	-	-

Table 2.1 has been reproduced from [113] which compares the key features provided by different parallel Haskells. Those features are as follows:

Scalability: is the ability to exploit the increasingly available hardware resources, e.g. the number of cores.

Fault Tolerance: represents the implementation where heaps are isolated for each PE, and provides a tolerance mechanism for individual PE failure.

Polymorphic Closures: is the ones that can have more than one type.

Pure: has no side effects.

Determinism: a model that guarantees that the same sequential execution result is produced by the parallel evaluation.

Implicit Task Placement: the mapping of the parallel tasks to the PE is achieved implicitly by the language implementation.

Automatic Load Balancing: the balancing of parallel tasks among available PEs.

Hierarchical Load Balancing: the implementation that provides different load balancing mechanisms at different levels of the hierarchy e.g. shared memory load balancing mechanism within a shared memory machines, and distributed memory load balancing mechanism across the distributed memory machines.

Shared memory implementations have limited scalability, as they only work on a single shared memory machine, e.g. a multi-core or NUMA. On the other hand, distributed memory implementations work on shared memory architectures, as well as on distributed memory architectures. They can still deliver good performance on multi-cores, as long as the tasks to be communicated are large and the communication rate is low [14, 23, 22].

As we shall see in the following chapters, GUMSMP is also scalable, and provides improvements for two aspects of the parallel implementations, namely implicit task placement, and automatic load balancing. This is a result of integrating GHC-SMP, which is designed for multi-core architectures and GHC-GUM which is designed for clusters. Thus, GUMSMP is designed to provide an architecture-aware system, tuned for a cluster of multi-cores architectures.

2.6 Load Balancing

To take advantage of parallel architectures, work must be decomposed into smaller tasks to be carried out in parallel. The tasks must be assigned to PEs to approximately equalise computational load on PEs. To achieve good parallel performance, it is crucial to efficiently exploit the resources accessible as well as minimise the overhead of communication and preserve the data locality [18].

Load balancing (i.e. the distribution of tasks among the available PEs) has previously been widely researched as it is a central mechanism affecting the performance of parallel applications. Achieving scalable performance on large clusters of multi-cores, especially for applications with irregular or dynamically generated parallelism is challenging as they require dynamic load balancing techniques to efficiently exploit the underlying large numbers of cores [50]. It is relatively difficult to map the tasks to the PEs in applications that dynamically produce irregular

parallelism. Therefore, heuristics or approximations mechanisms are frequently employed to dynamically balance the load during the program execution. Information in relation to the task dependencies and communication patterns may be used to guarantee better data locality. Nonetheless, load imbalance could be the consequence of retaining the data locally, subsequently leading to a trade-off [18].

Individual task pools are maintained by each PE to hold the locally generated tasks. In some shared memory systems, PEs can access a single shared task pool [87]. There are two primary categories of load balancing: work stealing and work pushing.

1. **Work-Pushing** (active load distribution, load distribution commenced by the sender) is a mechanism where PEs with a large number of tasks push them to other PEs without receiving explicit work requests. Otherwise, if there is a single shared task pool and new task joins it, it is distributed to some PE.

2. **Work-Stealing** (passive load distribution, load distribution commenced by the receiver) is a mechanism where idle PEs explicitly request work to execute. If there is a shared task pool, idle PEs steal tasks from that pool.

Each of the categories has advantages and disadvantages. To be able to distribute a task between PEs, work-pushing simply necessitates that there is contact from the first PE to the second PE, for example as a message with an attached task. In contrast, the second (idle) PE in the work-stealing mechanism has to actively contact the first PE, for example with a work-request message, and then the first PE has to transfer a representation of the work to be executed to the second PE. While work-pushing encourages even load balancing, it might negatively affect the data locality as a PE which offloads work may not be aware of the load on the receiver PEs, which might affect the data locality by the unnecessary movement of tasks.

The primary benefit of work-stealing is that work is only transferred if it is known to be needed, and the positive effect on data locality as it avoids overloading PEs with tasks. Nonetheless, work stealing can result in unbalanced load

distribution in situations where only a small number of PEs are generating large quantities of parallel tasks.

In both categories it is perceived as better to retain load information pertinent to other PEs. However, gathering this information could necessitate substantial communication, meaning that all machines must strike some sort of balance between the conflicting objectives of an even load balance and a low degree of communication. Consequently, lots of implementations employ random work allocation.

The notion of work-stealing first introduced by Burton and Sleep [31], and later by Halstead's application of Multilisp as a prominent system using work-stealing [78]. Following this, the randomised work stealing approach was investigated by Rudolph et al. [144] on shared memory parallel machines, and by Karp et al. [90] on distributed memory parallel architectures. Today, many runtime systems apply variants of the work stealing mechanism for the execution of parallel application on both shared memory and distributed memory parallel architectures. In OpenMP, load distribution is achieved at the level of loop by assigning different iterations to PEs. The majority of research on OpenMP is centred on shared memory machines, but more contemporary research concentrates on the issues of optimizing OpenMP programs for distributed memory machines [50]. HPC languages, like X10 [38], Chapel [35], and Fortress [9], and PGAS languages, like Titanium [177] and UPC [168] provide parallelism based on extending the OpenMP's parallel constructs which are centred on scaling to large distributed memory machines [50].

Cilk [26, 70] is a parallel programming language based on extending C/C++ with parallelism support. Its runtime system adopts the widely studied load balancing mechanism in the form of dynamic work-stealing for fully strict computations and provides scalability on shared memory machines, on network of work station as well as for wide area networks [27, 170].

A range of persistence-based load balancing algorithms are supported by Charm++ [89]. The standard mechanism incorporates monitoring load imbalance by collecting data for objects, calculating the degree of imbalance, and performing suitable rebalancing algorithms in either a centralized or hierarchical

manner [178].

Using PGAS and Remote Direct Memory Access (RDMA), Dinan et al. [50] achieved scalable work stealing mechanism to 8192 cores. In subsequent research, a hierarchical method known as retentive work-stealing was utilised to scale work-stealing to 150K+ cores by taking advantage of the principle of persistence-based rebalancing to meliorate the imbalance issue in scheduling task-based applications [102].

In the context of GpH (Section: 2.4.5.1), the distributed and shared memory runtime systems GHC-GUM (Section: 2.5.1) and GHC-SMP (Section: 2.5.2) employ randomised work-stealing as a default load balancing mechanism where the representation of work is based on graph structure with step-wise evaluation to support laziness. A detailed discussion of load balancing for GpH is provided in Section 3.5.

Chapter 3

GUMSMP Design and Implementation

3.1 Introduction

This chapter focuses on the design of a sophisticated Runtime System (RTS) to support the high level parallel coordination in GpH. In particular we develop a new implementation, GUMSMP that combines distributed and shared memory capabilities. This chapter explores the GUMSMP design space, and presents the GUMSMP design and implementation [6].

The chapter is structured as follows. Section 3.2 presents the design objectives. The main components of parallel Haskell runtime systems (GHC-GUM, GHC-SMP, and GUMSMP) are outlined in Section 3.3. Section 3.4 presents the thread management component, which is shared between the three implementations. Section 3.5 presents the new GUMSMP load distribution mechanism, highlighting the differences between the different implementations in terms of load balancing, as well as discussing our main contributions for improving the GUMSMP load balancing mechanism to support hierarchical architectures. Section 3.7 presents the GHC-GUM communication adopted by GUMSMP. Section 3.6 discusses the memory management components, explaining how each implementation performs garbage collection (GC).

3.2 Design Objectives

The main design objectives for GUMSMP are:

- *Asymmetric load distribution:* While striving for an even load balance overall, we employ different load distribution policies at the inter-node (across the cluster) and intra-node (within a multi-core node) levels, thus creating an asymmetric load balancing design. At the inter-node level (where communication is expensive), we accept a significant imbalance. At the intra-node level (where communication is cheap), we aim to optimise for an even load balance, enabling the GHC-SMP's work pushing mechanism. Compared to GHC-GUM, which was designed for flat networks, the load distribution mechanism in GUMSMP is more aggressive, accounting for the availability of several cores in each multi-core node. We implement the hierarchical, combined load balancing mechanism in GUMSMP (Section 3.5.3).

- *Gateway routing and distribution*: In the design, one core acts as a *gateway* to the remainder of the multi-core node. It is responsible for communication, and collects information concerning the load on the remote nodes. The advantage of this design is that only one core is responsible for the additional cost of maintaining a (partial) picture of the load across the network. The disadvantage of this design is that this core may then become a bottleneck for higher core numbers. We implement the concept of gateway routing and distribution in GUMSMP (Section 3.5.3.1).

- *Mostly passive load distribution:* We refine the passive work distribution policy between multi-core nodes, where work is only sent remotely when requested (work-stealing), by developing a hierarchy-aware load distribution achieved by enabling a node to *pre-fetch* work, utilising a *low-watermark* mechanism on the spark pool (Section 3.5.4).

- *Effective latency hiding:* The system must be able to overlap the inter-node communication with useful computation. Thus, remote data lookup is implemented as a split-phase operation, with implicit synchronisation. We em-

ploy this as a proven technology for large-scale, parallel systems [42]. Moreover, the combination of the previous implemented techniques in GUMSMP contribute to the efficient latency hiding.

GUMSMP, like the two other parallel Haskell implementations (GHC-SMP and GHC-GUM, presented in Sections 2.5.1, and 2.5.2 respectively), is based on the parallel graph reduction model, whereby the program to be executed in parallel is represented as a graph structure on physically distributed, but virtually shared memory. Parallelism is exploited by reducing the independent program graphs in parallel, where graph segments are communicated at two levels, using different, tailored technologies on the small-scale, at the physical shared memory level (multi-cores), and also on the large-scale, at the distributed memory level (clusters).

The design is a progression of the successful GHC-GUM and GHC-SMP technologies that already exist at both levels. In particular, our design employs a mechanism of work-stealing for passive load distribution, combined with an adaptive, dynamic mechanism for automatically distributing work and data on a cluster (between the nodes of multi-cores), where communication is based on the GHC-GUM communication model of explicit message passing. Within the node (a multi-core), a combination of passive and active load distributions are employed, whereby communication between cores is carried out as direct access to the shared memory within the node. Technically, we achieve this design by integrating the functionalities of existing implementations: GHC-SMP (within the node) and GHC-GUM (across the nodes) of the RTS for GHC, as indicated in Figure 3.1.

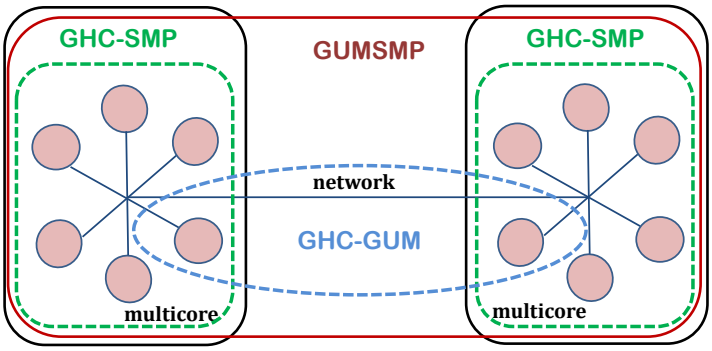

Figure 3.1: Matching parallel Haskell implementations and architectures

3.3 Main Components for Parallel Haskell Implementations

To describe the design and implementation of GUMSMP, we identify the main components of GUMSMP shown in Figure 3.2. The structure of these components is inherited from GHC-GUM. In the following sections we describe how these components are implemented in each system. The main components are:

1. *Thread Management:* is responsible for deciding when to generate a new thread and for determining how to schedule threads (Section 3.4).

2. *Load Balancing:* is responsible for distributing the load in parallel systems so that the idle time for core is reduced (Section 3.5).

3. *Memory Management:* is responsible for controlling access to local and remote data (Section 3.6).

4. *Communication:* is responsible for transferring data and work between the nodes (Section 3.7).

In the following discussion of the details of these components, we present how they are implemented in our new runtime system GUMSMP, as well as in the existing runtime systems GHC-GUM, and GHC-SMP.

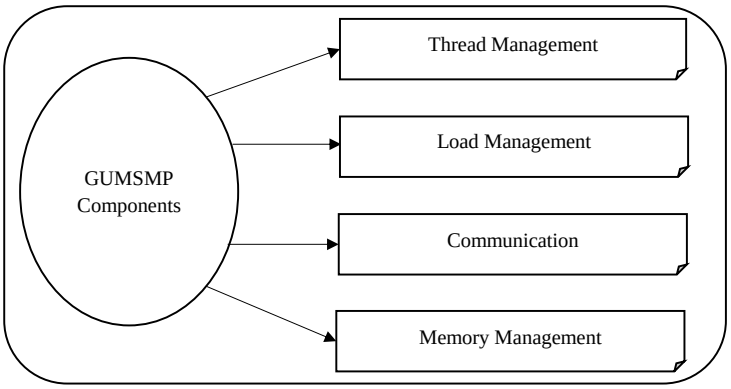

Figure 3.2: GUMSMP system components

The implementation of GUMSMP *shares* the same **thread management** mechanism with GHC-GUM and GHC-SMP. In particular, it manages lists of

runnable tasks as FIFO queues. It *combines* the **memory management**, **communication**, and **load balancing** components of both systems. In particular, it implements a virtual shared heap across all nodes, using PVM communication, and creates local worker threads for each node, which correspond to the number of cores within the node. Moreover, it provides hierarchical **load balancing** mechanisms with different levels for inter-node and intra-node.

3.4 Thread Management

The thread management model used in all the three parallel Haskell implementations (GHC-SMP, GHC-GUM, and GUMSMP) is called the "evaluate and die" thread management model, and was originally developed for the GRIP (Parallel Graph Reduction Machine) [133] in which potential parallelism is represented as `sparks` (pointers to unevaluated graph structures called `thunks`). Spark generation is cheap, simply adding a pointer to a thunk. This is essential to reduce the parallelism creation overhead, and the communication cost of sharing sparks locally, or over the network.

A spark indicates that it might be useful to evaluate a thunk in parallel, but this does not mean that it is necessary to do so, i.e. the parallelism is advisory rather than mandatory. If sparks become too numerous, they may be discarded by the RTS. Sparks are managed in the "spark pool" (a FIFO lock-free queue in the RTS), and generated by the `par` primitive.

An important concept of the "evaluate and die" model is a form of auto in-lining: if the sparked expression has not been evaluated by another thread when its value is needed, then the current thread will evaluate it as part of the computation. This behaviour is called "thread subsumption", because the potential parallel work is in-lined by the parent thread. As a result, the spark has fizzled, making it useless and discarded by the garbage collector. This behaviour helps increase the size of a thread; i.e. its "granularity" by delaying decisions regarding whether it should be generated. This is similar to the independently developed lazy task creation model [124].

In principle, there are three cases when the value of a spark is acquired:

1. If the expression has been evaluated previously, its value is returned directly.

2. If the expression is under evaluation either by local, or remote thread, the current thread will block (Section 3.4.2).

3. If another thread has not yet evaluated the expression, then the demanding thread will execute the computation itself.

3.4.1 Data Structures

The main component of GUMSMP is the PE, where the collection of communicating PEs implements the virtual machine. Within each PE, a set of operating system (OS) threads (worker threads, one worker thread per core) execute the Haskell threads. One Haskell Execution Context (HEC) is maintained for each core. The HEC is the data structure, where the data required by an OS worker thread in order to execute Haskell threads is contained. Figure 3.3 shows the different levels of abstraction over the hardware, supported by GUMSMP as follows:

HW Level: a core represents an independent hardware unit for computation, and a node represents a single multi-core machine.

OS Level: a single OS thread is associated with each HW level core.

RTS Level: a HEC is executed by the OS thread, and runs Haskell threads. A PE refers to a group of HECs within the multi-core.

As illustrated in the figure, different multi-cores are connected across the network. Within each multi-core, there is 1:1 relationship between each core and the OS thread, which in turn executes one HEC. Each HEC can then execute multiple Haskell threads, which are light-weight threads internal to and managed by the GUMSMP RTS, and not to be confused with the heavy-weight OS threads.

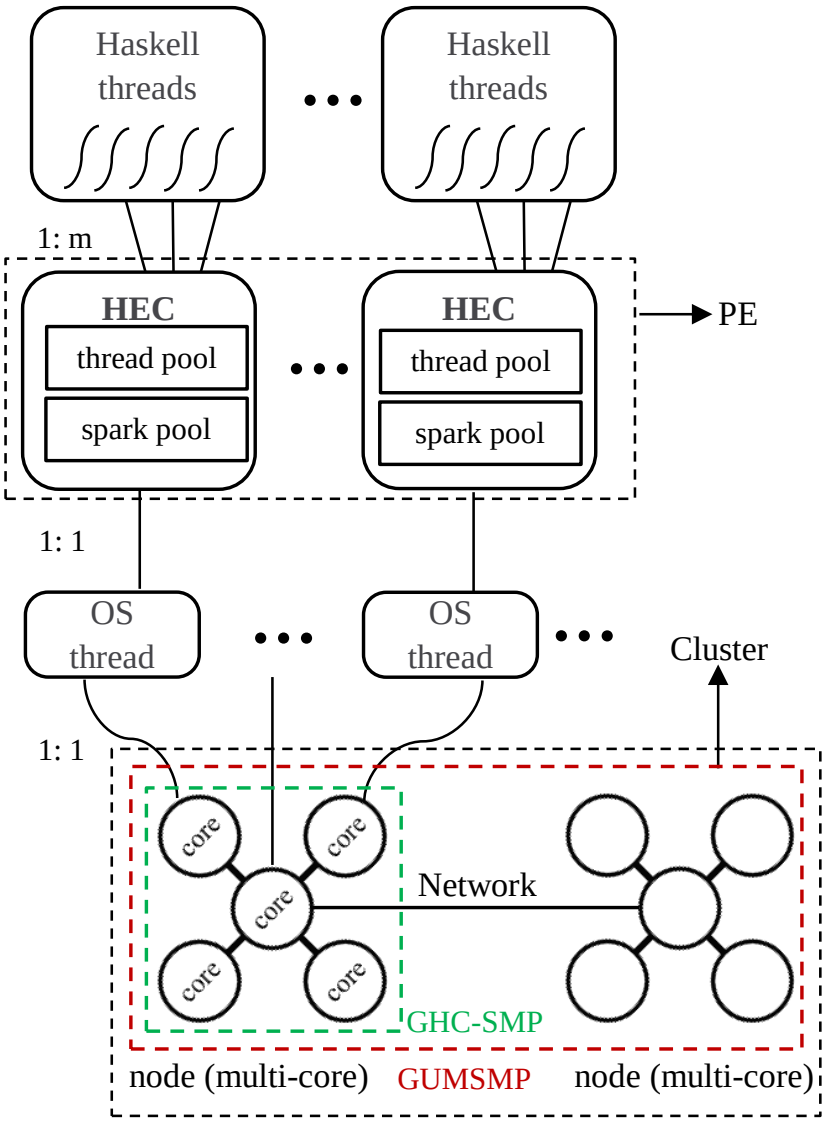

Figure 3.3: Levels of abstractions over the HW supported by GUMSMP

Two main work pools are maintained for each HEC to manage parallelism:

1. A spark pool is a flat pool organised in a FIFO queue, where the RTS maintains *sparks*, representing potential parallelism.

2. A pool of runnable *threads* is organised in a FIFO queue, representing parallelism and are executed by a HEC.

In all three implementations, the thread is an internal representation of a computation, and represented by a heap allocated TSO (Thread State Object), which encompasses the Haskell thread's state, especially its logical registers, together with its stack, where it runs, and points to the thread's SO (Stack Object). The

thread's stack grows dynamically until it attains a tunable maximum stack size. The HEC is considered idle when there are no threads remaining to be run in its thread pool.

The overhead of managing the thread pool is significantly higher than that for spark pool management. The reason for this is that additional information is necessary for a thread, such as a live thread priority, which is essential if more flexible scheduling is to be achieved.

When the HEC has no more work to do, it seeks out local work in the runnable queue, then searches for a spark in its spark pool, and then in other local HECs spark pools (as in GHC-SMP and GUMSMP). If a spark is found, this is turned into a thread by creating a TSO and SO, and the HEC begins to evaluate it. Otherwise, it searches for remote work looking for a spark in a remote PE (as in GHC-GUM and GUMSMP), and hence an independent thread might execute the sparked expression.

3.4.2 Synchronisation

Shared closures (nodes in the graph structure) can be either normal-form closures representing data, or thunks, representing work. In order to prevent two Haskell threads from evaluating the same thunk simultaneously, thereby duplicating work, the thunk has to be locked when the thread commences evaluation. However, locking is costly, and in addition can increase run time by around 50% as the measurements in [80] show. The mechanism implemented to achieve this synchronisation is called "blackholing". In reality, two main variants of the blackholing mechanism are available, balancing the overhead reduction against the risk of work duplication [80, 119]:

1. **EagerBH:** In this mechanism, when thread A enters a thunk in order to evaluate it, it immediately overwrites it with a Black Hole (BH). If another thread sees the BH, it will block until the previous thread finishes the evaluation. The possibility of a second thread initiating a duplicate evaluation is only a matter of a few instructions away. The cost involved is the extra memory stored in every thunk, as compared with sequential execution.

2. **LazyBH:** This uses the scheme described in Harris et al. [80] which requires a thread to walk its stack each time it returns to the scheduler, and then claim each of the thunks evaluated using an atomic instruction. If a thread is observed evaluating a thunk that has previously been claimed by another thread, the current execution is suspended and the thread is put to sleep until completion of the evaluation. Every thread returns to the scheduler at regular intervals (for example, to carry out garbage collection). This means that the same thunk cannot be evaluated continually in multiple threads indefinitely. As most thunks are entered, evaluated, and updated during a single scheduler time-slice, the locking overhead is considerably less than that which would occur when locking every thunk.

In practice, the difference in performance between EagerBH and LazyBH is minor, as LazyBH often catches all possible duplications except for thunks that are very shortly lived.

With the blackholing mechanism, when another thread tries to evaluate a thunk and finds a BH, it will block on the BH and attach its TSO to the BH pool (a global pool consists of a set of threads that are blocked on BHs). When the thunk evaluation has been completed, the thunk will be updated with its value, and all the threads blocked on this closure will be awakened and moved to the runnable pool, as part of the main scheduler loop (Figure 3.4).

For GHC-GUM, and GUMSMP, a thread might block on three closure types:

1. BH: a closure that is under local evaluation (will be re-awakened when local data becomes available).

2. FetchMe: global indirection to a closure under evaluation in a remote PE (will be re-awakened when remote data arrives).

3. RBH (Revertible Black Hole): a closure sent remotely for evaluation, but for which no location on the remote PE has been received (will be re-awakened when in-transit data has arrived).

3.4.3 Main Scheduling Loop

The management of threads and spark pools is achieved using FIFO queues. The core of each PE's execution is the following scheduling loop, which is executed by each HEC within the PE, with communication work restricted to the gateway HEC. The scheduling loop is executed until a FINISH message is received.

During the program execution, threads may take the form of any of the following states:

- **Running:** executed by a HEC.
- **Runnable:** waiting to be scheduled on a HEC.
- **Blocked:** waiting for data generated by another local or remote thread to complete.
- **Fetching:** waiting for data to arrive from a remote PE.
- **GC:** performing garbage collection.

```
1  while True do
2      switch sched_ state do
3          case SCHED_ RUNNING
4              break;
5          case SCHED_ INTERRUPTING
6              performGC ;
7              shut_ down;
8          case SCHED_ SHUTTING_ DOWN
9              Exit;
10     endsw
11     ScheduleCheckBlackHole(hec); //traverse the BH queue and
       wake up any tso that has data available
12     ScheduleFindWork(hec);
13     if messageArrived then
14         processMessages(hec);
15     end
16     schedulePushWork(hec); //if we have extra work and there
       are idle hecs, then push work to their pools
17     if emptyRunQueue(hec) then
18         continue ; //look again for work
19     end
20     tso = popRunQueue(hec);
21     result = stgRun(tso); //perform graph reduction on tso
22     switch result do
23         case out_of_heap
24             pushOnRunQueue(hec,tso); performGC;
25         case out_of_stack
26             enlargeStack(tso); pushOnRunQueue(hec,tso);
27         case time_expired
28             pushOnRunQueue(hec,tso);
29         case finished
30             if bound then
31                 return
32             else
33                 continue;
34             end
35     endsw
36 end
```

Figure 3.4: Main scheduling loop for GUMSMP, combining GHC-SMP and GHC-GUM functionality

3.5 Work Distribution Mechanism

3.5.1 GHC-GUM

The following description is summarised from [163, 164]. In GHC-GUM, if there are no more threads to run in the thread pool, the scheduler searches for sparks in its spark pool. If a spark is found, it is activated by turning it into a thread and generating a TSO to hold essential information about the thread and begin evaluating it. If the running thread is blocked on an unevaluated value, it will be put in a queue. When the required data arrives, the blocked thread will be awakened and transferred to the runnable pool. The data becomes available through concurrent evaluation by another thread, or when another PE sends its value following evaluation.

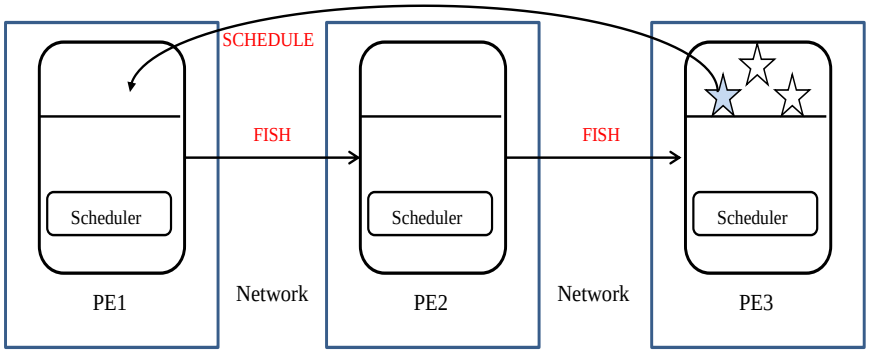

Figure 3.5: Work distribution in GHC-GUM

If there is no spark in the PE's spark pool, the scheduler requests work by sending FISH (work-request) message, PE1, as illustrated in Figure 3.5. The FISH message swims randomly from one PE to another searching for work. It includes the originating PE's id and age number which represents the maximum number of PEs to visit. If the recipient PE has no spark in its spark pool (PE2 in the figure), it forwards the message to another PE, which is chosen at random after increasing its age. If the recipient has a spark (PE3 in the figure), then it sends this to the requesting PE as a SCHEDULE (reply with work) message [3]. If no spark is found and the message limit is reached, the unsuccessful FISH is returned to the originating PE, which then waits before sending another FISH

[3]It is possible to send several pieces of work, if a FISH originates from a remote cluster [2]

message. This avoids inundating the machine with FISH messages when there are only a few busy PEs. For the same reason, each PE has a limited number of outstanding FISH messages. This mechanism is called "work stealing", or passive work distribution, since work is only requested by the idle PE.

3.5.2 GHC-SMP

The following description is summarised from [119]. An HEC's spark pool is implemented as a bounded work-stealing queue; this makes spark distribution cheap and asynchronous. A work-stealing queue is a lock-free data structure wherein the owner can push and pop from one end of the queue without synchronisation. Other threads can steal from the other end of the queue, meaning that only one atomic instruction is required. To avoid a race between popping and stealing threads from the queue when it is almost empty, popping incurs an atomic instruction. On the other hand, when the queue is full, the new spark that is to be pushed is discarded. This means that potential parallelism might be lost.

Figure 3.6 demonstrates the work stealing mechanism for sparks in GHC-SMP. In this example, HEC2 has no assigned work, it searches for a spark, either in its spark pool or in any other local HEC's spark pool (HEC1 in the figure). If a spark is found, then HEC2 creates a "spark thread" to reduce the thread creation overhead, which in turn steals the spark and begins evaluating it. Once this process has finished, it steals another spark.

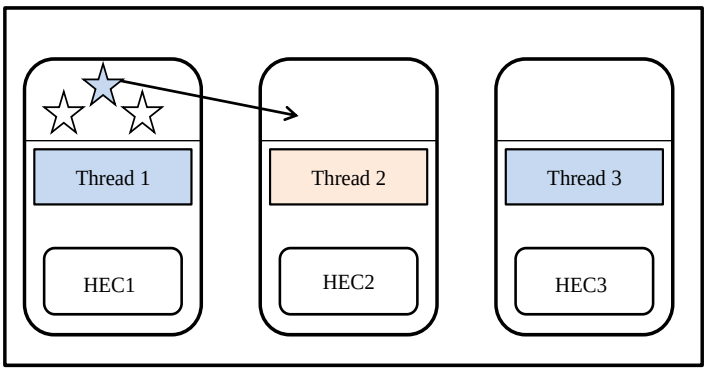

Figure 3.6: Work distribution in GHC-SMP

The "spark thread" is an ordinary thread, with the exception that it executes

the following steps in a loop:

1. If the local run queue is not empty, it exits.
2. It removes a spark from the local spark pool, or if that is empty, steals a spark from another HEC's pool.
3. If there are no sparks to steal, it exits.
4. It evaluates the spark to Weak Head Normal Form (WHNF).

Thus, the "spark thread" will evaluate sparks to WHNF sequentially, until no more sparks are found; at which point it exits, allowing the TSO to be collected by the GC. It is necessary to create a "spark thread" to avoid creating a new thread and fresh TSO for every spark and to discard it after completing the evaluation for recovery by the GC. In this way there will be only one thread executing multiple sparks. This also fixes the problem of latency between creating parallel tasks and being able to execute them in another core.

Strategically speaking this is a simple and effective method, whereby the cost of creating the "spark thread" is spread over multiple sparks, and the "spark thread" removes itself immediately when other work arrives. As the "spark thread" is an ordinary thread, it will be blocked in the normal way; if it blocks on a black hole; the scheduler will then create another "spark thread" to maintain the running of the available sparks. Having too many "spark threads" is not a problem, as a "spark thread" will always exit when there are other threads around. However, if sparks are too large and become blocked, this could lead to the creation of an excessive number of running "spark threads".

In GHC-SMP, the *work stealing* mechanism is used *for sparks*, and *work-pushing for threads*, implementing cheap migration on a physically shared memory machine. This is achieved as follows: in between running threads, the HEC with more runnable threads available checks to determine if there are idle HECs. If this is the case it will push runnable threads to the idle HECs thread pools by temporarily acquiring ownership of the idle HEC.

3.5.3 GUMSMP

The main objective of the new GUMSMP work distribution mechanism is to balance the load between the multi-cores. Naturally, the aim is to achieve even load balancing to ensure the best utilisation of all computing resources. However, a combination of multi-cores at the lower level (where several local cores can execute tasks that may in turn generate new parallelism), and a high-latency network connecting nodes, at the higher level (which makes the transfer of work and data expensive), demands the use of different policies to attain a balance between even load distribution and communication costs.

The GUMSMP work distribution mechanism is summarised in Figure 3.7. Within a multi-core, the exchange of work is considerably cheaper, and therefore, can be undertaken far more aggressively: an idle HEC (HEC2 from PE1 in the figure) will directly access the spark pools of other HECs within the same physical shared memory machine (HEC1 from PE1 in the figure), and collect work from there following the work-stealing approach, if it has no sparks of its own. If a spark is found, then the "spark thread" will be created to evaluate the sparks to WHNF, until there is no spark to steal, at which point it will exit as demonstrated in details in Section 3.5.2. Additionally, an HEC with a filled runnable threads pool may actively push threads to idle HECs, following the work-pushing approach of GHC-SMP, with the aim of more rapidly distributing work among all the HECs.

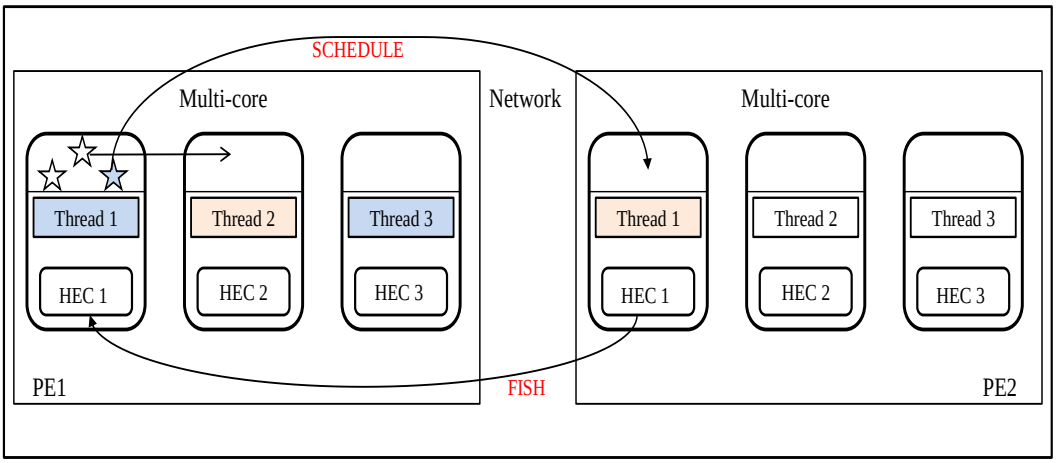

Figure 3.7: Work distribution in GUMSMP

At the cluster level, we use explicit FISH messages (as in GHC-GUM), with

a delay to acquire sparks from remote PEs (HEC1 from PE2 sends the fish to HEC1 from PE1 in the figure). The delay needs to reflect the communication costs on the network, in order to avoid flooding the system with FISH messages, while also being able to react sufficiently quickly when becoming idle.

We combine active and passive load distribution at the intra-node level, and passive load distribution (including work pre-fetch) at the inter-node level. Notably, the GUMSMP design for load distribution is hierarchy-aware. When looking for work, each HEC prefers local sparks from its own spark pool, or directly steals sparks from the pools belonging to other HECs running on the same PE. Only if no local spark is available will a FISH message be sent to another PE in the system. The concrete work balancing algorithm for GUMSMP is presented in Function 3.8, distinguishing the components related to the intra-node (GHC-SMP) and the inter-node (GHC-GUM) interaction.

```
1  void ScheduleFindWork(Hec *hec , Task *task)
2  if emptyRunQueue(hec) then
3  |   // get local work; GHC-SMP-style
4  |  if anySpark(hec) then
5  |  |  for i ← 1 to num_hecs do
6  |  |  |  if emptySparkPool(hec[i]) then
7  |  |  |  |  continue;
8  |  |  |  end
9  |  |  |  spark = tryStealSpark(hec[i]);
10 |  |  |  if spark != NULL then
11 |  |  |  |  break;
12 |  |  |  end
13 |  |  end
14 |  |  if spark != NULL then
15 |  |  |  tso = createSparkThread(hec,spark);
16 |  |  |  pushOnRunQueue(hec,tso);
17 |  |  end
18 |  else
19 |  |  // get remote work; GHC-GUM-style ;
20 |  |  pe = choosePE();
21 |  |  sendFISH(hec,pe);
22 |  end
23 end
```

Figure 3.8: ScheduleFindWork function in GUMSMP, combining GHC-SMP and GHC-GUM functionality

3.5.3.1 The Role of the Gateway HEC

In GUMSMP, one HEC is nominated as a gateway, and communication is restricted to this HEC. Other designs are possible, such as every HEC may communicate, which would then require synchronisation to send and receive messages. Our design restricts the communication to the gateway HEC and hence offers simplicity and avoids the need for synchronisation mechanisms to send messages and packing graph structures. Such synchronisation would involve unacceptably large overheads, increasing with high core numbers, and with an increase in the packet size to be communicated, thus hampering scalability. The advantage of restricting communication to a gateway HEC is that the gateway can incorporate the most accurate picture of the current system's information; e.g. the load on different machines. Currently, GUMSMP does not maintain the system load information but it represents potential improvement. Furthermore, as a gateway to other nodes, it can prefer to accumulate those sparks in its spark pool that would be the most profitable to export, thus creating a finer distinction between the sparks available. We investigate the implications of making this distinction in Section 4.3.3.

On the other hand, this gateway HEC might become a bottleneck for high core numbers, where faster communication is important for getting remote work, and for responding to remote requests. An alternative design option we consider is preventing the gateway HEC from performing any computation, thus responding faster to messages from remote PEs. However, with such a design, each node in the system loses one computation engine, which might degrade performance, especially if the computation-to-communication ratio is large.

We have fully implemented the two alternatives with the analysis of the performance results in Section 4.3.4.

3.5.3.2 Exporting Sparks

In GUMSMP, when a FISH arrives from another PE, the HEC first searches the spark pool of the gateway HEC, then the spark pools of the other HECs, to serve the FISH message as presented in Figure 3.9. This reflects our design using a single dedicated gateway to other cores in charge of communication, but also

identifies work to export.

A further option would be to select a spark from the HEC with the largest spark pool, and send it as a response to the message. However, this would require traversing all the HECs to identify the one with the largest spark pool, thereby imposing additional overheads, something that we wish to avoid.

```
1  void ExportSpark(HEC *hec , Int requestingPE)
2  if sparkAvailable then
3  |  spark = tryStealSpark(gateWayHec);
4  |   // first search the spark pool of the gateway HEC
5  |  if spark != NULL then
6  |  |  PackAndExport(spark,requestingPE);
7  |  else
8  |  |   // then search the spark pool of other HECs
9  |  |  for i ← 1 to num_hecs do
10 |  |  |  if emptySparkPool(hec[i]) then
11 |  |  |  |  continue;
12 |  |  |  end
13 |  |  |  spark = tryStealSpark(hec[i]);
14 |  |  |  if spark != NULL then
15 |  |  |  |  break;
16 |  |  |  end
17 |  |  end
18 |  |  if spark != NULL then
19 |  |  |  PackAndExport(spark,requestingPE);
20 |  |  end
21 |  end
22 end
```

Figure 3.9: ExportSpark function in GUMSMP

3.5.3.3 Sparks Placement

Once a stolen spark arrives at a node, the system should decide on a spark pool to place it in. For GUMSMP, the default choice is to assign it to the spark pool of the gateway HEC. Since HECs can exchange work cheaply in their spark pools, this indirect way of retrieving work should not incur any significant delay. However, a general problem with work distribution in a virtual shared heap model is the danger arising from *heap fragmentation* i.e. logically related data structures reside on different PEs. This can occur when the related data is spread over several nodes, mainly due to work-stealing or a FETCH request.

One RTS parameter indicative of high heap fragmentation is the size of Global Indirection Table (GIT). We explore this issue in detail in Section 4.3.3.

3.5.4 Hierarchy-aware Load Balancing

In contrast to a flat network of single-cores, an idle multi-core represents several unused computation engines. To account for the hierarchical nature of clusters of multi-cores, we therefore provide a refinement to this pure model, in the form of *pre-fetching work*, which is controlled by a *low-watermark* associated with the spark pool. Moreover, we also provide a spark segregation mechanism in the form of an "import-spark-pool", in an attempt to improve heap fragmentation in hierarchical architectures.

3.5.4.1 Watermarks

One simple (but flexible) mechanism that gives better control of spark distribution is to use *low-* and *high-watermark*s for the spark pool. When using this approach, work offloading decisions can be based on the size of each spark pool, as shown in Figure 3.10. The *low-watermark* specifies a minimum number of sparks to be held in the local spark pool. If the number of sparks falls below this watermark, no further sparks will be exported, and the PE will attempt to obtain additional sparks from other PEs. This mechanism is designed for high latency systems, and aims to conduct pre-fetching work; thus, supporting effective latency hiding, which is one of our main design principles.

The *high-watermark* indicates the maximum number of sparks that should be held in the spark pool. If the number of sparks exceeds this limit, then the PE will attempt to off-load sparks actively to other PEs, without receiving work requests. It uses SCHEDULE messages, in the same manner in which a PE serves FISH messages. Thus, the PE will temporarily and locally switch from a lazy load distribution to an eager load distribution, until the spark pool size again falls below the *high-watermark*. Where all PEs have a large number of sparks, a back-off mechanism will be used to introduce delays between each SCHEDULE message (as described earlier for FISH messages).

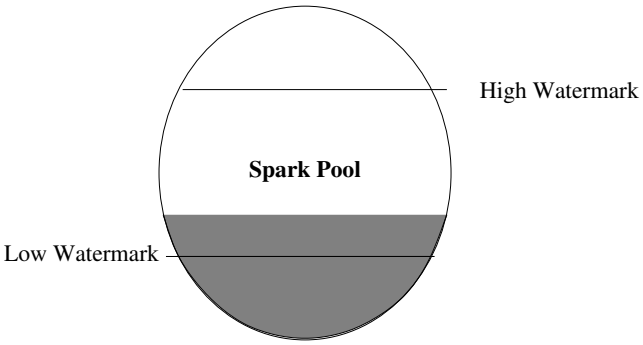

Figure 3.10: Low- and High-watermark mechanisms for load distribution in GUMSMP

3.5.4.2 Spark Segregation

A possible means of addressing heap fragmentation would be to separate locally generated sparks from imported sparks. We propose two main alternatives to implement this mechanism as follows:

Import Spark Pool: The import spark pool is the mechanism implemented in GUMSMP; there is a separate spark pool dedicated to imported sparks, as demonstrated in Figure 3.11. This will ensure related pieces of work are kept together in one pool, but it does require additional stealing steps, to acquire external work.

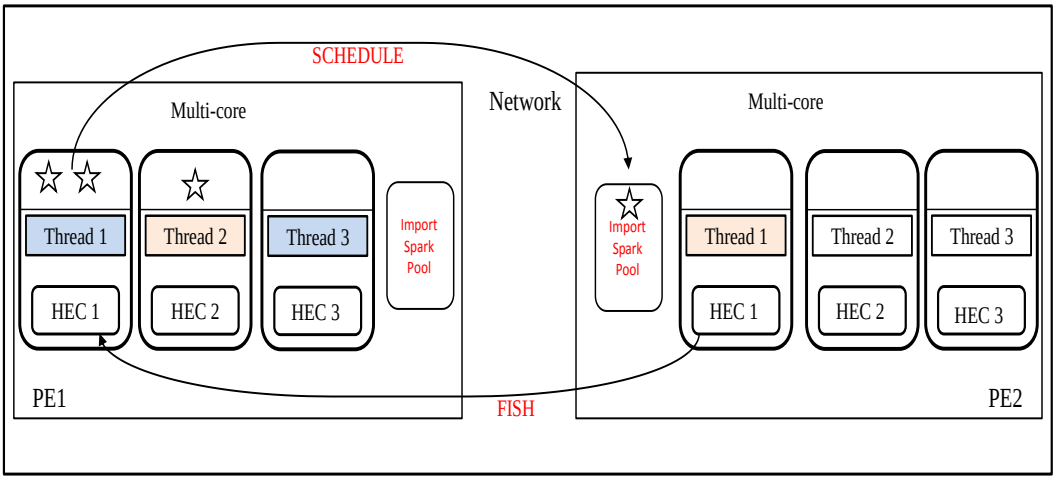

Figure 3.11: Work distribution in GUMSMP with import-spark-pool

An additional spark pool of this type would also be useful in situations in

which no HECs are idle at the time the new spark arrives. This can arise if a thread, which has previously been blocked on remote data, has been awoken in the meantime and then generated fresh sparks. Placing the imported sparks into a dedicated spark pool would defer the placement decision to a later time, when idle HECs would be available. Committing too early would not be an ideal use of the dynamic information of the system. Related to the separation of sparks into local and imported is a decision regarding whether to duplicate work when sending sparks to other PEs. The current design aims never to duplicate work, because there is an obvious danger that performance could be degraded. However, in view of the more aggressive load distribution policy required for GUMSMP, it might be acceptable to duplicate some work, if that work remains separate from the main pool of sparks and is activated only when other methods of obtaining work fail. The current implementation already provides a means by which to control potential work duplication, by defining a globalisation policy, and determining which kinds of closures to generate a (unique) global address when packing a graph structure (Section 3.7).

According to the separation of sparks as local or global, different policies have been identified for:

- Selecting spark for local evaluation, or
- Exporting a spark to remote PE, in response to work-request messages.

Concrete settings for these policies are:

1. Prefer local: local sparks are preferred; if they are not available then global sparks can be selected from the import-spark-pool.
2. Prefer global: global sparks are preferred, so first select global sparks from the import-spark-pool, if there are no global sparks, then local ones are sought out.
3. Local only: only select local sparks.
4. Global only: only select global sparks.

Detailed measurements to evaluate the effectiveness of the different policies are provided in Section 4.3.3.

Spark Tagging: An alternative mechanism can be achieved by tagging an imported spark to indicate that the spark is global. Then the tagged spark can be handled according to the policies discussed with the import-spark-pool; e.g. prefer (tagged or global) spark, or prefer (un-tagged or local) sparks, either for local evaluation or for remote exporting. Spark tagging would be a useful mechanism as the tag can be used for different purposes, such as grouping sparks together that derive from the same source, to improve data locality, or to separate imported sparks from locally generated ones. The difficulties involved in implementing this approach concern how to reach the tagged spark cheaply, as it will be placed in the spark pool of the gateway HEC, and there might be several locally generated un-tagged sparks. In which case, a traversal of the spark pool is required to find the global spark, which will then be an unavoidable overhead. Moreover, further complexity is involved, as care must be taken to handle tagged sparks, because they should be un-tagged prior to their evaluation.

3.6 Memory Management

3.6.1 GHC-GUM

The following description is summarised from [163, 164, 106]. Every PE has a local memory integrated into the global distributed heap, as indicated in Figure 3.12; and a two level addressing scheme, one for Local Addresses (LAs), and one for Global Addresses (GAs). This is used to reference values in the shared heap. GAs enable each PE to garbage collect locally, without the need to synchronise with other PEs. A GA is a globally unique identifier for a closure, which is created as a result of sending work from one PE to another in response to a work-request message. After a thunk, representing work, is sent to the requesting PE, the original thunk is overwritten with a FetchMe closure, a global indirection, containing the GA of the new copy of the thunk at the destination.

The aim of overwriting a thunk with FetchMe is to indicate that it is being evaluated by another PE, and to indicate its new location, should the result be needed subsequently by the original PE. The GA consists of a locally unique identifier, the PE identifier of the destination, and a weight, as discussed below.

A Global Indirection Table (GIT) is maintained within each PE to map the global identifiers to the local address of the corresponding heap closure. The GIT acts as a source of roots for local GC. This design enables each PE to garbage collect independently, provided that the GIT is adjusted after each GC to reflect the new location of the local heap closures.

GAs are garbage collected using the standard distributed weighted reference counting algorithm [101]. When a GA is created, it has an initial weight that is split whenever the reference is shared. This mechanism aims to minimise the synchronisation required among referrers to a single closure. When a global object is locally garbage collected, the associated reference weight is returned to the owning PE. The mapping of global to local addresses is required to ascertain whether a copy of a newly imported graph structure already exists on that PE, in addition, to avoid duplication of data and work. If a newly imported graph structure exists, the version of the graph, that has been evaluated less, will be subsumed by the more evaluated version.

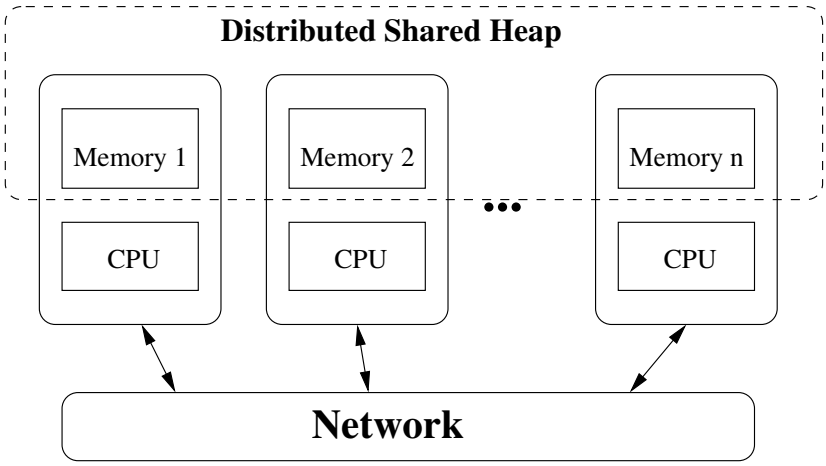

Figure 3.12: Distributed shared heap in GHC-GUM

3.6.2 GHC-SMP

The following description is summarised from [114, 119, 176]. In short, GHC-SMP uses a generational, parallel (but not concurrent), stop-the-world GC. Parallel GC requires all threads to stop the computation and performs GC at the same time, whereas concurrent GC can take place with one GC thread while other threads are doing computation work [117]. This section discusses details that are

important for assessing GHC-SMP vs. GUMSMP performance on NUMA architectures (in Section 5.2). GHC-SMP's memory management is based on the concept of a blocked-structured heap, where the shared heap is divided into non-contiguous, fixed-sized blocks. A block allocator manages the blocks, which can be singly allocated, or linked together into lists to form an allocation area, that can then be provided to each HEC to allocate fresh objects. They can also be linked in contiguous groups to allocate large objects with sizes larger than a block size. The operating system provides the block allocator with memory initially and when it runs out of it.

The main benefits of the block-structured are:

1. Flexibility: Blocks are not contiguous, so they can be linked together in order to provide the required size for individual regions of memory, including heap, generations, steps, and allocation areas for mutators to allocate fresh objects.

2. Easy Management of Large Objects: During the GC, there is no need to copy large objects, but rather, a linked list is maintained for each step of each generation, so large objects are moved by re-linking their blocks to the linked list of the destination step and generation.

3. Reducing heap fragmentation, by quickly recycling free memory for re-use in other contexts.

4. In the situation where more memory is required at a time when GC cannot be performed, such as a C procedure deep inside the RTS, additional blocks can be allocated on demand, thus delaying GC until a more convenient time.

All objects in the heap consist of two main components: an info-pointer, which points to an info-table providing more layout information to guide the GC such as the object type, and associated entry code.

The garbage collector implemented in GHC-SMP is a generational copying garbage collector, based on dividing the shared heap into generations of fixed-size blocks. Generations are numbered from 0 to n, with 0 being the youngest. The youngest generation, in which the objects are allocated are frequently garbage

collected whenever memory is exhausted. Objects are promoted from the younger generation n to the older generation $n+1$, which is collected less frequently when they survived specific number of collections. A remembered set is maintained to keep track of all pointers referenced from mutable objects in the older generation to the younger ones. Whenever a mutable object creates a pointer into a younger generation, an entry for that object must be added to the remembered set and considered a root for GC.

With the "weak general hypothesis", objects allocated recently are the most likely to die young and become unreachable. As a considerable portion of memory is allocated by functional programs, it is desirable to avoid the immediate promotion of young objects as they are likely to die by the time of the first GC. Therefore, each generation n is further subdivided into kn steps; so reachable objects from generation n are only promoted to generation $n+1$ when it is from step k. Objects from younger steps remain in generation n, but with an increased step count.

In copying collection, promotion takes place by evacuating objects, this means copying them into the *to-space*. Then, each object is scavenged in the to-space; by evacuating each pointer in the object, and replacing the pointer with the address of the evacuated object. The GC completes, when all the objects in the to-space have been scavenged.

This GC is parallel (but not concurrent) and stop-the-world; thus it is initiated by a HEC with an exhausted allocation area; this takes place when all the HECs have been synchronised to start the GC. The initiating HEC is responsible for pre-GC initialisations, it then releases the other GC threads to perform the GC. After completing GC, all other GC threads wait in the GC exit barrier, and are released by the initiating HEC after performance of any post-GC tasks.

For parallel copying GC, it is important to evacuate or scavenge each object using different threads. Each GC thread synchronises to acquire a private *to-space* allocation block. Local per-HEC remembered sets are maintained to avoid synchronisation costs and to improve data locality, as TSOs that have been executed on a given core, with the data they refer to, are likely to be present in the core cache, and therefore traversed by the garbage collector on the same core. GC

starts by evacuating roots from the *from-space* to the *to-space*, and then in a loop traverse the *to-space* to scavenge each object. A block in the *to-space* represents a unit of work because the heap is blocked-structured.

Work stealing queues are used to achieve load balancing of the GC. When the GC begins, each HEC already has a considerable data in its cache. Therefore, the GC thread takes blocks to scavenge from its own queue in preference to stealing, beginning with blocks from the oldest generations. If no work is available in its own queue, then the HEC will try to steal work from the queues of other HECs. This design improves locality and reduces the contention of a single, global work queue, which was originally implemented in GHC-SMP. In fact, stealing work from the queues of other HECs in order to balance the load is to be avoided with minor collections as it has a detrimental effect on locality [119].

Lock Contentions: during parallel GC, synchronisation is required for the following parts:

1. There is one global lock in the block allocator to obtain a new block for a GC thread: Each GC thread needs to allocate a block into which objects can be copied when they are evacuated. Contentions to this lock are reduced by allocating multiple blocks simultaneously, and by keeping the spare ones on a private partly-free-list associated with the thread. When a GC thread requires a fresh allocation block, it first searches in its partly-free-list to reduce synchronisation overhead.

2. One lock per step in the large-object lists: Large objects with sizes greater than a block size are allocated in a block group of contiguous blocks. A linked list of large objects is maintained for each step of each generation. During the GC, those large objects are treated differently from other objects as they are not copied, but instead are moved by re-linking them from one linked list to another; and therefore a lock is required.

3. The per-object evacuation lock: To prevent multiple GC threads from copying the same object, an atomic instruction is required for synchronisation. This synchronisation represents the major source of overheads for the parallel copying GC with up to 30% of the GC time [114]. In improvements to

the original design, this contention was reduced by relaxing the lock when copying immutable objects, resulting in a 7% improvement. Since the rate of actual collisions is very low, the space wasted by duplicate copying is negligible [119].

3.6.3 GUMSMP

The GC component of GUMSMP integrates the garbage collections of GHC-GUM and GHC-SMP, and therefore the entire discussion in the previous sub-sections apply here as well. Figure 3.13 represents the distributed shared heap implementation in GUMSMP. The PE in GUMSMP represents an instance of GHC-GUM, which includes a number of GHC-SMP HECs. Therefore, for a PE to perform a GC, there is no necessity for synchronisation with other PEs; however, within each PE, all HECs are synchronised to perform GC following the same mechanism of GHC-SMP GC.

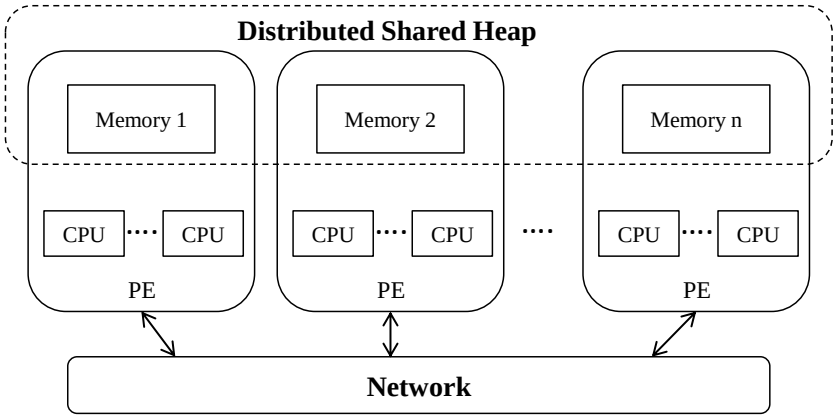

Figure 3.13: Distributed shared heap in GUMSMP

3.7 Communication

This section describes the communication mechanism of GHC-GUM (summarised from [106, 51, 2, 163, 164]), which was adopted in GUMSMP. When a thread enters a FetchMe closure for the purpose of evaluation, that thread is put on a global blocking queue. Any subsequent thread trying to enter the FetchMe will also join the queue. A FETCH message is then sent to the PE that owns the

GA (Global Address) referred to by the FetchMe closure. The sending PE overlaps the time of communication with the execution of other local threads, or the generation of new threads. Upon receiving the FETCH request, the target PE packages up the required closures as well as a (part of) its sub-graph, and sends it in a RESUME message. If the required closure is under evaluation at the time of receiving the FETCH, then a BlockedFetch closure will be created and put in a global BlockedFetch queue, for a RESUME message to be sent later whenever the evaluation is completed. On receiving the RESUME message, the graph is unpacked and the FetchMe is overwritten with a (local) indirection to the root of the received graph. All the other threads blocked by the global closures are then awoken. Finally, an ACK message is sent to the PE, which sent the RESUME message, to confirm the new location of the received closure.

Packing and Communication Scheme: For a closure to be communicated, the graph representing it must be packed, or serialised; i.e. it must be converted into a self-contained array of bytes to be communicated. GHC-GUM uses asynchronous, bulk communication, which has the effect of reducing the total amount of communication, and also permits latency hiding, where communication is overlapped with computation.

In high-latency networks, packing only a single closure into messages may be too expensive. GHC-GUM is designed for full sub-graph packing; that is limited by the fixed packet size, as default. When a closure is packed, the reachable graphs are added to the packet, which reduces the number of FETCH messages that need to be sent. In other words, the graph is packed breadth first, up to a fixed limit. In rare cases, this might result in the sending of over-abundance data. The maximum number of thunks per packet can be specified as a tunable parameter to the RTS.

During the packing of a graph, the addresses of the closures are stored in a temporary table, in order to enable the detection of sharing and cycles. If the packet is at full capacity, FetchMe closures and GAs are then used to refer to the graph remaining. The reminder of the graph is sent lazily, i.e. on demand, rather than immediately (as the related RTS for Eden).

Although each closure packed is made global, there are a few specially packed closures:

1. Values that are already in normal-form are not globalised. Since they are already evaluated, they can be copied freely between machines.

2. Black Holes, i.e. closures that are being evaluated, are packed as FetchMe's to the black hole. The unpacking algorithm first reads the packet and then reconstructs the graph breadth first.

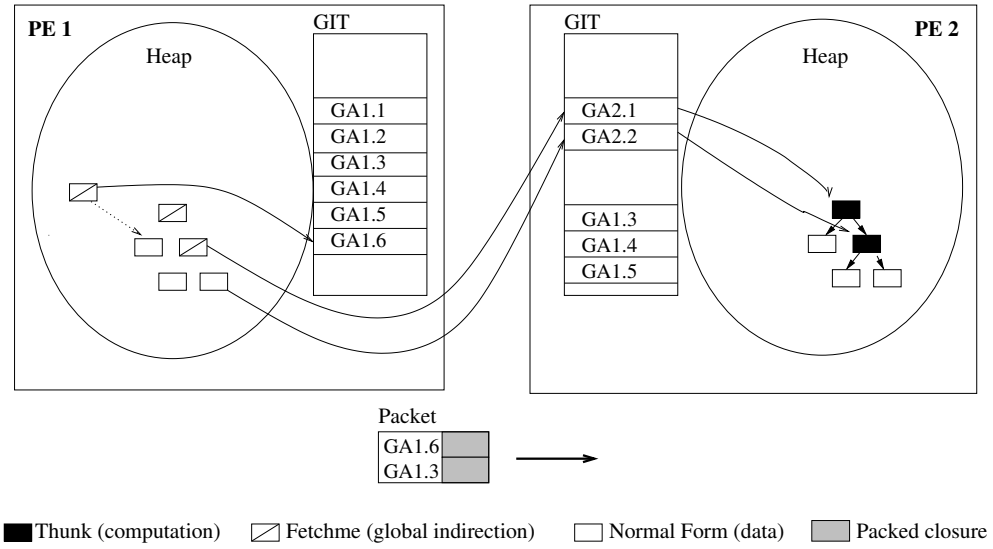

Figure 3.14: Transfer of graph structure [163, 164]

Figure 3.14 details the allocation of GAs and the transfer of graph structures on two PEs. This picture depicts the heaps on two PEs after the transfer of the five closure graph with root GA2.1 on PE2 (originally GA1.1 on PE1) has been completed. The graph is traversed breadth first by the packing algorithm. A new GA is allocated for each closure that does not already have one; in this example, GA1.1 to GA1.5. Thunks must be treated differently from normal-forms when packing the closure.

In order to prevent work from being duplicated, thunks are never copied, rather they are moved between processors. As a result, an original thunk is replaced with an RBH closure. If other threads request the value before the transfer has been completed, they will be blocked. In contrast, normal-forms can be copied freely. The graph is unpacked by the receiver, check is made for the

possible presence of any other copies of the imported closures, to maintain sharing. New GAs are allocated for thunks, determining where the closure is located. In this example, GA2.1 and GA2.2 are newly allocated. Their original GAs, GA1.1 and GA1.2, are no longer needed, and so can be garbage collected. When the entire graph has been unpacked, a mapping of old to new GAs is emitted to the sender; whereby, all RBHs are subsequently replaced with FetchMes, representing global indirections to the new GAs, GA2.1 and GA2.2; whilst the old GAs become garbage.

Figure 3.14 also depicts the ongoing transfer of a two closure graph, which shares one closure with the first graph. Note that in the packet, GA1.3 refers to the same (shared) closure as that now available on PE2, so that when unpacking, the second graph sharing of the closure is maintained on PE2.

3.8 Communication vs. Evaluation

In the implementations of parallel Haskell, duplicate evaluation of thunks is avoided by blackholing (Section 3.4.2), which makes a duplicate evaluation of the same thunk highly unlikely. A potential race exists in GUMSMP between a thread that is evaluating a thunk, and another thread that tries to pack it for communication. In this section we discuss how the lock-free mechanism implemented tackles this potential race. It is necessary to distinguish between different reasons for packing and type of the closure being packed. We might pack a closure to send to remote PEs in response to a work-request message (FISH) or return back a result of previously imported work (FETCH). Since the window for packing a closure is wide (a large sub-graph may need to be traversed), another thread might claim the thunk being packed and evaluate it locally. Therefore, to avoid a race between the packing and evaluating thread, we implement a lock-free mechanism, based on checking the closure type at three different points during the packing, and recording changes to the closure resulting from the evaluation. We check the closure type before we start packing, and during the packing, and finally after packing, just before sending the packet. We distinguish between those different closure types as follows:

1. **Thunk** represents a closure with work that has not yet been evaluated.

2. **BH** represents a closure that has been claimed locally for evaluation. Therefore, it is under evaluation by another local thread at the time we start the packing.

3. **IND** represents a fully evaluated closure with indirection to its result.

We also need to make a distinction between whether the closure to be packed is a root of the graph or part of the graph structure. Since IND points to fully evaluated data, it can be packed safely. A BH closure on the other hand is treated differently based on whether it is a root or part of the graph structure. If it is not the root, then we pack it as FetchMe, no matter whether the evaluation has been completed after the packing immediately prior to sending off the packet. This is safe because if we pack a FetchMe, then on arrival, the remote PE will send a FETCH message asking for that closure. Therefore, the main overhead in this case is the additional communication involved.

The main case to consider is a thunk, as it represents work, and we would like to avoid duplicating the evaluation of the thunk. We split the "packing" and "evaluation" operations into phases. Table 3.1 shows the phases involved for the packing and local evaluation of a thunk.

Table 3.1: Different activities for packing and evaluation of a thunk

Packing	Evaluation
1. Get local copy of the closure type.	1. Enter thunk code.
2. Enter the packing code.	2. Write BH.
3. Compare local copy with the closure type.	3. Evaluate thunk.
4. Write RBH.	4. Update thunk with its result.
5. Pack the thunk.	
6. Check the closure type.	
7. Send the packet.	

Any interleaving of the two sequences in these phases is possible. In order to manage this complexity, we use a transition diagram in Figure 3.15 to present details of how the implementation handles different cases of interleaving. The right and left sides represent the different states from the evaluating thread, and the packing thread points of view respectively. As shown in the legend for

Figure 3.15 (top left), each state shows 5 main components reflecting the state of the closure in the heap (global), as well as the state of the closure from the packing and evaluating threads point of view (local). Moreover, it shows the state of the thread entering the closure for packing (left) or evaluation (right). It also reflects the transition that takes place for each thread, and illustrates how the packing thread responds to the interleaving that is triggered by the local thread, claiming the thunk under packing. Note that the red boxes represent problematic interleavings, and the discussion below focuses on those cases.

If the packing thread starts packing a thunk, it will maintain a local copy of the header of the closure, which is a thunk. Then, just before the thunk is packed, another thread might claim the thunk and write BH (the red box on the LHS of Figure 3.15). We detect this problematic case by comparing the header of the closure to be packed, which is now the BH with the local copy thunk. Having detected the thunk is now under evaluation, there are three main cases to consider in order to recover:

1. The closure is the root of the graph and the purpose of packing is to send a SCHEDULE message with work to respond to the work-request FISH message. In this case, the packing process can be safely aborted, because any reply of work is sufficient to serve the FISH. Thus, we have two possibilities when responding to the work-request. Either trying to steal another spark locally, or forwarding the FISH message to another PE. Our implementation forwards the FISH message.

2. The closure is the root of the graph and the packing purpose is to send a RESUME message to respond to the data-request FETCH message. In this case, we abort the packing and create BlockedFetch closure and add it to the BlockedFetch queue, because the data represented by this thunk is required on another processor. If the thunk is evaluated fully in the meantime, then the packing thread will re-pack the IND instead of creating a BlockedFetch.

3. The closure is in the body of the graph. In this case, we pack a FetchMe for this closure, regardless of the purpose of the packing. When the FetchMe is entered on the other PE, a FETCH message is then sent, asking for the

data of this closure.

On the other hand, if the packing thread claims the thunk faster (the red box on the RHS of Figure 3.15), then the thunk will be overwritten with the RBH, at which point the evaluating thread will be blocked on the RBH and placed in the Black Hole queue. By checking different aspects several times during the packing, the packing thread can identify closure changes. However, the window of duplication is not entirely closed, as with the highly unlikely case of possible duplication for a very short-lived thunks, as shown in Table 3.2, leading to work duplication (the red box on the bottom of Figure 3.15). This represents a very rare

Table 3.2: Race between packing and evaluation.

Packing	Evaluation
1. Get local copy of the closure type.	1. Enter thunk code.
2. Enter the packing code.	
3. Compare local copy with the closure type.	
4. Write RBH	2. Write BH.
5. Pack the thunk.	3. Evaluate a thunk
6. Check the closure type.	4. Update thunk with its result.
7. Send the packet.	

case, as duplication would only occur if the two threads were actually updating the header at the same time, and the evaluating thread wrote a BH, just after the packing thread compared the local type with the type in the heap. In this case, the packing thread might proceed with packing, but if the evaluating thread completes the evaluation and updates the thunk with its result, then the third check just before sending off the packet will catch the update, and abort the packing accordingly.

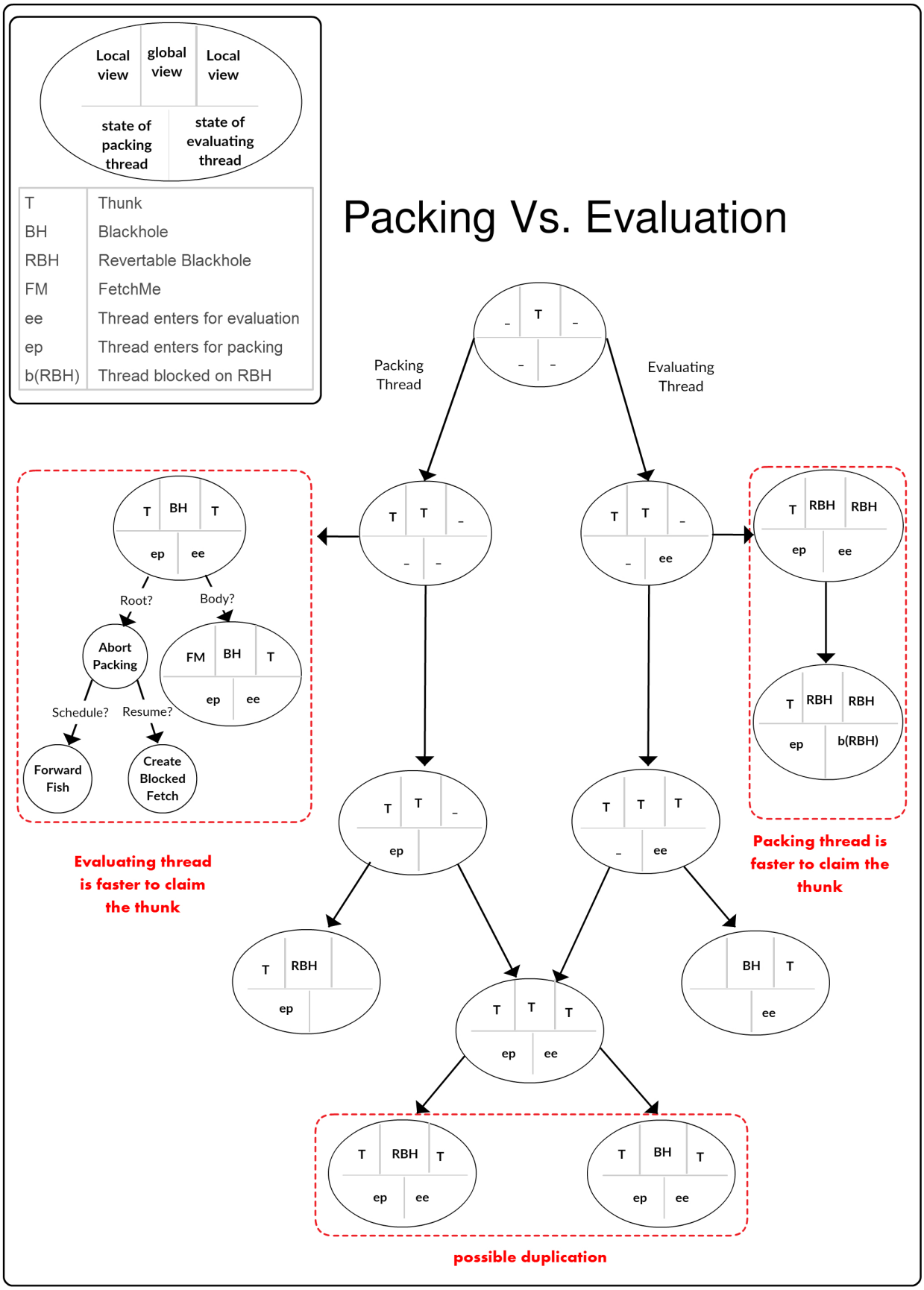

Figure 3.15: Packing vs. Evaluation

3.9 Summary

In this chapter we have presented the design of GUMSMP, discussing its main objectives. We have also demonstrated the main components of GUMSMP, highlighting features shared among the three parallel Haskell implementations; such as thread management, as well as new GUMSMP features to support *hierarchical load distribution*. We illustrated how GUMSMP combines GHC-GUM, and GHC-SMP, as well as discussing a set of design alternatives specific to GUMSMP, and the motivations for the decisions made. Our design focuses on flexible work distribution policies in hierarchical architectures. In particular, we have performed asymmetric load balancing, using different load distribution policies at different levels of the hierarchy. At the cluster level we used a less aggressive policy, which may produce some load imbalance but reduces the total amount of communication. Within a multicore node we used a more aggressive load distribution policy, which exploits the low communications overhead provided by physical shared memory. In the next chapter we show how those alternatives affect the performance of GUMSMP.

Chapter 4

GUMSMP Tuning

4.1 Introduction

A core design principle of GUMSMP is to perform adaptive, dynamic management of work and data. Thus a crucial step in the implementation of GUMSMP is the development of load balancing policies tailored for hierarchical architectures like clusters of multi-cores, or large NUMAs. This chapter focuses on these policies, and assesses their impact on performance. It documents the process and assesses the results of GUMSMP performance tuning, investigating different approaches to improving the performance on a cluster of multi-core. The majority of the results in this chapter are published in [6]. The results in Sections 4.3.3 and 4.3.4 are as yet unpublished.

This chapter is structured as follows. The baseline performance is presented in Section 4.2.2. We consider the following seven performance tuning mechanisms: the *low-watermark* for work pre-fetching in Section 4.3.1; Asymmetric load distribution policies in Section 4.3.2; Distinguishing Local and Global work in Section 4.3.3; The use of dedicated gateways in Section 4.3.4; Optimising the number of cores Per PE in Section 4.3.5; Optimising the setting of the allocation area in Section 4.3.6; A more active load management is discussed in Section 4.3.7.

4.2 GUMSMP Performance

4.2.1 Setup and Programs

Throughout the thesis, we used the following benchmarks that exhibit a range of parallel patterns, with characteristics specified in Table 4.1, and sequential performance specified in Table 4.2.

- `parfib` is a *divide-and-conquer* program, which computes for a given value, the well-known Fibonacci number `fib x` using a depth threshold of `y`
- `parmap-of-parfib` is a *data-parallel* program, with nested divide-and-conquer parallelism, computing `y` instances of parallel `fib x` computation.
- `coins` is a *divide-and-conquer* program, which computes the number of ways to pay a given value `y`, from a fixed set of coins `[55, 88, 88, 99, 122, 177]` (parameter `x` specifies the program's version and `z` the depth threshold)[1].
- `sumEuler` is a *data-parallel* program, which computes the sum of the Euler totient function on the list interval `[1..x]`, using a cluster size of `y`.
- `worpitzky` is a *divide-and-conquer* program, which checks the Worpitzky property over the Stirling numbers `x y` using a depth threshold of `z`.

Additionally, we have measured the performance of the following larger benchmarks.

- `minimax` is a *divide-and-conquer* AI application that performs an alpha-beta search in a 2-player game on a `x`×`x` board, up to a depth of `y`;
- `maze` is a *nested data-parallel* AI application for finding the path through a fixed maze.
- `mandelbrot` is a *data-parallel* application for computing a `mandelbrot` set over a given window size, and number of iterations.

[1]The input for coins used in Section 5.2 is (7 5200 3)

Table 4.1: Programs characteristics

Program	Application Area	Paradigm	Regularity	Input Parameters [x,y,z]
parfib	Numeric Analysis	D&C	Regular	52 23
parmap-of-parfib	Numeric Analysis	Data par with nested D&C	Regular	43 20
coins	Search Application	D&C	Irregular	7 4000 3
sumEuler	Numeric Analysis	Data par	Irregular	100000 180
worpitzky	Symbolic Computation	D&C	Regular	2 27 20
minimax	AI Search Application	D&C	Irregular	4 9
mandelbrot	Graphics	Data par	Irregular	-2.0 -2.0 2.0 2.0 4096 4096 3024
maze	AI Search Application	Nested Data par	Irregular	
blackscholes	Financial Application	Data par	Irregular	1000000000 500000

- blackscholes is a *data-parallel* application, which represents the implementation of the Black-Scholes algorithm for modelling financial contracts, by providing a number of options x, and the granularity y.

The starting point for the selection of benchmarks is the established nofib/parallel suite, and other parallel benchmarks used in previous papers [115]. We select a subset of these benchmarks covering regular and irregular parallelism, different parallel paradigms e.g. data parallel, nested data parallel, or divide and conquer, as well as different application domains as indicated in Table 4.1. The focus is on computation bound programs in order to test the computation component of the RTS. We do not explore other issues such as concurrent IO, foreign function calls, etc.

Our measurements in this chapter are made on a Beowulf cluster of multicores, where each node is an 8-core CPU (Intel Xeon E5504 running at 2.00GHz, and 12GB RAM). All 32 nodes are connected via a non-specialised Gigabit ethernet connection. All machines are running Linux CentOS 6.6. The implementation of the GHC-SMP RTS is based on GHC 6.12.3, using GCC 4.4.7 for compilation, and PVM3.4.5 for message passing. Each point in the measurement represents the median of three executions. The programs are compiled as follows:

```
ghc -parpvm -threaded -cpp --make -o prog_pp_thr prog.hs
```

Table 4.2: Sequential performance

Program	Runtimes(s)	Total memory(Gb)	Maximum Residency(Kb)
`parfib`	6907.9	9836.5	38,0
`parmap-of-parfib`	4670.6	4772.9	39.7
`coins`	2632.5	5620.5	92.7
`sumEuler`	2431.7	2969.8	166.3
`worpitzky`	2100.6	2350.4	35.5
`minimax`	792.15	1008.4	48.2
`mandelbrot`	4955.7	5940.7	503.5
`maze`	2871.1	6450.5	40.6
`blackscholes`	6132.5	4863.8	2,04 Gb

Note that we have not explored the de facto standard optimisation settings (-O2). Appendix A gives evidence that while this omission changes the absolute runtimes, the relative performance remains unchanged.

Table 4.3: Runtimes of GHC 6.12.3 vs. GHC 7.10.2

	`coins`			**`sumEuler`**		
No. cores	6.12	7.10	**diff(%)**	6.12	7.10	**diff(%)**
1	2779.6	2801.7	-0.7	2561.7	2253.9	13.6
2	1513.7	1515.8	-0.1	1348.3	1175.0	14.7
3	1033.2	1016.4	1.6	904.8	785.3	15.2
4	804.3	778.6	3.3	693.7	597.7	16.0
5	651.5	628.1	3.7	561.3	476.4	17.8
6	560.9	533.6	5.1	481.5	403.3	19.3
7	494.5	466.9	5.9	414.3	345.7	19.8
Geom Mean.			2.6			16.5

Our measurements are based on GHC 6.12.3, and Table 4.3 compares the performance of this version with the most recent version 7.10.2 for `sumEuler` and `coins` on up to 7 cores. GHC 7.10.2 shows predominantly lower runtimes in both cases. For `coins`, the difference in runtimes is fairly small with average difference of 2.6%, and for `sumEuler`, the average difference is 16.5%. In terms of the speedup on 7 cores, GHC 7.10.2 achieves slightly higher speedup (6.5 for `sumEuler` and 6.0 for `coins`) compared with GHC 6.12.3 where the speedup is (6.2 for `sumEuler` and 5.6 for `coins`). This variation between different versions is expected, as more recent versions aim to improve performance and have more

performance optimisations enabled. However, such differences are small, and mean that our measurements could have achieved up to 16.5% improvement if the most recent version of GHC was used.

4.2.2 Baseline Performance

4.2.2.1 Cross-System Performance

Historically, the sequential performance of functional languages trailed that of sequential imperative languages; their higher-level of abstraction means they still do to some degree [79]. However, advanced compiler optimisations have significantly narrowed this gap. This section quantifies the difference in sequential performance.

It provides a comparative performance evaluation of the sequential C, and the parallel Haskell for the `sumEuler` program. As demonstrated in Table 4.4, for the `sumEuler` of 60000, the sequential runtime of GUMSMP parallel Haskell is 41% greater than C. Moreover, the cut-off point of the parallel Haskell version, where it beats the sequential C version is with 2 cores. The scalability of `sumEuler` is discussed in more detail in Section 5.3.4.1

A comparison between C and an earlier version of GHC-GUM (GHC 4.06) for matrix multiplication showed a factor of 5 difference (5.8s vs. 30.3s) [108].

Table 4.4: Runtimes for `sumEuler` for parallel Haskell compared with sequential C version

Language	**Runtimes(s)**
sequential C	287.32
parallel Haskell 1 core	405.9
parallel Haskell 2 cores	205.9

Parallelising imperative languages is a challenging task for several reasons. Importantly, imperative languages are not designed with parallelism in mind, and so impose a need for the sequential ordering of commands. Moreover, they lack the high-level structuring constructs offered by functional languages, in particular high-order functions [79]. Therefore, they deliver lower productivity, as

the programmer is involved in the coordination of the low-level parallelism (Section 2.4.1).

4.2.2.2 Single Multi-core Performance

This section compares the performance of the three implementations: GHC-SMP, GHC-GUM, and GUMSMP on a single multi-core node of the Beowulf cluster specified in Section 4.2.1. The goal of this comparison, is to assess the potential for improvement when moving from a flat cluster design, as used in GHC-GUM, to a hierarchical design, as used in GUMSMP. Furthermore, by comparing the performance with the existing shared memory implementation, GHC-SMP, we can also quantify the additional overheads of the GUMSMP design on a single multi-core.

Figure 4.1 shows the performance results from three representative benchmarks for all three systems. They are measured on up to 7 of the 8 cores, because Marlow and others [119] report and discuss performance degradation when using all n cores, on a n-core multi-cores. We observe that, as expected for all the programs, GHC-SMP yields the best performance. GUMSMP is typically within 8% of GHC-SMP performance representing moderate overhead due to the hierarchical design. While the performance of GHC-GUM is usually close to the GUMSMP results, in the case of `minimax` its performance is significantly lower. Further analysis of this result reveals that this is due to the overheads associated with virtual shared heap management, which have to be paid in the distributed memory GHC-GUM but not in GUMSMP, which in this setup uses one PE of up to 7 HECs. We observe that on a single multi-core GUMSMP outperforms GHC-GUM in all cases, with the exception of `parfib`, where the difference is less than 4%.

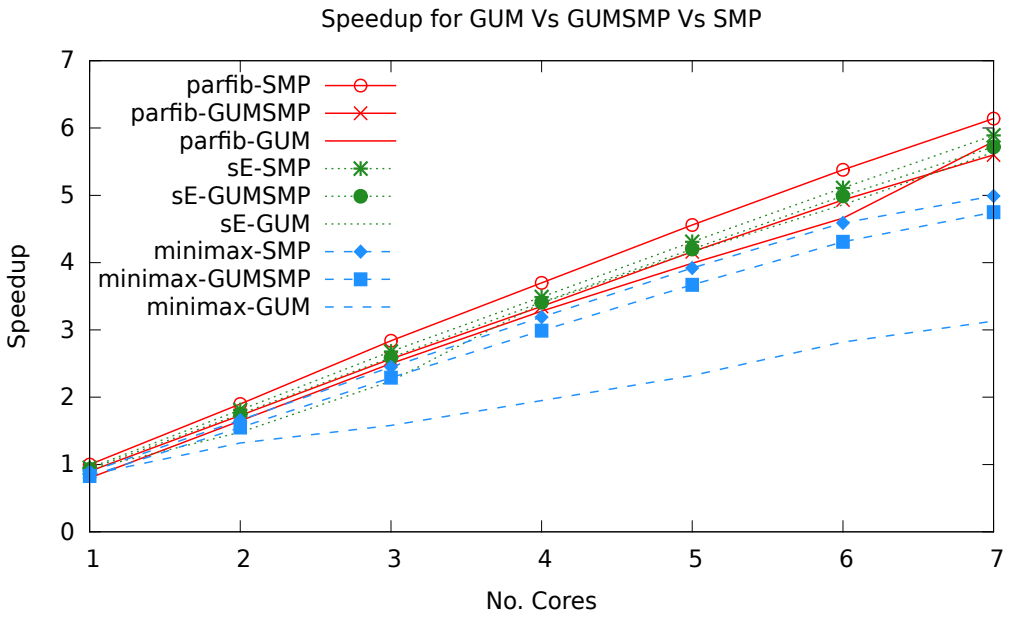

Figure 4.1: Speedup of three representative benchmarks using GHC-GUM, GUMSMP, and GHC-SMP on a *single multi-core*

4.3 Performance Tuning

This section focuses on studying the effect of different load balancing policies on the performance of GUMSMP.

Load distribution on hierarchical architectures, such as clusters of multicores, is necessary to account for the multiple cores in each cluster node and the large differences in latencies. For example, the latencies between cluster nodes are far greater than those within the node. This requires adaptation of basic policies developed in GHC-GUM, and designed for flat clusters. In particular, we examine the mechanisms to *pre-fetch* work, and to enable the multiple cores on a node to receive work as quickly as possible, thereby improving the load balance.

We start the performance tuning process by presenting the speedup of our GUMSMP in Figure 4.2, which shows the basic configuration of GUMSMP, as discussed in Section 3.5.3, before illustrating the effect of the hierarchy-aware load balancing mechanisms. Throughout the discussion in this chapter, the number of cores per PE is fixed at 5. In Section 5.3.3, we show how to optimise ideal settings for the number of cores per PE.

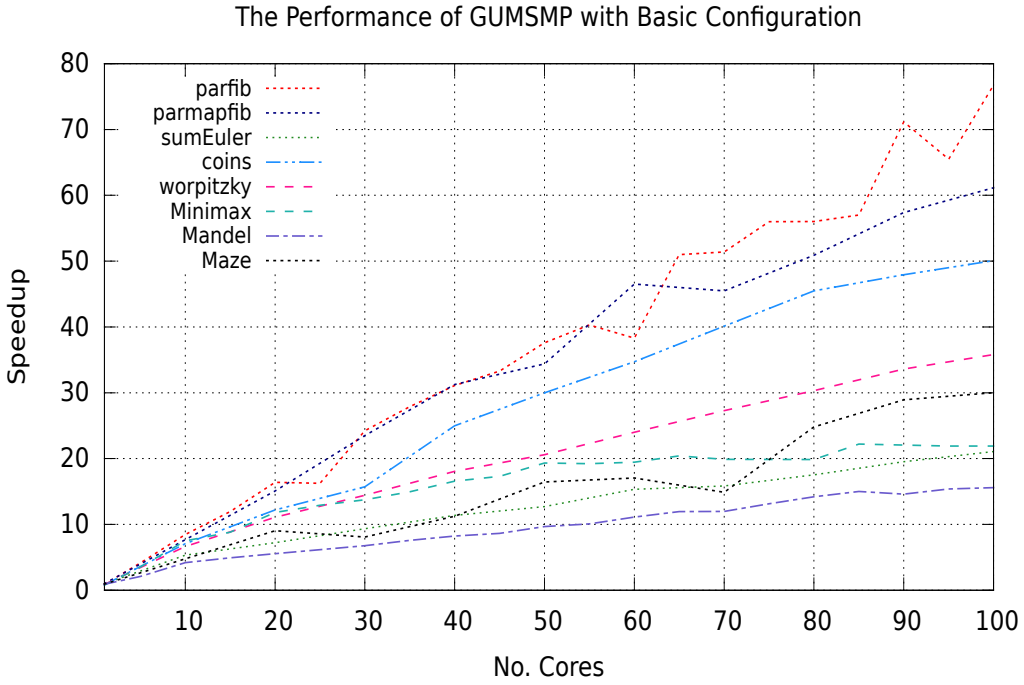

Figure 4.2: Speedup of GUMSMP on up to 100 cores with the basic configuration

The figure shows that for many programs GUMSMP speedup is acceptable without further tuning. Unsurprisingly the simplest micro-benchmark, `parfib`, exhibits the best overall speedup of 77 on 100 cores. All other benchmarks still scale up to 100 cores, which significantly exceeds that of a single multi-core machine. Other programs show low speedup; i.e. the large programs which have more complex data dependency, and more synchronisation result in greater overheads to switch between different threads. Nevertheless, they show greater improvement as we show in the following sections discussing different load balancing policies and their effect on performance.

4.3.1 Low-Watermarks for Pre-Fetching

Our *low-watermark* mechanism was designed to improve the load distribution on hierarchical networks, over the default load distribution policy (Section 3.5.4.1). The original version was designed for flat, single-core networks [105]. In a flat network, passive load distribution, i.e. sending a FISH message when a PE becomes idle (Section 3.5.1) is effective for distributing work. The danger of sending work (pro-)actively is the potential for a drastic increase in the total amount of

communication, because of unnecessary movement of work away from its input data. However, in a hierarchical system, comprised of multi-core nodes, the default mechanism acquires the amount of work necessary to feed all the cores only very slowly, as shown by the data below.

To visualise the load balance throughout the execution, Figures 4.3 and 4.4 show the per-PE profiles of activities in terms of the number of running threads in the `mandelbrot` program, with and without the *low-watermark* (note the different x-scales in both graphs). To clarify, dark green is good utilisation, light green is low utilisation, and red is idle time. The utilisation numbers show percentages of time, not including the time spent on garbage collection over the total runtime; therefore, they do not approach the maximum of 500% for executions running on 5 cores. A per-PE profile shows PEs on the y-axis and, time on the x-axis. This configuration shows 16 bars, representing the 16 PEs used in the run. For each PE, a total of 5 HECs is used, comprising 80 cores in total. The darkness of the green value at each point in time shows the utilisation (i.e. the number of running HECs), as an average over a fixed time window. A utilisation lower than 6% is shown as a red area, representing idle time.

The last line in the profile summarises the range of average utilisation across the PEs. In the concrete per-PE activity profile in Figure 4.3, we observe that PE1 has considerably more work (dark green) and higher utilisation than the other PEs, which only have sufficient work to keep one HEC busy (light green). The average utilisation on PEs 2-16 confirms this behaviour, as it ranges between 40% and 49%. Moreover, Figure 4.3 shows several periods of idle (red) time. The main reason for this behaviour is that `mandelbrot` is data-parallel, whereas the gateway HEC of PE1 is the only one generating sparks when the execution starts. Other PEs send FISH messages, asking for work from PE 1. Using a pure work-stealing load distribution policy in Figure 4.3 will only lead to the receipt of one spark each time a work-requesting message is sent. Therefore, the imported spark is executed by the gateway HEC, but the other HECs mostly remain idle, resulting in a delay picking up sufficient work to keep all 5 HECs on each PE busy. Thus, the average utilisation of PEs 2–16 remains below 50% of the 500% possible in this execution of 5 HECs.

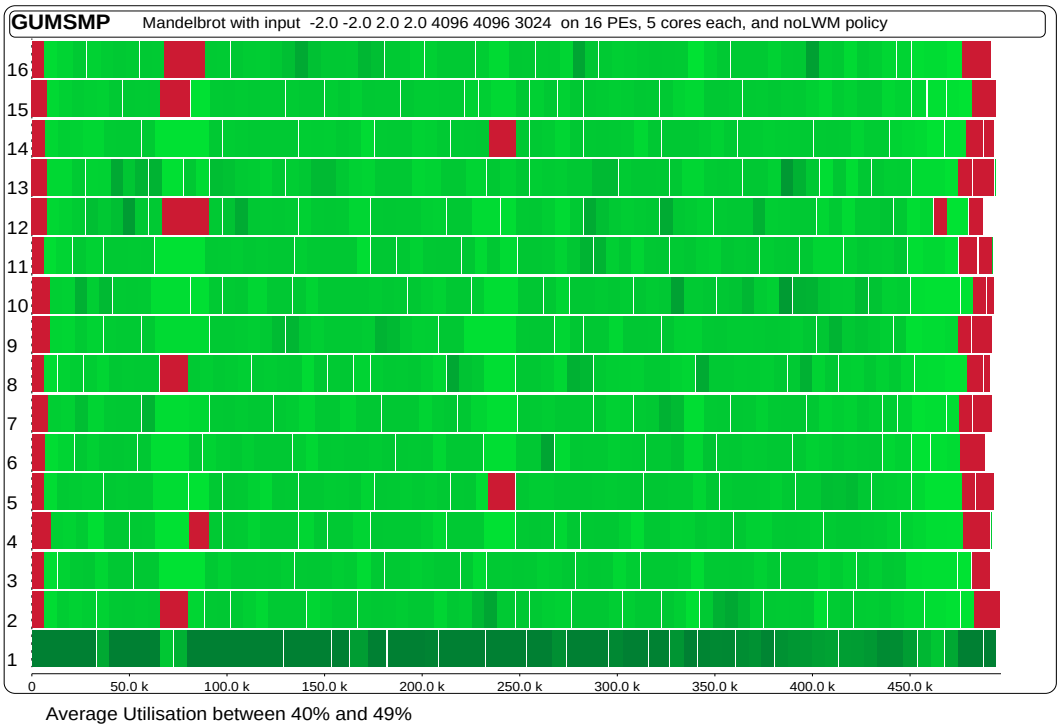

Figure 4.3: Mandelbrot load distribution without low-watermark on GUMSMP

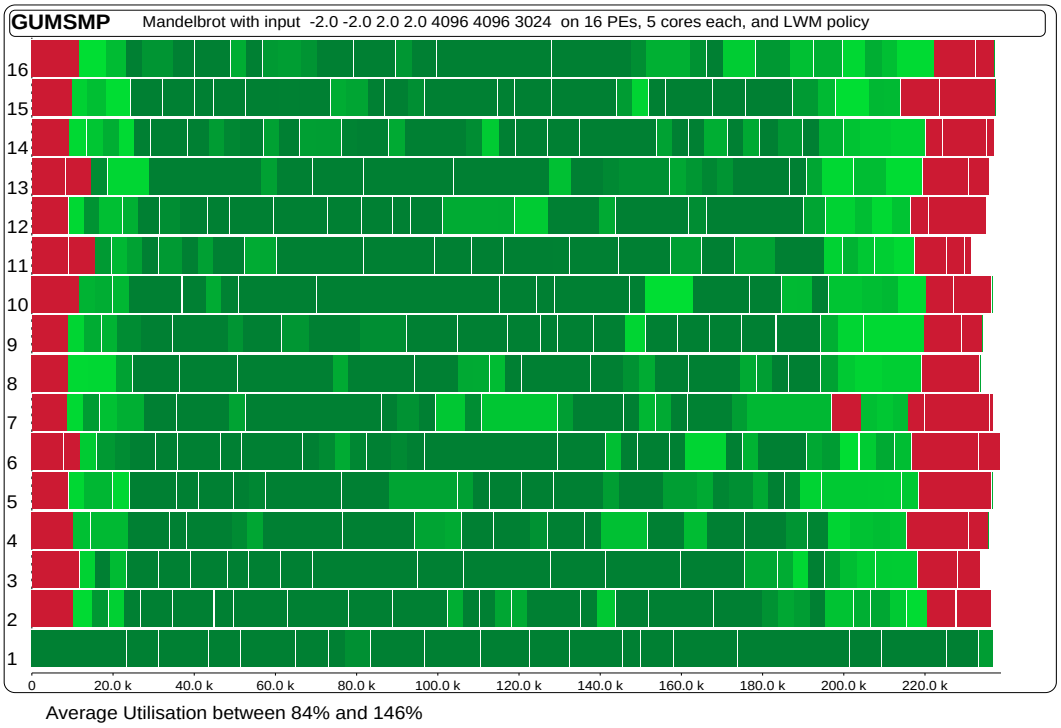

Figure 4.4: Mandelbrot load distribution with low-watermark on GUMSMP

In contrast, Figure 4.4 illustrates the behaviour when the *low-watermark* is enabled, which is set at 5 to match the number of HECs; whereby, the other PEs continue sending messages requesting work until the number of sparks in all the local spark pools reaches the *low-watermark*. Thus, the average utilisation for the other PEs is significantly higher, typically between 84% and 146%, shown as darker green. Although some PEs are unused toward the end of the computation, the high utilisation over most of the execution results in halving the runtime, from 496s to 238s (a drop of 52%).

Mandelbrot is a larger benchmark with a large GC time of approximately 13% of the total execution time compared with programs such as `sumEuler`, where the average GC time is about 5% of the total execution time as we later show in Section 5.3.4.1. The `sumEuler` program, which performs less GC shows a higher average utilisation when enabling the use of the *low-watermark*, where utilisation ranges between 260% to 315% out of the 500% possible for 5 HECs.

In summary, the *low-watermark* policy enables pre-fetching of work, so that the spark pools reach the level of the specified *low-watermark*, which is typically set to the number of HECs available on a single PE, and therefore only depends on the architecture's static parameter. This results in a swifter distribution of the available parallelism throughout the computation, and, in turn, leads to a higher utilisation of the PEs, as summarised at the bottom of the per-PE graphs.

The *low-watermark* policy applies to all the PEs with the exception of the main PE, because it is less likely to require pre-fetching in order to remain busy. Conversely, in the shut-down phase of the execution, parallelism is scarce, and so the withholding of sparks starves the other PEs of work. More desirable than entirely disabling the *low-watermark* mechanism on the main PE would be adjusting its value dynamically, depending on the current system's load. Ideally, we want to decrease the value when the load drops. The current implementation does not provide the necessary information for this kind of monitoring yet. However, we discuss potential improvements in this regard in future work (Section 6.2.1.3).

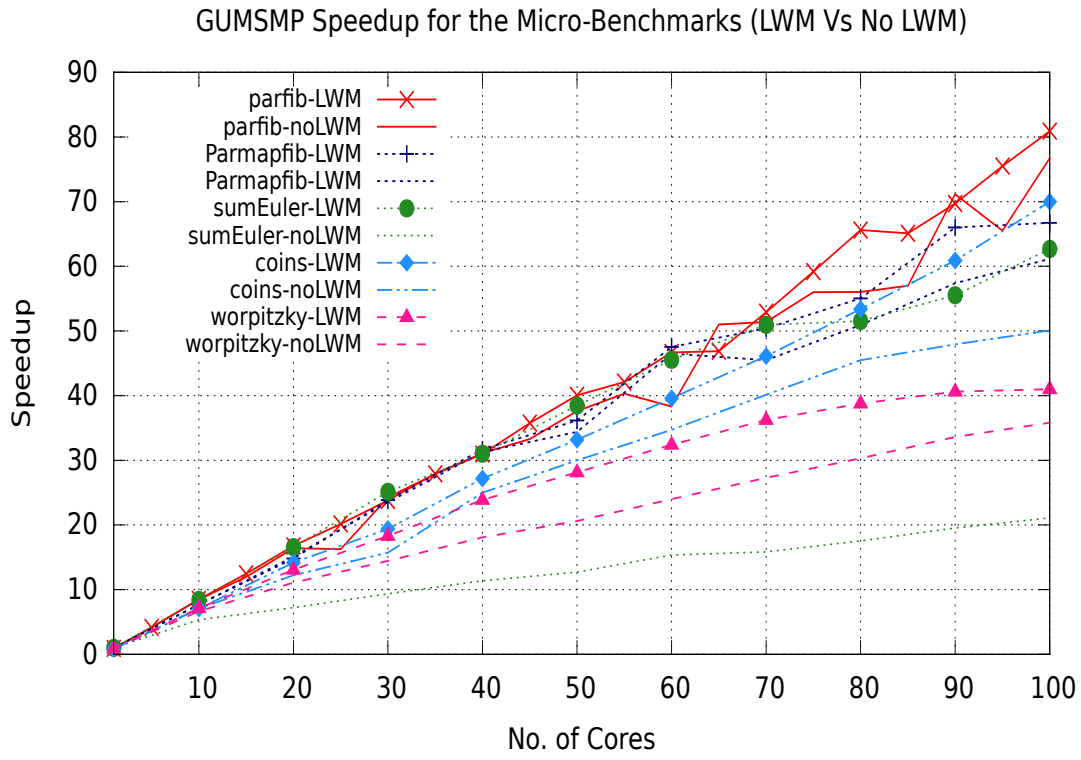

Figure 4.5: Speedup of smaller benchmarks with(out) low-watermarks

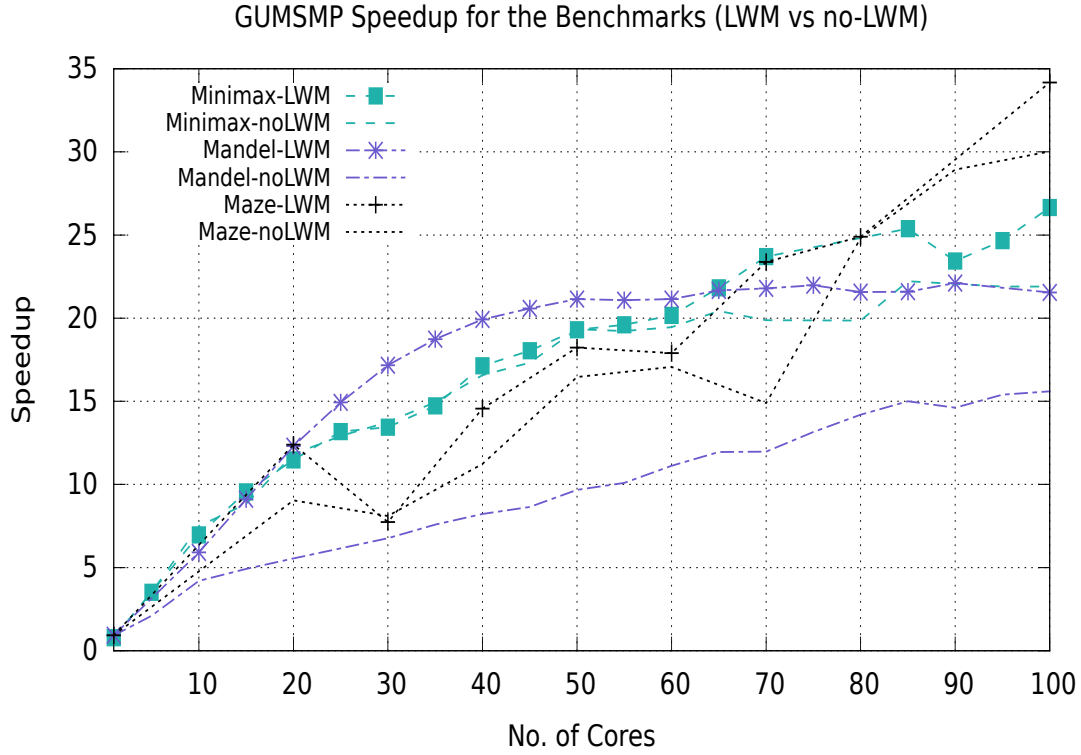

Figure 4.6: Speedup of larger benchmarks with(out) low-watermarks

To assess the impact of the *low-watermark* mechanism on all programs; Figure 4.5 compares the speedups for the programs using a *low-watermark* tailored to the number of cores (ticked plots), with the speedups in a setting without the *low-watermark*, and using the default passive load distribution mechanism (unticked plots). This comparison shows that the *low-watermark* mechanism consistently improves performance, by a factor of up to 3 in the case of `sumEuler`, as Table 4.5 demonstrates.

This behaviour is underlined by the results from the larger benchmarks, as shown in Figure 4.6. All three benchmarks exhibit a consistently improving speedup when using the *low-watermark*, with improvements between a factor of 1.1 and 1.3 on 100 cores. This reflects that, across a range of core numbers, the improved load balance and the lower number of idle periods throughout the computation, as shown in the per-PE profile in Figure 4.4.

Table 4.5: Summary of the improvement of *low-watermark* mechanism on 100 cores

	Runtimes (seconds)		**Speedup**		$\frac{LWM}{NoLWM}$
Programs	No LWM	LWM	No LWM	LWM	
`parmap-of-parfib`	76.4	70.0	61.2	66.7	1.1
`parfib`	90.0	85.4	76.7	80.9	1.1
`sumEuler`	115.4	38.8	21.0	62.6	3.0
`coins`	52.5	37.6	50.1	70.0	1.4
`worpitzky`	58.7	51.2	35.8	41.0	1.1
`minimax`	36.2	29.7	21.9	26.7	1.2
`mandelbrot`	317.6	230.0	15.6	21.5	1.3
`maze`	95.6	84.0	30.0	34.2	1.1
Min.					**1.1**
Max.					**3.0**
Geom Mean.					**1.4**

In terms of scalability, all the micro-benchmarks scale well up to 100 cores, significantly exceeds that of a single multi-core machine. The simplest micro-benchmark, `parfib`, exhibits excellent overall speedup of 81 on 100 cores.

For the data-parallel `sumEuler` program, the speedup shows variations over

an increasing number of cores. This is mostly because of the amount of parallelism being fixed (i.e. the number of blocks of data items being processed), which means that for higher core numbers there is a greater risk of load imbalance occurring towards the end of the execution. In general, the RTS is designed to handle parallelism dynamically and adaptively; therefore, divide-and-conquer programs that generate a significant amount of parallelism throughout their execution show better scalability than data-parallel programs. Furthermore, in the case of `sumEuler`, it is crucial to use a low-watermark mechanism to achieve a speedup of 63 on 100 cores, as shown in Figure 4.5. The the `worpitzky` program delivers lower speedup. This divide-and-conquer program generates lower and more irregular parallelism, compared with `parfib`.

As anticipated, the speedups for the larger benchmarks are lower than those for the micro-benchmarks, ranging between 21 for `mandelbrot` and 34 for `maze`. The larger benchmarks involve significantly greater data transfer in terms of the size of the total data communicated, resulting in higher communication overheads (i.e. up to 48MB for `mandelbrot` as opposed to 0.01MB for `sumEuler`). Evidence of this higher overhead is illustrated in Figures 4.7 and 4.8 which demonstrate the communication in terms of the number of messages and the size of data communicated. Even though, smaller benchmarks such as `parfib` and `parmap-of-parfib` communicate large number of messages (as demonstrated in Figure 4.7), those messages are smaller in size. On the other hand, the large benchmarks which use larger data structures, the amount of graph communicated is between 1.2Mb for `maze` and up to 48Mb for `mandelbrot` (as demonstrated in Figure 4.8).

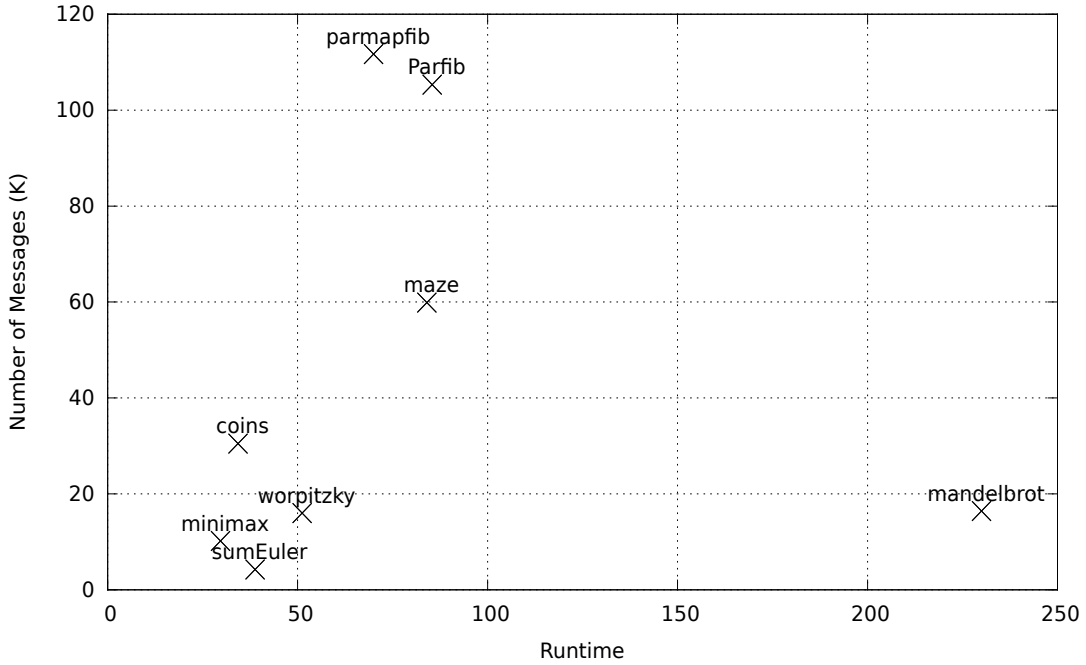

Figure 4.7: The number of messages communicated on 100 cores

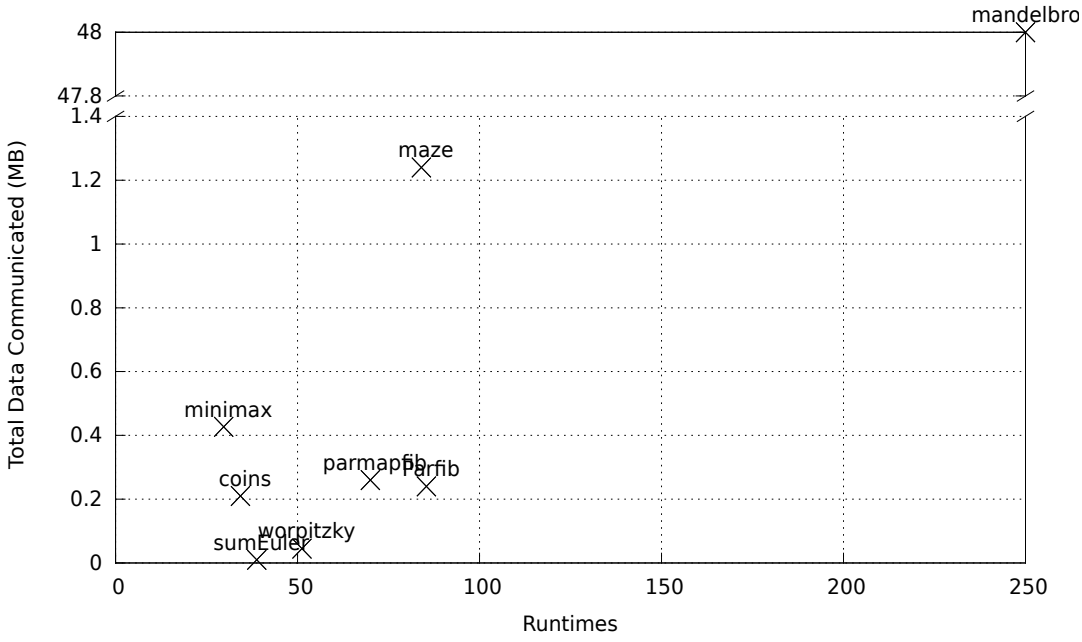

Figure 4.8: The total data communicated on 100 cores

Of the larger benchmarks (`minimax`, `maze`, `mandelbrot`), `maze` shows the best scalability, with a speedup of up to 34 on 100 cores. In addition, it can further improve with the use of active work distribution (Section 5.3.6). Moreover, in

terms of scalability, it can scale well, beyond a single cluster (Section 5.4).

The performance of `minimax` delivers a speedup of 27 (the speedup increases further to 31 when enabling inter-node sparks in Section 4.3.2). However, With this program, the speedup tails off with high core numbers, due to the aforementioned overhead. This is not an inherent limitation of the system, however; as the `mandelbrot` benchmark shows; it still scales, but instead has a flatter slope than the micro-benchmarks.

These results show that even without any specific tuning for a large-scale hierarchical architecture, it is possible to deliver speedups on up to 100 cores, well beyond what has previously been reported for GpH benchmarks, which can be seen as a contribution to the GpH performance evaluation in itself. The amount of data exchange required for these programs is substantially higher, and this factor limits the speedup. The implementations of these benchmarks themselves were originally developed for flat moderate size clusters, and tested on up to 32 nodes; however, they have not been further tuned to the hierarchical configurations used in this thesis.

4.3.2 Asymmetric Load Distribution Policy

Using the same load distribution policy, both on the inter- and intra-node level, carries the danger that local HECs may steal large-grained parallelism, which would be more productively executed on another PE in the network. Therefore, we choose different policies at different levels, which we term *asymmetric load distribution*.

The `parmap-of-parfib` and the `minimax` benchmarks profit most from this behaviour. In both cases, there is a poor saturation with parallelism across PEs with other PEs only picking up the small (nested) computations generated by large computations. In order to address this issue, we use an asymmetric load balancing policy to block intra-node spark exchange in the start-up phase of the parallel execution. This prevents other HECs from picking up work on the main PE, which would then monopolise the parallelism on the main PE. Specifically, we ensure that we send out at least n sparks to other PEs, before the other HECs on the main PE are permitted to pick up work. This refinement of the default

load distribution policy accounts for the multilevel structure of the architecture, initially favouring inter-node spark exchanges; making it possible to achieve the large-scale distribution of large work, causing significant improvements to the performance of programs with nested parallelism.

To quantify the impact of this policy, Figure 4.9 presents the speedups for the `parmap-of-parfib` and `minimax` using the *low-watermark*, and a combination of *low-watermark* and favouring inter-node spark distribution. The inter-node sparks consistently delivers better results. For the `parmap-of-parfib` micro-benchmark and for the `minimax` benchmark, we observe an improvement of up to 19%, all these are on a 100 core configuration. We expect this policy to be generally beneficial for programs with nested parallelism, where the outer parallelism should be off-loaded to another node.

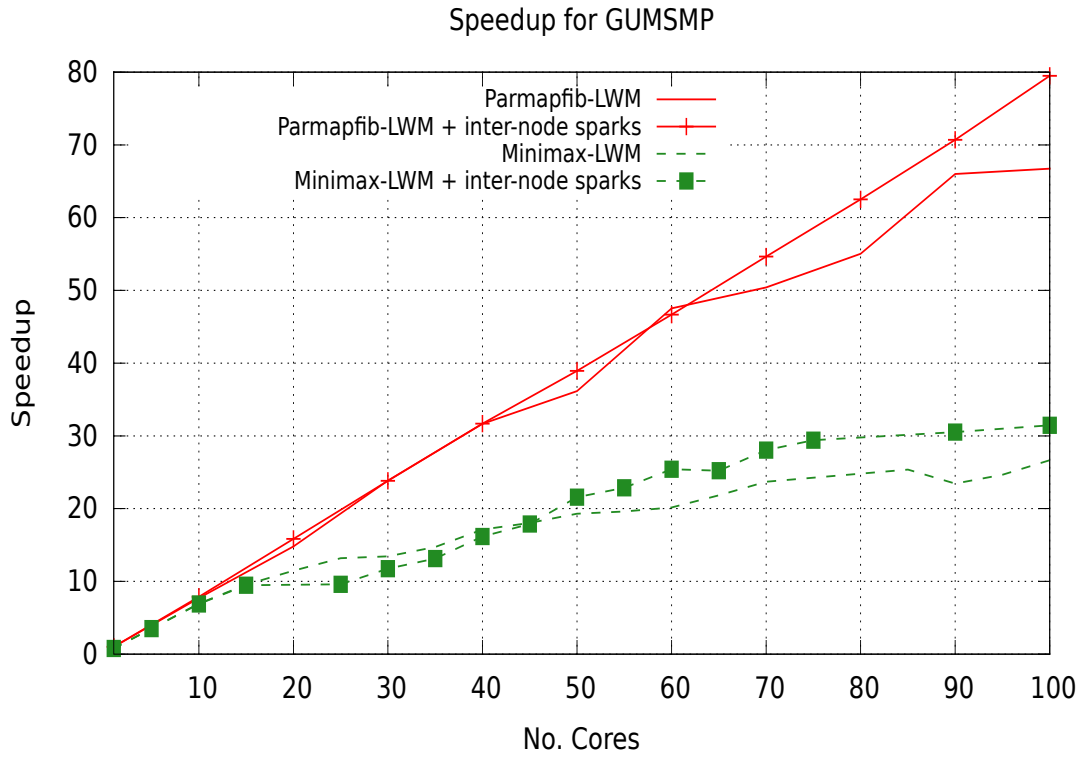

Figure 4.9: Speedup of using an asymmetric load distribution policy, enabling *inter-node sparks*

For `parmap-of-parfib`, we observe a large variability in the runtimes of up to 26% for the setting where no *low-watermark* is used, compared with the use of the asymmetric policy combined with the *low-watermark* (i.e. runtimes

between 66.3 seconds and 90.5 seconds on 90 cores). This behaviour represents a time issue in terms of selecting a spark for sending remotely compared with local stealing. If the gateway HEC is faster in responding to the work request messages and sends the large spark before it is picked up by local HEC, then a faster execution is observed. In contrast, asymmetric load distribution policy guarantees that the larger sparks are sent remotely, and hence provides both good and more predictable performance.

For non-nested parallelism, however, it is not necessary to provide the intra-node load distribution; this then risks increased idle time during the start-up phase of the parallel execution. Indeed, no improvement is observed for `sumEuler` in excess of the *low-watermark* mechanism previously discussed.

4.3.3 Distinguishing Local and Global Work

A general problem with virtual shared heap is the *fragmentation* of the heap, where related data structures reside on different nodes. This increases the communication required to obtain the remote data structures; thereby affecting the data locality and degrading the overall performance. With GUMSMP, imported sparks are added to the spark pool of the gateway HEC (Section 3.5.3.3). Since the distribution of work between the HECs on one multi-core is fairly cheap, this does not cause problems from a load balancing perspective. However, mixing local and imported sparks in the same pool can prove problematic in terms of heap fragmentation, leading to more inter-node pointers, and thus more communication. Moreover, the *low-watermark* mechanism we introduce to improve the performance of GUMSMP leads to an increase in communication as a side effect in order to improve the average utilisation, but at the same time it may contribute to the heap fragmentation.

As a metric for heap fragmentation, the size of the GIT table is used (GA residency). GA is a concept inherited from GHC-GUM, and used to refer to the remote graph structure sent to a remote PE as discussed in detail in Section 3.6.1. In this section we present a case study, focusing on `parfib` as an example of large average GA residency: on execution of input 52 with threshold value of 23, on 120 cores, exhibits an average GA residency of 7000.

Spark segregation is a new concept we introduce in GUMSMP as an attempt to preserve the data locality, and hence improve overall performance, by reducing heap fragmentation. It is based on the implementation of a separate spark pool, dedicated to imported sparks, to ensure related pieces of work are kept together in one pool.

With spark segregation (unlike with default settings), sparks are not treated equally; a distinction is made between sparks that are imported from a remote PE in response to work request messages *(global)*, and sparks that are locally generated by threads in the same PE *(local)*. Based on this distinction, the concept of an import-spark-pool is implemented as a separate spark pool for each PE. In this implementation, whenever a spark is received from a remote PE, that spark is retained in the import-spark-pool. The other local spark pools, for the local HECs within a PE, maintain the local sparks; i.e. sparks generated locally. We then introduce different mechanisms for local evaluation of sparks, as well as for sending sparks remotely, as Table 4.6 demonstrates.

Table 4.6: Policies for exporting and selecting sparks when using the import-spark-pool.

	Spark Select Policy (for local evaluation)	**Spark Export Policy(for exporting sparks)**
local only	*select* local spark only, do not search in the import-spark-pool.	*export* local spark only, do not search in the import-spark-pool.
prefer local	prefer local sparks to *select*, then look for global sparks in the import-spark-pool.	prefer local sparks to *export*, then look for global sparks in the import-spark-pool.
prefer global	prefer global sparks to *select* from the import-spark-pool, then look for local sparks.	prefer global sparks to *export* from the import-spark-pool, then look for local sparks.
global only	*select* global spark only, do not search in any local spark pool.	*export* global spark only, do not search in any local spark pool.

The combination of those 2 policies, each with 4 possible values for exporting and local evaluation resulted in 16 different combinations. It is evident that, some policies are not practical such as export global only for divide-and-conquer programs, or export local only for data-parallel programs, as those might lead to a heavily imbalanced load and thus, poor performance. Therefore, we restrict our study to the following combinations of policies which are compared with the default setting, when the import-spark-pool is not in use:

1. Export: prefer local, Select: prefer global. Rationale: improve response time, by preferring global work for local evaluation, so that the result is ready when asked for.

2. Export: prefer global, Select: prefer local. Rationale: improve data locality, and avoid sending away local work.

3. Export: prefer local, Select: prefer local. Rationale: study the effect where neither data locality, nor response time is considered.

4. Export: local only, Select: prefer global. Rationale: as (1) but more aggressive, i.e. never send away global work.

5. Export: local only, Select: prefer local. Rationale: as (3) but more aggressive, i.e. never send away global work.

We study the effect of these different policies on average GA residency (as a measure of heap fragmentation), and overall performance (as a bottom line metric). We expect some mechanisms to work better with a different parallelism paradigm, e.g. with divide-and-conquer we expect to find more dynamically generated, local sparks. In this case, when remote sparks arrive at a time when more local sparks are being generated, then with the default mechanism, the global spark might be re-exported to another remote PE, thus increasing the danger of heap fragmentation as a consequence of forwarding the work, and thereby affecting the response time. With spark segregation, by keeping the import spark in the import-spark-pool, we expect to improve heap fragmentation by using the prefer global policy for local evaluation, and the prefer local policy for exporting sparks remotely.

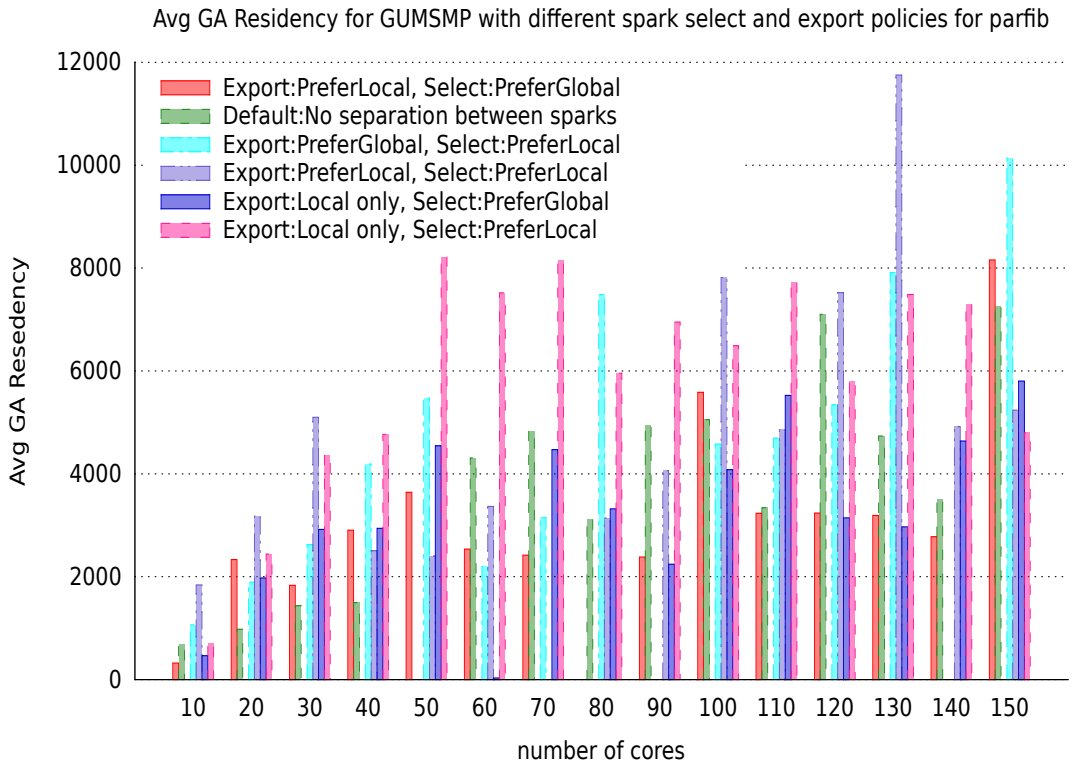

Figure 4.10: Average GA residency for different policies.

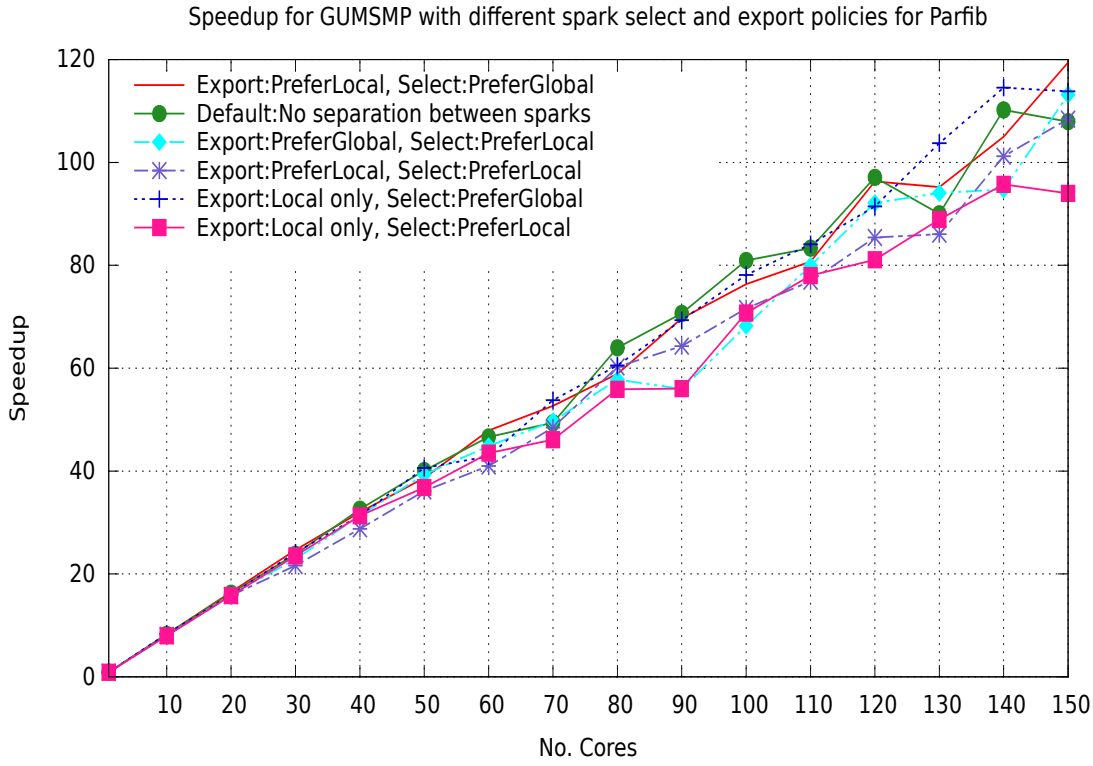

Figure 4.11: Parfib speedup for different policies.

Unfortunately, the import-spark-pool mechanism is not effective at reducing heap fragmentation or improving the overall performance. As demonstrated in Figure 4.10 for average GA residency for `parfib`, it is apparent that there is large variation in terms of average GA residency (i.e. the highest difference between the lowest and the highest average GA residency on 130 cores is 295%). However, policies based on *preferring local sparks for local evaluation* exhibit a larger average GA residency in general, and therefore a lower performance, as illustrated in Figure 4.11. Specifically, the three policies that are based on selecting local sparks for local evaluations (Export: prefer global, Select: prefer local, and Export: prefer local, Select: prefer local, and Export: local only, Select: prefer local) show the highest average GA residency in all cases (Figure 4.10), and lowest performance (Figure 4.11).

On the other hand, policies based on *preferring global sparks for local evaluation* exhibit lower GA residency, and therefore deliver a competitive performance to the default mechanism. Specifically, the two policies that are based on selecting global sparks for local evaluations (Export: prefer local, Select: prefer global, and Export: local only, Select: prefer global) show the lowest average GA residency in all cases (Figure 4.10), and competitive performance (Figure 4.11).

In fact, different factors affect overall performance, such as data locality, response time, and heap fragmentation; therefore, improving one factor (such as lower heap fragmentation) might not directly deliver an improvement to overall performance.

4.3.3.1 Future Spark Segregation Work

We plan in future work to take spark segregation mechanism further, and to investigate different ways to gain concrete improvements in this direction. As part of our plan for auto-tuning the RTS (Section 6.2.1.2), we could investigate the possibility of auto-tuning different policies for exporting or selecting sparks, e.g. during the system start-up phase preferring a global spark for evaluation to distribute work quickly when the execution starts, but to then later during the main execution switch to preferring local, to attempt to improve locality. Different factors require further investigation, such as the suitability of a specific

policy for the parallel paradigm of the application, as well as considering the fact that the success rate of policies differs, therefore specific policies designed to improve specific aspects might not be effective, because of a lower success rate; e.g. for the Export: prefer global, Select: prefer local, we demonstrate the success rate for export and select as in the Table 4.7.

Table 4.7: Different cases for the Export:prefer global, Select:prefer local policy

	Export: prefer global	**Selection: prefer local**
Success	exporting global spark.	selecting local spark.
Failure	nothing exported, forward work request.	no local or global spark found.
Fall-back	exporting local sparks.	selecting global sparks.

Moreover, we could also consider combining the spark-tagging approach with the use of the import-spark-pool. Therefore, sparks in this pool might then be further annotated, according to their originating PE, or level in hierarchical architectures, thereby providing a topology-aware mechanism, or possibly by the user providing encapsulating information on a desirable co-location. In such cases, once imported sparks have been turned into threads, the scheduler may prefer other imported sparks that have some affinity with the previous one, e.g. those coming from the same PE.

The RTS infrastructure i.e. the import-spark-pool can be a basis for further work exploring a concrete combination of spark selection and exporting policies.

4.3.4 Dedicated Gateways

In the design of GUMSMP, one HEC in each node serves as a gateway to the remaining HECs, which means communication is restricted to this HEC, which also performs a computation task (Section 3.5.3.1). In this section, we investigate an alternative design, where the gateway HEC is restricted to performing communication tasks only, with no computation, and where the system relies solely on other HECs for computation, aiming to speed up the communication, despite sacrificing some computational capacity. The evaluation of such a design on the

multi-core reveals a *drop* in performance as a direct effect of losing one computation engine in each node, i.e. up to 20 HECs in a configuration of 20 nodes. This is obviously a large sacrifice, but as the trend for parallel machines is moving towards many-core, with up to 64 core, such a configuration might be useful for reducing response time to messages from a remote PE, as well as for making work available to keep local HECs busy with computations. Thus, it could aid scalability. However, as we discuss later in the NUMA evaluation, such configurations suffer from memory management overheads. Table 4.8 demonstrates the effect of using dedicated gateways, which shows an average 18% performance degradation as a result of losing 20 HECs. However, we observe an improvement of 11% for `mandelbrot` as a communication bound program which does a significant large data transfer as shown in the prior Figure 4.8. While `maze` also communicates large data, the granularity of `mandelbrot` computations is significantly lower (i.e. about 4K sparks compared to 33K), which results in large communication-to-computation ratio for `mandelbrot`, and hence it benefits from the dedicated gateway providing faster communication.

We speculate that in general for communication bound programs, dedicated gateway HEC can lead to an improvement.

Table 4.8: The effect of using dedicated gateways

100 cores			
Program	**Runtime (dedicated gateways)**	**Runtime (no dedicated gateways)**	**Time increase (%)**
`parfib`	112.5	94.7	18.9
`sumEuler`	41.5	36.6	13.3
`coins`	44.4	34.3	29.4
`worpitzky`	54.8	47.6	15.1
`maze`	100.7	70.3	43.2
`mandelbrot`	209.5	234.1	-11.7
Geom Mean.			18.0

4.3.5 Optimising the Number of Cores Per PE

GUMSMP is designed for hierarchical architectures like clusters of multi-cores, or large NUMAs; where the system can use a shared heap on each node and distributed heaps across nodes. But how many cores in each node should be used? Section 5.3.3 investigate how to optimise the number of cores for each PE.

4.3.6 Optimising the Setting of the Allocation Area

We shall see that in Section 5.2.2, memory management represents the main bottleneck affecting the performance of GUMSMP as a component inherited from GHC-SMP. In particular, the main issue identified in that study is having several cores per PE sharing the same available heap, and producing large live data sets, which leads to an increase in the GC time. By providing a larger allocation area for each PE, reflecting multiple executions trying to access data, we aim to reduce the GC time and improve performance. Section 5.3.4, reports an investigation of the effect of increasing the allocation area on performance.

4.3.7 More Active Load Management

While GUMSMP mostly provides passive load management, this could be combined with some active load management at both inter- and intra-node levels. In such a setting, each idle HEC will steal sparks from the pools of other local HECs, following the passive load distribution approach, but additionally a HEC with a full spark pool will actively push sparks to the spark pools of other HECs, thus combining the passive and active load distribution at the intra-node level. For inter-node level, this is equivalent to the use of *low-* and *high-watermark* mechanisms (Section 3.5.4.1) to actively push sparks to remote PEs. The rationale for this combined policy is to improve the load balance on architectures with low communication costs, and to be pro-active in sending work to idle HECs at the intra-node level, and to remote PE at the inter-node level. Section 5.3.6 provides an investigation of such a combined policy at the intra-node level and discusses its effect on the performance of programs with a large degree of parallelism.

4.4 Summary

We presented the performance results for our GUMSMP system. We assessed the performance of several tuning policies and explored the performance of some of alternative design choices, as discussed in the previous chapter. We concluded that large, hierarchical architectures require a more aggressive work distribution policy than flat networks. A primary optimisation is a refined work-stealing policy, which applies the concept of a *low-watermark* tailored to the number of cores per-node, allowing the system to *pre-fetch work* at the node level. This proved to be crucial for the performance of some of the test programs: for the micro-benchmarks, speedup improved by a factor of up to 3, and for the larger benchmarks by a factor of up to 1.3. As an additional refinement, we favoured inter-node load distribution in the start-up phase of the parallel execution; thereby ensuring that early work, which tends to be large, is picked up by other PEs, rather than by other cores on the same machine. This policy significantly improved the load balance of the programs with nested parallelism: on 100 cores, the speedup improved by up to 19%.

The performance results for five micro-benchmarks, and three more communication intensive benchmarks demonstrate the scalability of our multilevel design up to 100 cores, well beyond the size of the individual multi-cores, with absolute speedups of up to 81. We report on scalability results on up to 300 cores in Section 5.4. Our implementation enables the execution of GpH programs on networks of multi-cores, thereby extending previous work on GHC-GUM, and these results represent the first systematic study of GpH performance on the 100 core scale.

The speedups for the three larger benchmarks are relatively low, e.g. between 21 and 34 on a 100 core architecture. This is partly due to the increased amount of communication inherent in the applications, which in turn also increases the overheads for managing the virtual shared heap. However, it is also partly due to having more complex data dependencies, and greater overheads to switch between different threads.

Moreover, we explored alternative designs by demonstrating the effect on performance when restricting the gateway HEC to perform communication only,

which negatively affects overall performance, because of the loss of one computation engine from each node, which cannot offset the gain in communication speed on the core numbers tested. We discussed our implementation of the import-spark-pool as an attempt to reduce heap fragmentation, and to therefore improve overall performance. Our results illustrated that separating sparks into local and global requires further tuning of the policies using more information such as using a spark tagging mechanism in order to be effective.

Chapter 5

GUMSMP Evaluation

5.1 Introduction

This chapter presents a performance evaluation of GUMSMP in comparison to GHC-SMP on a state-of-the-art physical shared memory NUMA machine (published in [8]). Additionally, it provides an evaluation of GUMSMP implementation on a cluster of multi-cores, comparing GUMSMP performance with the performance of GHC-GUM.

This chapter is structured as follows. Section 5.2 discusses the scalability issues associated with the NUMA machines. Section 5.2.1 demonstrates the scalability limit of GHC-SMP on NUMA architectures. Section 5.2.2 demonstrates that a hybrid system, GUMSMP, which combines both distributed and shared heap abstractions, consistently outperforms the shared memory GHC-SMP implementation. Section 5.3 evaluates GUMSMP on a cluster of multi-core machines, comparing its performance with GHC-GUM. It compares the two systems in terms of the amount of parallelism exploited, the threads created, and the communication volume. Sections 5.3.3 and 5.3.4 discuss further tuning for the memory management component of GUMSMP. We then show the performance of GUMSMP and GHC-GUM for data-parallel and divide-and-conquer programs on up to 128 cores. Section 5.4 provides scalability results for a range of benchmarks on up to 300 cores, by combining multiple clusters, with relatively low latency.

5.2 Balancing Shared and Distributed Heaps on NUMA Architectures

Current high-end servers offer 48 or 64 cores with a NUMA architecture that supports shared memory access across the entire address space. On such architectures, reduced synchronisation costs are bought at the price of memory latencies, which vary by a factor of up to 2.2, depending on in which NUMA region the memory bank is located (Section 2.2.1.1).

The measurements for this section are made on a 48-core NUMA machine, with four AMD Opteron-based processors, one per socket as depicted in Figure 5.1. Each processor contains two NUMA regions, and each region has six 2.8 GHz cores. The total RAM is 512 GB, which is evenly distributed as 64 GB for each region. A 2 MB L2 cache is shared by 2 cores in each region, and a 6 MB L3 cache is shared between all 6 cores within the same region. The machine runs x86_64 Linux CentOS 6.5. Memory latencies on this NUMA architecture vary by a factor of 2.2, as demonstrated in Table 5.1. The set of benchmarks used are the same as those specified in Section 4.2.1. The RTS of the parallel Haskell implementations is based on GHC 6.12.3, using GCC 4.4.7, and PVM 3.4.5 for message passing. For GHC-SMP, the performance of GHC 7.6.3 was tested, delivering similar results.

In our experiments, we choose 40 cores to evenly partition the machine into 2, 4, 5, and 8 regions. Table 5.1 demonstrates the variation in access time between nodes located in different NUMA regions, with 10 being the unit of local access time (measured using the Linux command *numactl -H*). In this example, the total number of cores is 48, located in 8 NUMA regions. If the node to be accessed is located in the same NUMA region, the access time is 10, but if it is located in a different region, the access time increases significantly up to 22, depending on how distant the region is. Even more problematic, for those applications that require frequent memory access, the memory bus can become a major bottleneck, degrading access times far below the values measured on an idle machine. These architectures pose a challenge to parallel languages, especially in cases where they make very dynamic use of memory, as many parallel functional applications do.

Table 5.1: Memory access times between different NUMA regions (10 is the basic unit for local memory access)

node	**0:**	**1:**	**2:**	**3:**	**4:**	**5:**	**6:**	**7:**
0:	10	16	16	**22**	16	**22**	16	22
1:	16	10	**22**	16	**22**	16	**22**	16
2:	16	**22**	10	16	16	**22**	16	22
3:	**22**	16	16	10	**22**	16	**22**	16
4:	16	**22**	16	**22**	10	16	16	**22**
5:	**22**	16	**22**	16	16	10	**22**	16
6:	16	**22**	16	**22**	16	**22**	10	16
7:	**22**	16	**22**	16	**22**	16	16	10

This section studies the impact of state-of-the-art NUMA architectures on the parallel performance of languages with automated memory management, in particular on GpH. We explore a range of systems, from purely shared memory, hybrid shared/distributed memory, to purely distributed memory. The underlying compiled parallel graph reduction execution model induces both frequent and highly random memory access, thereby aggravating the impact of the NUMA memory architecture. Hence, GpH programs are excellent test cases for exploring the impact of NUMA memory management. Our results, however, are not restricted to Haskell, nor to other parallel functional languages: the issues we explore affect all languages with automated memory management on NUMA architectures.

Moreover, we demonstrate that the scalability of the shared memory GpH implementation (GHC-SMP) is limited by heap contention, due to the synchronisation and locking overheads of the stop-the-world parallel garbage collector, discussed in Section 3.6.2. This limits the number of cores that can be usefully exploited to well below the 48 physical cores available on our AMD Opteron measurement platform. In contrast, our hybrid shared/distributed implementation (GUMSMP) can effectively exploit several distributed heaps on a physical shared memory machine, to reduce both memory contention and heap locking.

We also quantify the impact of heap contention on NUMA servers for the

memory intensive language GpH, and hence identify how parallel Haskell applications can best exploit emerging massively parallel shared memory hardware architectures.

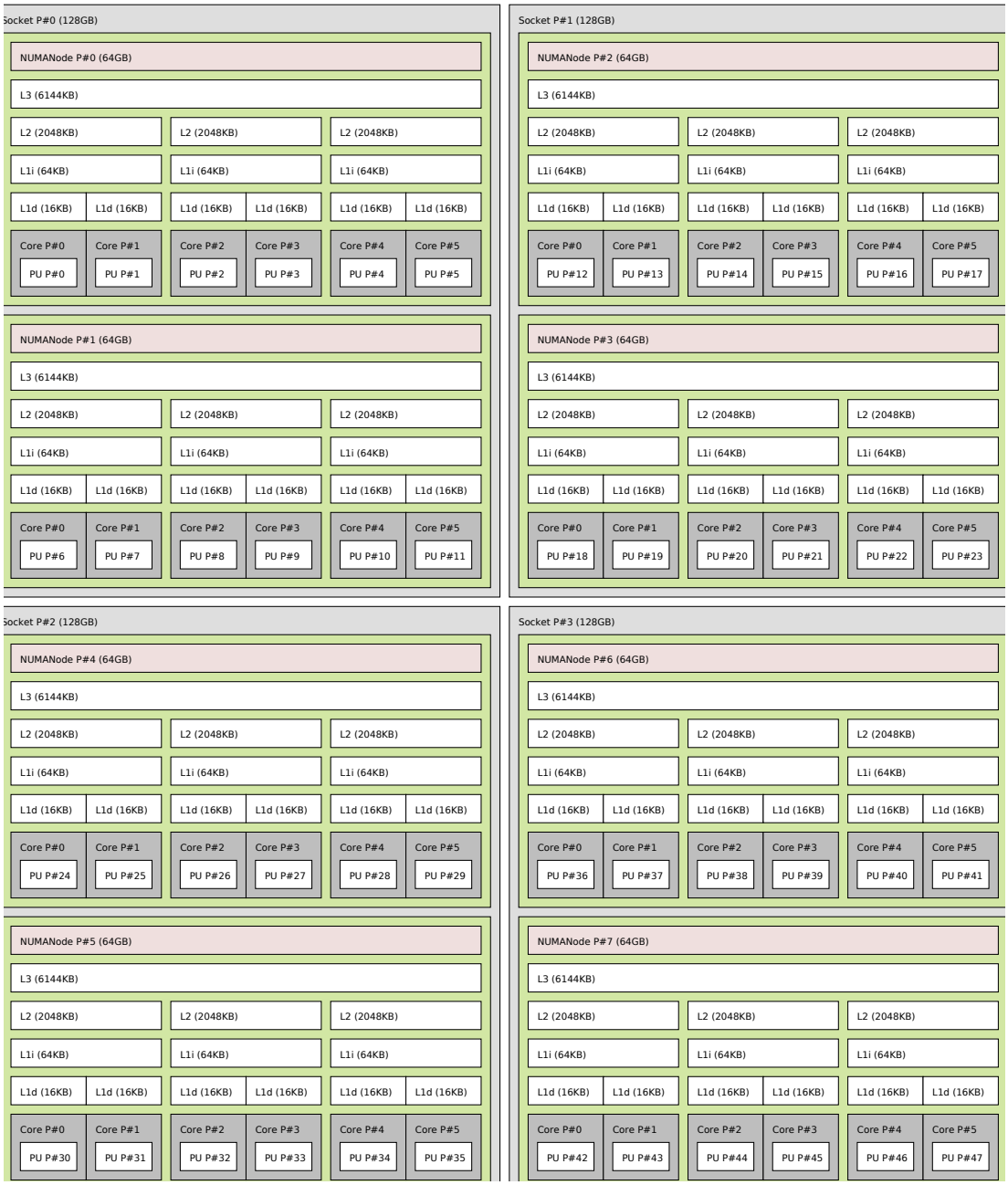

Figure 5.1: NUMA topology of a 48-core server

5.2.1 Scalability Limits

We start the evaluation by comparing the runtimes of GHC-SMP shared memory, and GHC-GUM distributed memory system. Table 5.2 compares runtimes

using the GHC-SMP, with those when using GHC-GUM; with the lowest runtimes per program given in boldface. These numbers reveal a significant degradation in performance for the shared memory GHC-SMP system beyond 15 to 25 cores, while the distributed memory GHC-GUM implementation continues to scale.

Table 5.2: Runtimes for GHC-SMP and GHC-GUM

	Runtimes (seconds)													
Cores	1		15		20		25		30		35		40	
system	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM	SMP	GUM
`parfib`	6004.2	6644.7	741.8	573.4	746.7	406.1	**666.7**	350.4	740.9	296.3	711.9	307.7	752.6	276.3
`coins`	5155.7	5690.7	**829.2**	485.0	857.5	432.9	834.8	384.3	940.4	340.4	1137.5	340.7	1095.1	318.3
sEuler	1507.9	1552.0	199.9	102.8	197.9	94.2	**182.3**	77.9	194.3	81.9	226.1	81.5	222.0	79.0
worpit	1842.3	1818.3	217.3	173.1	204.9	135.9	187.0	116.5	185.2	111.5	**169.9**	105.4	178.6	108.8
`maze`	3181.9	3289.4	1472.5	675.8	1424.4	505.5	**1404.3**	467.9	1553.9	419.3	1650.9	403.2	1527.9	348.7
mandel	4226.9	3772.6	1163.1	420.0	**631.5**	327.9	801.2	294.9	779.8	303.5	821.6	313.9	882.4	315.4
b-scholes	5133.1	5996.3	542.5	396.3	463.32	326.3	431.8	265.1	**406.9**	245.4	491.6	235.4	596.9	200.4

The program with a low heap allocation rate, `worpitzky`, scales best; i.e. achieving the lowest runtime in a GHC-SMP setting at 35 cores; however, even this program has a lower performance on 40 cores (on an 48-core machine). Meanwhile, `coins` which has a high allocation rate; represents the one with the lowest scalability, as performance starts to drop after 15 cores.

While GHC-GUM starts with higher execution times on 1 PE, it typically outperforms GHC-SMP from ca. 10–15 cores onwards. In consideration of this trend, the remainder of the section is based on a study that assumes there is an intermediate point in the range of the extremes of shared heap GHC-SMP, and distributed heap GHC-GUM with even higher performance.

5.2.2 Benefits of Distributed Heaps

The GUMSMP implementation of parallel Haskell combines the heap models for both GHC-SMP and GHC-GUM. It provides parameters for selecting the number of cores to be used, inherited from GHC-SMP, and for selecting the number of PEs (independent instances of the runtime system), inherited from GHC-GUM (Section 3.4.1).

The figures and tables in this section explore a range of configurations, from a purely shared heap to purely distributed heaps, using the GUMSMP implementation and a total of 40 cores. The columns in Table 5.3 show configurations in the form PE/N, indicating the PE instances of the runtime system, each with its own heap, are spawned, with N cores used in each instance, all accessing the same shared heap. Our goal is to establish a balance between PE instances and per PE core numbers that achieve the best results for this set of test programs. Figures (5.2,5.3, and 5.4) demonstrate the different configurations of cores per PE used in GUMSMP, and Figure 5.5 shows the configuration of GHC-GUM and GHC-SMP.

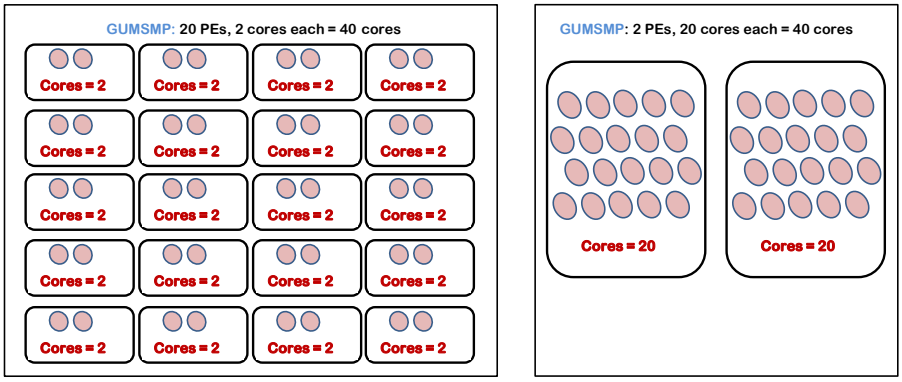

Figure 5.2: GUMSMP (20 PEs, 2 cores each) and (2 PEs, 20 cores each) configurations

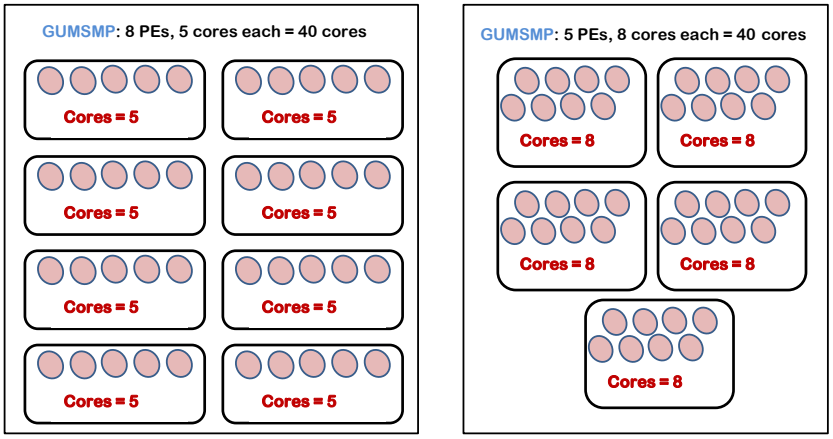

Figure 5.3: GUMSMP (8 PEs, 5 cores each) and (5 PEs, 8 cores each) configurations

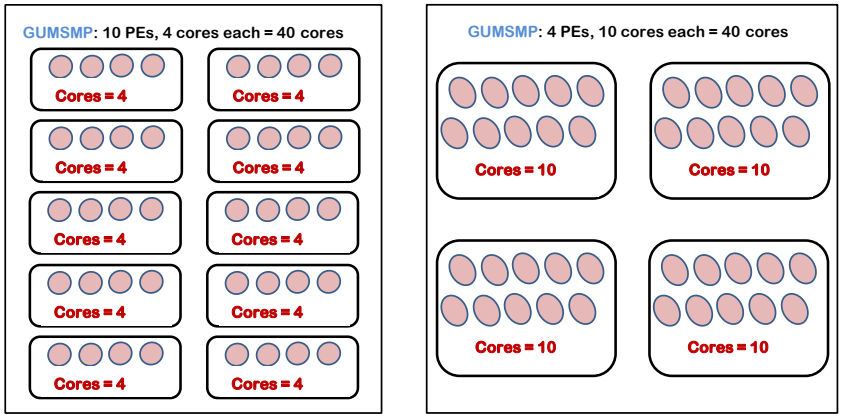

Figure 5.4: GUMSMP (10 PEs, 4 cores each) and (4 PEs, 10 cores each) configurations

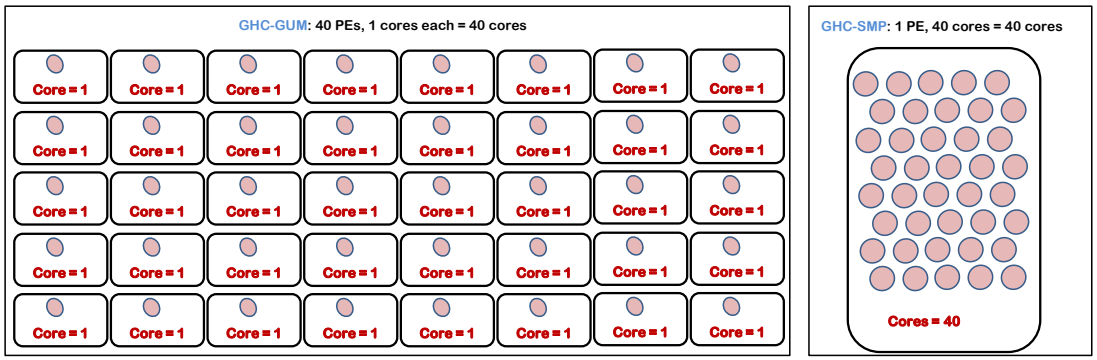

Figure 5.5: GHC-GUM and GHC-SMP configurations

Our main results, the runtimes presented in Figure 5.6, and Table 5.3 (lowest runtimes highlighted), show that for all programs a hybrid of distributed and shared heaps results in the best performance. We conclude that it is best to use up to 5 of the 40 physical cores, resulting in at least 8 separate PEs running simultaneously, one on each of the NUMA regions. With the communication bound `mandelbrot` application, we observe a further small improvement when using 8 cores. Notably, the improvement relative to the pure shared memory execution (GHC-SMP) is most pronounced for `maze` (a data intensive program) and `coins` (a divide-and-conquer program), with runtime improvements up to a factor of 4.5; whereas, improvements for other programs are between 2.2 and 3.3, which are still remarkable.

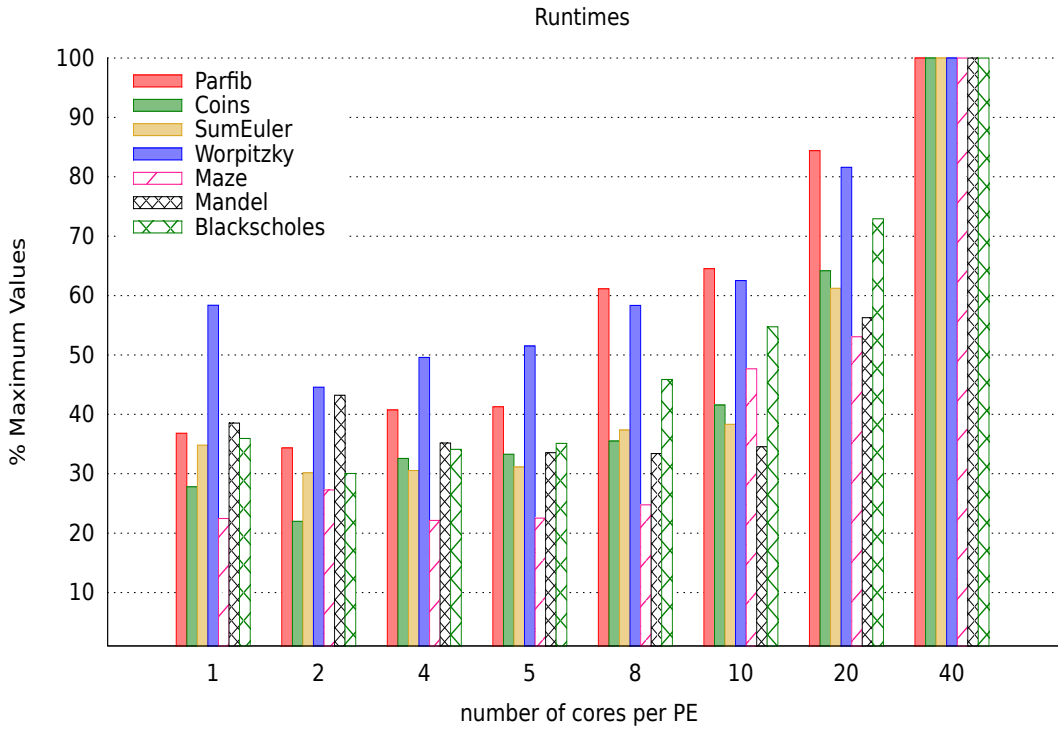

Figure 5.6: Normalised GUMSMP runtimes on 40 NUMA cores

Table 5.3: GUMSMP runtimes on 40 NUMA cores configurations

Configuration	**GUM**	**GUMSMP**						**SMP**	$\frac{SMP\ RT}{GUMSMP\ RT}$
PE/Cores	PE 40	20/2	10/4	8/5	5/8	4/10	2/20	N 40	
`parfib`	276.3	**258.5**	306.6	310.6	460.2	485.7	635.3	752.6	2.9
`coins`	318.3	**240.9**	356.8	364.5	388.7	455.3	702.9	1095.1	**4.5**
`sumEuler`	79.0	**66.9**	67.8	69.1	82.9	85.06	135.9	222.0	3.3
`worpitzky`	108.8	**79.6**	88.5	91.9	104.2	111.6	145.7	178.6	2.2
`maze`	348.7	375.6	**338.3**	344.0	378.3	728.36	810.7	1527.9	**4.5**
`mandelbrot`	315.4	372.5	303.3	**289.4**	**288.0**	297.9	485.1	882.4	3.0
`blackscholes`	200.4	**179.2**	203.6	209.5	273.7	326.8	435.4	596.9	3.3
Min.									**2.2**
Max.									**4.5**
Geom Mean.									**3.3**

To quantify the GC overhead, we present the percentage of GC time relative to the total execution time in Figure 5.7, and the number of synchronisation points that represents the number of locks required to get a new block for allocation during the current parallel stop-the-world GC, in Figure 5.8. There is a strong correlation between this GC percentage, and the runtime, indicating a

loss in performance for high core numbers, mainly due to memory management overheads. Part of this overhead is inherent to the parallel nature of the execution. All the programs typically generate a large number of threads; especially in the case of the shared heap implementation. Each thread defines a set of live heap cells, which need to be retained following garbage collection, as visualised in Figure 5.9, by the larger live heap referred to by the thread pool.

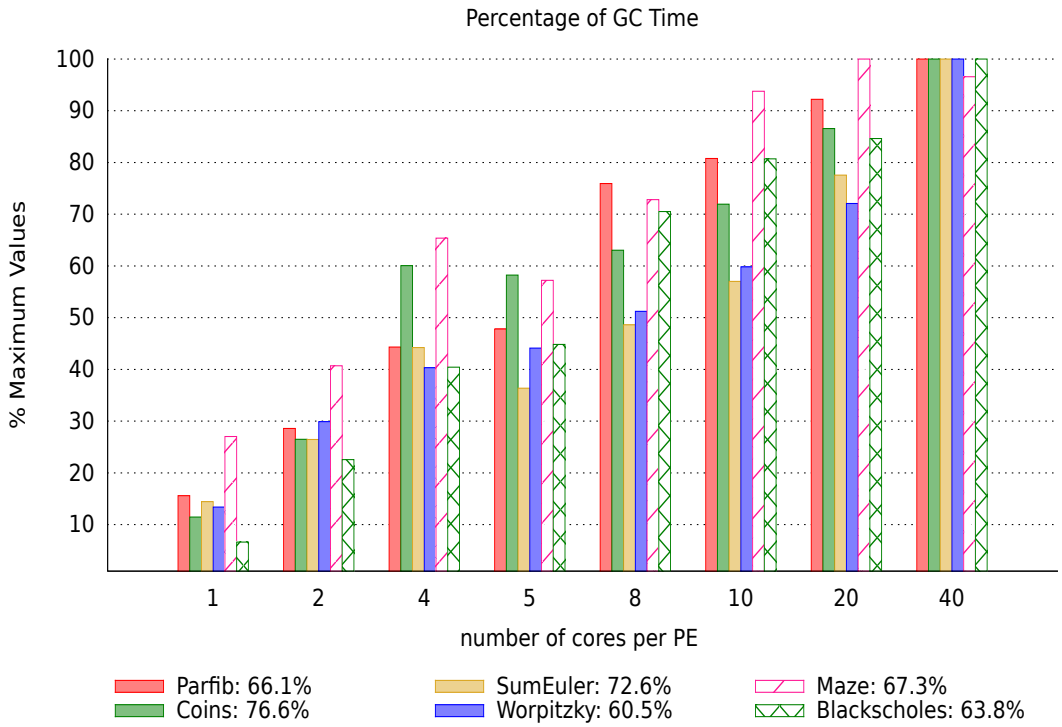

Figure 5.7: Normalised GUMSMP GC percentage on 40 NUMA cores

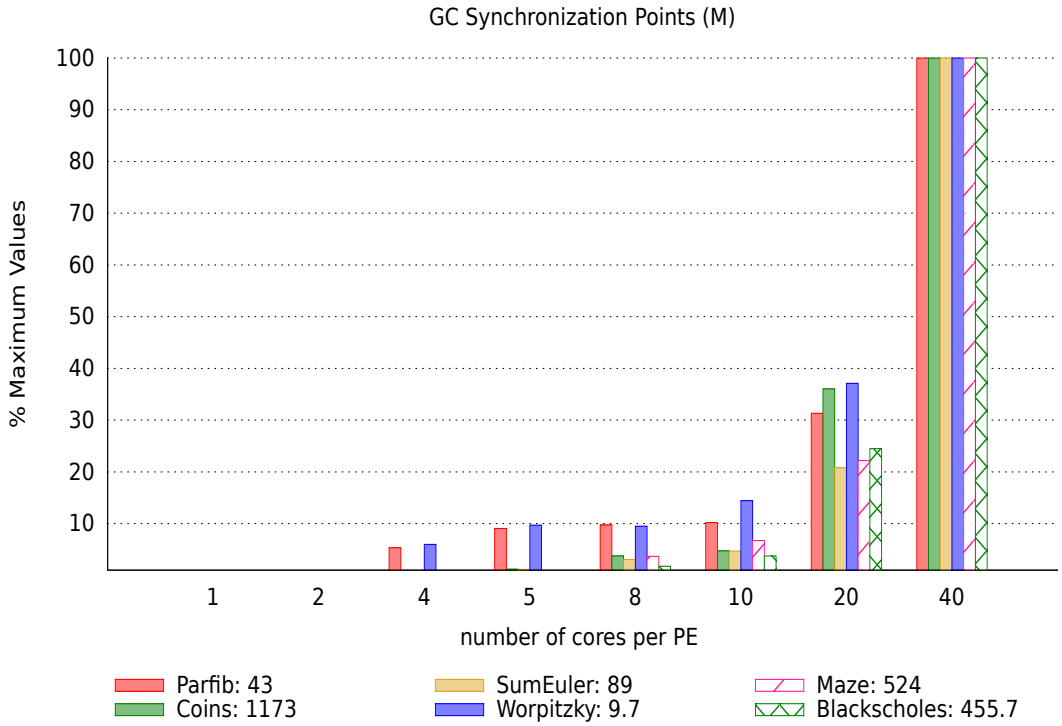

Figure 5.8: Normalised GUMSMP GC synchronisation points on 40 NUMA cores

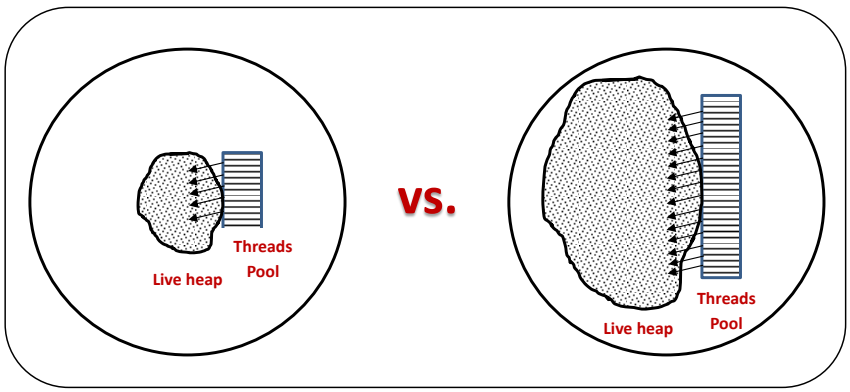

Figure 5.9: Sketch of the GC overheads due to a *large live heap* in a multi-threaded execution

The large amount of live data, which shows up as significantly higher values of memory residency in Figure 5.10, translates into the need for a (currently NUMA-agnostic) garbage collection to perform more work, which represents a major source of overhead. The other major sources of overhead, which are harder to quantify, are the synchronisation to perform the stop-the-world GC, and the

per-object locking required to prevent multiple threads from duplicating mutable objects when copying, as discussed in details in Section 3.6.2.

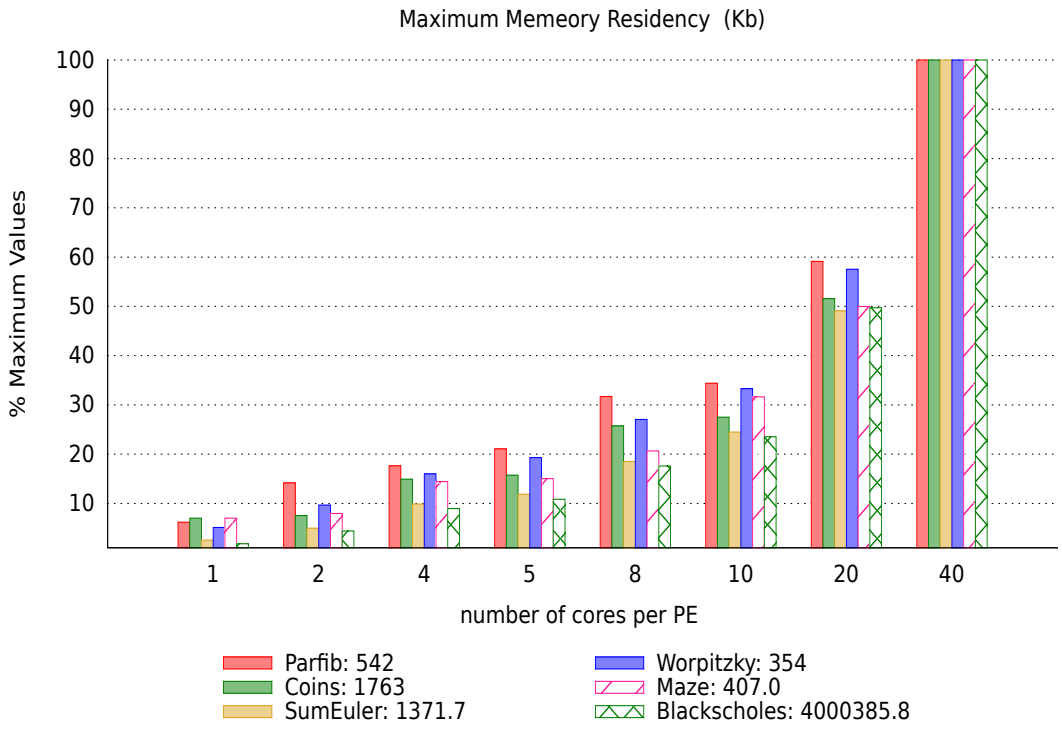

Figure 5.10: Normalised GUMSMP maximum memory residency on 40 NUMA cores

Crucially, in these experiments we always use the default minimum heap size of 0 for each PE; thus, there is no gain in the size of the initial heap when increasing the number of PEs. When increasing the minimum heap size, we observe a drop in runtime for GHC-SMP; this is as expected, because the garbage collections are less frequent. However, the GHC-SMP runtimes are still substantially higher than the GUMSMP runtimes.

An important metric for the performance of the memory management system is the allocation rate of the program. Figure 5.11 measures the amount of allocation per second, with increasing core numbers per PE. We explain the serious degradation in allocation rate, as an indirect consequence of the locking during GC, as discussed above. While the synchronisation overhead for stop-the-world parallel GC is largely independent from the live data set, the per-object locking overhead increases with both higher core numbers and larger live data set. As a combination of both overheads, the garbage collection phase becomes the con-

straining factor in the allocation performance. This behaviour is indicated by the consistent drop in the allocation rate beyond ca. 8–10 cores per PE.

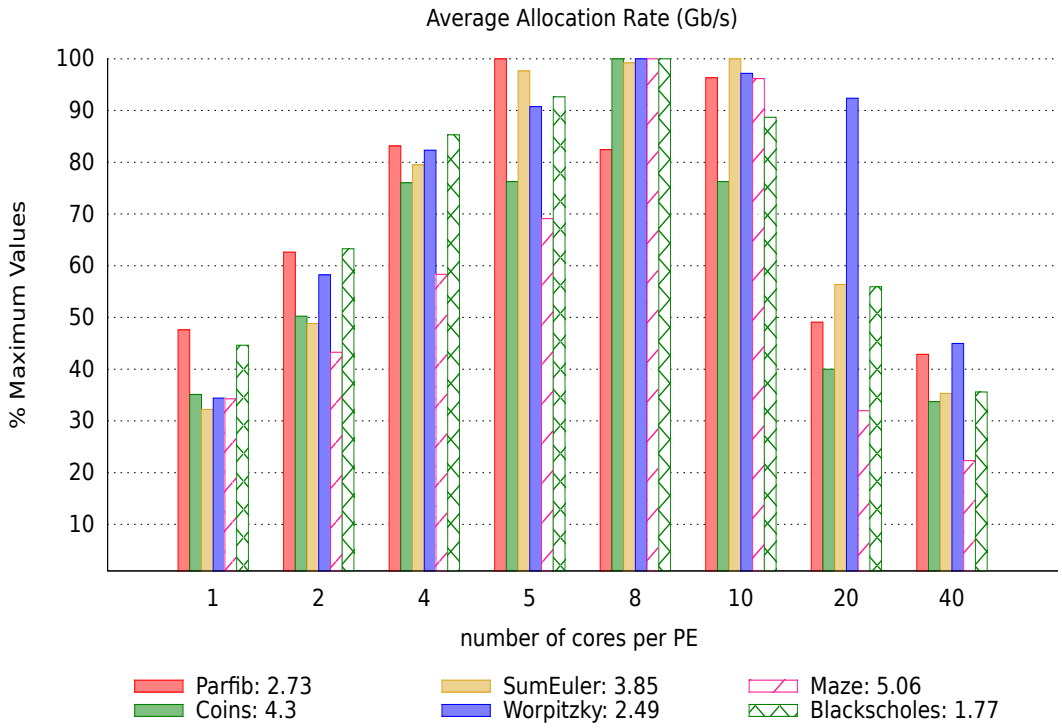

Figure 5.11: Normalised GUMSMP average allocation rate on 40 NUMA cores

The memory residency depicted in Figure 5.10 matches the profile of the GC percentage shown in Figure 5.7. This underlines the fact that the majority of the additional work done during GC for high core numbers is due to the size of the live data set in these configurations. We observe that the `blackscholes` program is one of the largest programs representing a stress test for the memory management system as it uses a large data structure (array of doubles), and therefore exhibiting the highest memory residency value.

In summary, the combination of global synchronisation for GC, and the locking overheads of allocation-triggered per-block locks, and promotion-triggered per-object locks, with the synchronisation to perform the stop-the-world parallel GC, account for a significant bottleneck in heavily allocating programs. This overhead, which becomes dominant with larger live data sets, is the main reason for the drop in performance observed in Table 5.3 and Figure 5.6. Notably, programs with a low allocation rate, such as `worpitzky`, exhibit the smallest

runtime improvement, over pure shared memory versions.

5.2.3 Summary and Discussion

Summary: We have investigated the impact of a NUMA memory model on the parallel performance of GpH, as a language with automated memory management, using 7 benchmarks on a state-of-the-art platform. We demonstrated that it is beneficial to use distributed heaps on NUMA, and specifically up to one heap per NUMA region. Hence, better performance is obtained for the benchmark programs with the hybrid shared/distributed memory models provided by our GUMSMP implementation. We observe best performance when using a single shared heap per NUMA region, which means in our measurements *using up to 5 cores per PE* in a configuration of 8 PEs, running on a hardware with 40 cores.

The main findings of this study are:

- GUMSMP's performance, with a maximum of 5 cores per PE is consistently better than the pure GHC-SMP execution, by a factor of up to 4.5. This configuration amounts to using a single shared heap for each NUMA region.

- For large core numbers, the GC overheads in the shared memory GHC-SMP increase drastically, primarily because of the larger live heap set.

- The allocation rate of GHC-SMP is typically much smaller than that of GUMSMP. We conjecture that this is a combination of the synchronisation overheads in the stop-the-world parallel GC, and the locking overheads incurred to prevent multiple GC threads from accidentally duplicating mutable objects, during parallel copying.

These improvements occurred, despite the fact that the RTS is not NUMA-aware, by simply structuring the heap into several distributed heaps and relying on the operating system for the concrete mapping. Further improvements should be possible with tighter integration of the RTS into the underlying operating system. Importantly, graph reduction based execution models, such as those

employed in modern systems, incur frequent and unstructured memory access. Therefore, the relative impact of different memory latencies is likely to be higher in our systems; thus, the core findings of this study can be seen as a stress test for modern runtime systems in the presence of NUMA architectures, contributing to studies of NUMA performance in languages with highly dynamic memory usage.

Discussion: The impact of non-uniform memory latencies on parallel performance has been studied recently in several contexts. A comparative empirical study by Bergstrom [20], involving running low-level benchmarks in the modified C language STREAM, summarises that 32-core Intel Xeon architectures provide larger cross-processor bandwidths, and suffer less from NUMA penalties compared to the widely used 48-core AMD Opteron architecture. This underlines the importance of NUMA in our measurements, using the latter architecture.

The baseline for our work is Marlow et al.'s [114] implementation of parallel, generational GC in GHC-SMP, which is the technology used in the main branch of the GHC runtime system (Section 3.6.2). This parallel generational GC was extended to concurrent GC in [117]: in this concurrent GC implementation a GC thread runs concurrently with mutator threads, avoiding the need for a stop-the-world GC. The implementation features local heaps, and parallel GC where each core has its own private heap collected independently. There is also a shared heap, which is collected less frequently, using the parallel stop-the-world GC; thereby, leading to less synchronisation. While this design is desirable and the new parallel concurrent GC achieves good performance improvements on up to 24 cores, scalability is lower than expected, and the implementation is significantly more complex than the current GC, which is of the parallel stop-the-world variety. Therefore, these modifications have not been merged into the mainline GHC.

Efficient automatic memory management on NUMA architectures is a challenge for aggressively allocating languages, such as declarative ones. One notable system that tackles these challenges is the Manticore system for parallel ML (Section 2.4.4.1), with the garbage collector implemented by Auhagen et al. [15]. It

combines a split heap design with a three phase, semi-generational GC that maximises locality and minimises global synchronisation. This was demonstrated to scale effectively with good utilisation, and steadily improved performance over all the available cores for both 48-core AMD Opteron, and 32-core Intel Xeon machines.

Modern Java implementations exhibit a similar trend. The measurements by Gidra et al. [73] of several DaCapo benchmark programs implemented in OpenJDK7, mirror our observations made for shared memory parallel Haskell programs: scalability is poor on a 48-core NUMA architecture, with a stop-the-world collector representing the main bottleneck. They provided more detailed measurements on the sources of overhead than we did, and identified the scanning and copying phases of remote objects, i.e. objects in remote NUMA regions, as the main overhead during GC, linking this to specific NUMA features. Moreover, they also demonstrated the effect of the pause time required for the stop-the-world parallel GC, which monotonically increases over time, resulting in unscalable GC, as the number of cores increases.

With a similar interest to that in our study, Alnowaiser [10] studied locality characteristics in two Java benchmarks. The study evaluated and analysed the locality characteristics of a rooted sub-graph for NUMA GC, using two DaCapo and SPECjbb2005 benchmarks. While data locality is generally high, on average more than 80% of objects are co-located with the root, and large, distributed graphs suffer from being exposed to load balancing techniques that diminish data locality. The author suggests modifications to the GC heuristic, using the root location as a locality heuristic for GC, and ensuring that GC is structured to process the roots on the same memory node in one phase.

While the work outlined above mainly presents observations on performance and scalability, several authors have developed concrete improvements at the application level, as well as inside the RTS. In particular, Terboven et al. [161] offers concrete recipes for the parallel programmer to enhance the performance

of OpenMP programs with task-level parallelism. These recipes are designed to improve data locality under several different workloads, and are based on extensive measurements of different task-level OpenMP implementations, using a range of benchmark programs.

While the above paper achieves performance improvements through changes at the program level, Yi Su et al. [156] developed NUMA-aware, thread placement algorithms inside the RTS for OpenMP considering the critical path when addressing NUMA latencies. They used on-line profiling of information obtained from hardware counters, to direct thread placement; thereby, improving performance by minimising the critical path of the OpenMP parallel regions. These algorithms have been evaluated using four NPB OpenMP applications, achieving between an 8% and 26% improvement over the default Linux thread placement algorithm.

5.3 Cluster of Multi-cores Results

In this section, we provide an evaluation of the performance of GUMSMP on a cluster of multi-cores, comparing the performance with the performance of GHC-GUM. The measurements in this section are made on the homogeneous Beowulf cluster of multi-cores, as specified in Section 4.2.1.

5.3.1 Evaluation of GUMSMP and GHC-GUM

GHC-GUM can be configured to use a hierarchical network as a flat network; in essence running one instance of the RTS for each available core. While this setup cannot make use of the physical shared memory, it does provide a useful reference point for the GUMSMP performance results. Crucially, the same GpH programs are used with no changes, as both systems represent RTS implementation for the GpH dialect of parallel Haskell.

Our study of the performance on NUMA in the previous section provide us with an insight into further possible improvement into the performance of

GUMSMP. In particular, the main factor affecting the performance of GUMSMP is the memory management inherited from the GHC-SMP. Therefore, in the evaluation conducted in this section we consider a new setting of GUMSMP, where the heap configuration is further tuned to tackle the memory management overhead. We demonstrate how further tuning of the allocation area settings for GUMSMP improves the overall performance, as a result of the lower memory management overhead, measured in terms of the percentage GC time (Section 5.3.4).

In our evaluation, the *low-watermark* mechanism is used for all the programs tested, configured to be equal to the number of local HECs. In the same way, the *low-watermark* is set to be 1 for GHC-GUM, to match the single PE instance.

Another important, tunable parameter for the RTS is the delay between receiving the unsuccessful FISH message, and sending another FISH message. A setting for the FISH delay is required to strike a balance between obtaining work as quickly as possible, and avoiding swamping the machine with FISH messages, as this endangers the scalability of the system. Similarly, the *low-watermark* mechanism is advantageous in GUMSMP in order to send FISHes more quickly, to obtain sufficient parallelism to feed all the local cores on a multi-core. In GUMSMP, the role of the FISH delay value (inherited from GHC-GUM) is more aggravated, due to the role of the gateway HEC in mediating any communication to other PEs. Thus, if the gateway HEC is in a delay period, it will not immediately send a FISH, despite the request is originating from a different HEC. This is reflected by longer idle times with moderate FISH delay values, compared to GHC-GUM executions. In all of the results presented, we approximate the setting for the FISH delay, that was established with GHC-GUM on flat networks, by dividing the GHC-GUM setting by the number of local HECs; e.g. with 3 and 4 HECs, we use the FISH delay values of 66 and 50 milliseconds respectively, as opposed to 200 milliseconds established for GHC-GUM.

In general, there is a useful set of tunable parameters to tune different aspects of the runtime system summarised in Table 5.4

Table 5.4: A useful set of tunable RTS parameters

Component	Parameter	Set	Effect
Load Balancing	low-watermark	-ql<n>	set the minimum number of sparks to keep locally to n sparks
	sparks to export initially	-qsx<n>	set the number of sparks to export remotely before local stealing to n sparks
Communication	fish delay	-qF<n>	set the delay time between unsuccessful fish and sending a new fish message to n milliseconds
	thunks per packet	-qT<n>	set the maximum number of thunks to pack in one packet to n thunks
	globalisation policy	-qG<n>	generate global address for these closures: 0=nothing, 1=thunks, 9=all
	packet size	-qQ<n>	set the size of the packet to communicate to n bytes
Memory Management	allocation area	-A<n>K	set the minimum allocation area size to n KB.
	Heap size	-H<n>M	set the minimum heap size to n MB.

5.3.1.1 Generated Parallelism

In this section, we study the behaviours of both systems in terms of generating and exploiting parallelism. The same level of potential parallelism, in the form of sparks, is generated by GHC-GUM, and GUMSMP for all programs. However, GUMSMP exploits more parallelism, as each HEC can quickly create threads, whereas in GHC-GUM, each PE must communicate to get work. This behaviour is more pronounced for divide-and-conquer programs or programs with nested parallelism, such as `maze`. For these programs, more parallelism is generated and exploited locally. As demonstrated in Table 5.5, GUMSMP exploits up to 2.9 times as many sparks as GHC-GUM. GHC-GUM on the other hand benefits more from thread subsumption, discussed in Section 3.4.

Table 5.5: GHC-GUM and GUMSMP amount of parallelism on 96 Beowulf cores

	Generated Sparks	**Converted Sparks on 96 cores**		
Programs		GHC-GUM	GUMSMP	$\frac{GUMSMP}{GUM}$
`parfib`	1346K	9956	28554	2.9
`coins`	17.7K	2949	3490	1.2
`sumEuler`	0.55K	553	553	1.0
`worpitzky`	7K	3751	5864	1.6
`mandelbrot`	4K	4093	4095	1.0
`maze`	33.3K	16534	20483	1.2
Min.				**1.0**
Max.				**2.9**
Geom Mean.				**1.5**

5.3.1.2 Communication and Threads

To study the behaviour of GUMSMP, in terms of the amount of threads generated, as well as the amount of communication, we use two divide-and-conquer and two data-parallel programs. In particular, we use data generated from the evaluation on 40 Beowulf cores to predict the trends for these aspects as the number of cores per PE increases. Table 5.6 demonstrates the number of data messages (excluding the number of work request FISH messages), showing that as expected, the communication rate falls as the number of cores per PE increases. This demonstrates the benefits of the GUMSMP design in providing one gateway HEC representing the communication engine for the set of local HECs within the same multi-core, as opposed to GHC-GUM, where each individual PE performs communication explicitly. However, this trend is different in the case of `parfib`, as a program that generates massive amounts of regular parallelism, 1346K sparks that are more aggressively exploited by local HECs. We correlate this trend with the large increase of the number of local threads generated for `parfib` which is up by a factor of 144 (i.e. 49919/346) for 8 cores per PE setting, as demonstrated in Table 5.7. In such case, we observe that the amount of communication does not have a direct effect on the overall performance, since there

is large computation-to-communication ratio compared with other programs that generate a lower amount of parallelism, and only small data items need to be communicated.

Table 5.6: GHC-GUM and GUMSMP messages volume on 40 Beowulf cores

Configuration	**GUM**	**GUMSMP**			
PE/Cores	PE 40	20/2	10/4	8/5	5/8
`parfib`	10.9K	5.4K	18.7K	18.8K	26.8K
`sumEuler`	4.4K	4.2K	4.0K	3.9K	3.6K
`worpitzky`	25.9K	25.4K	19.3K	16.0K	12.1K
`mandelbrot`	22.6K	21.2K	17.8K	15.1K	12.2K

In terms of the number of threads created, GUMSMP creates more threads in general than GHC-GUM, resulting from its being aggressive to exploit parallelism locally. This number of threads increases with the increase in the number of cores per PE. We conclude that for GUMSMP, the communication volume decreases, with the increase in the number of threads locally created, as the number of cores per PE increases.

Table 5.7: GHC-GUM and GUMSMP number of threads created on 40 Beowulf cores

Configuration	**GUM**	**GUMSMP**			
PE/Cores	PE 40	20/2	10/4	8/5	5/8
`parfib`	346	4085	16288	18722	49919
`sumEuler`	87	61	67	77	78
`worpitzky`	2099	3571	3238	3194	3237
`mandelbrot`	1814	3393	3260	3556	6949

5.3.2 The Performance of GUMSMP and GHC-GUM

The previous section reveals that GUMSMP offers an advantage over GHC-GUM, by exploiting larger amounts of parallelism and restricting the communication component to a single HEC for each PE. However, as discussed in the NUMA evaluation, another important factor affecting performance is memory

management (Section 5.2.2). In this section, we investigate different tuning for GUMSMP, motivated by the bottlenecks identified on NUMA. The first optimisation we investigate is how to tune the number of cores per PE (Section 5.3.3). Then we investigate how to reduce the memory management bottlenecks, and the resulting improvement in the performance of GUMSMP (Section 5.3.4).

5.3.3 Optimising the Number of Cores Per PE

As outlined in Section 4.3.5, we systematically investigate in this section how to optimise the number of cores per PE in a *distributed memory cluster*. In this experiment, we fix the total number of Beowulf cores to be 84 and test all possible combinations of cores per PE and their effect on performance for two data-parallel and two divide-and-conquer programs. This serves as guidance for our work to optimise the performance of GUMSMP on clusters of multicores.

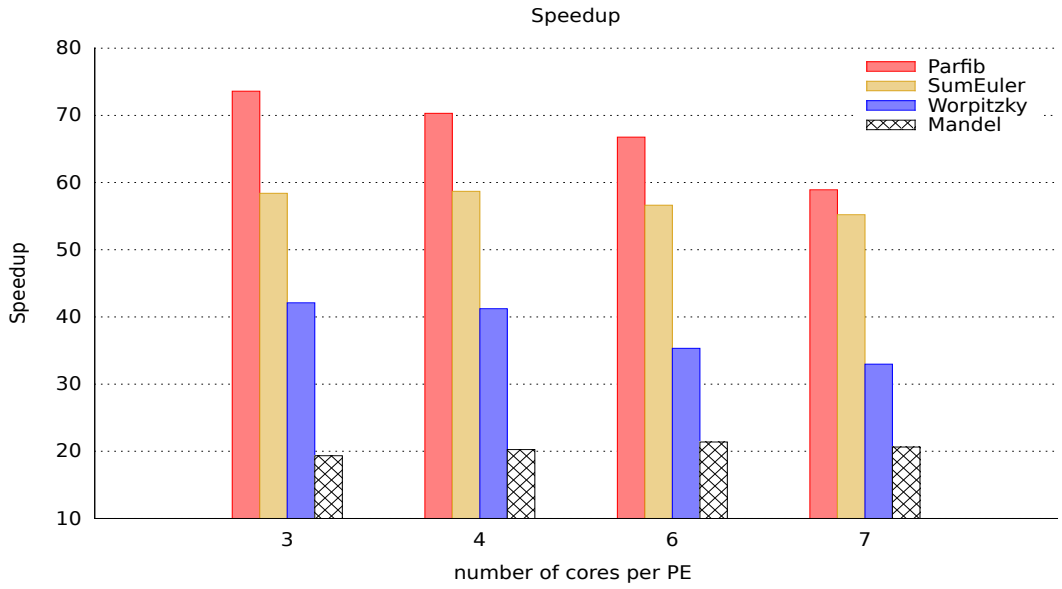

Figure 5.12: GUMSMP speedup on 84 Beowulf cores

Figure 5.12 shows that using GUMSMP with 3 cores per PE instance consistently results in better performance with divide-and-conquer programs, with a speedup of up to 74 on 84 cores for `parfib`. Data-parallel programs still perform better using GUMSMP with a larger numbers of cores per PE, and achieve the best performance using 4 cores per PE for `sumEuler` and 6 cores per PE for `mandelbrot`, with a speedup of 59 and 21 respectively on 84 cores. However,

with the increasing numbers of cores per PE instances, the performance degrades, as a consequence of the shared memory management discussed in the previous section. We therefore fix the number of core per PE to 3 for divide-and-conquer programs, and 4 for data-parallel-programs.

5.3.4 Optimising the Setting of the Allocation Area

As outlined in Section 4.3.6, we investigate in this section the effect of optimising the heap setting for GUMSMP by increasing the allocation area available for each PE on the performance of GUMSMP. In particular, we show the performance of GUMSMP with the standard allocation area setting, and with a larger allocation area. The larger allocation area is set by multiplying the default allocation area size, with the number of cores: $512K * N$ to reduce contention on this shared area. Then we compare the performance with GHC-GUM for programs exhibiting different parallel paradigms: data-parallel, and divide-and-conquer.

5.3.4.1 Data Parallel Programs

For data-parallel programs, we fix the number of cores per PE to 4 and measure the performance on the cluster of up to 32 PEs, resulting in up to 128 cores. Figure 5.13 shows the speedups for all the data-parallel programs under GUMSMP and GHC-GUM. Notably, GUMSMP delivers better speedups for `sumEuler` and `mandelbrot` with the standard heap settings showing improvements of 23% and 11% respectively.

In general, GUMSMP performs better with data-parallel programs, as for such programs with a single source of parallelism, it is beneficial to send off a large computation early, with other HECs collect parallelism locally. This structure of parallelism is a natural match for the hierarchically structured RTS. Consequently, we observe a significant reduction in communication, compared to the GHC-GUM instance; the number of messages dropped by up to 21% for `mandelbrot` as demonstrated in Table 5.6, for the setting of 4 cores per PE. The effect of communication is more pronounced for `mandelbrot`, as this program uses large data structures. This program also communicates the largest number of graphs. In particular, for `mandelbrot`, on 40 cores, GUMSMP communi-

cates up to a factor of 1.35 fewer graph structures than GHC-GUM (i.e. 71.7MB for GHC-GUM, and 53MB for GUMSMP).

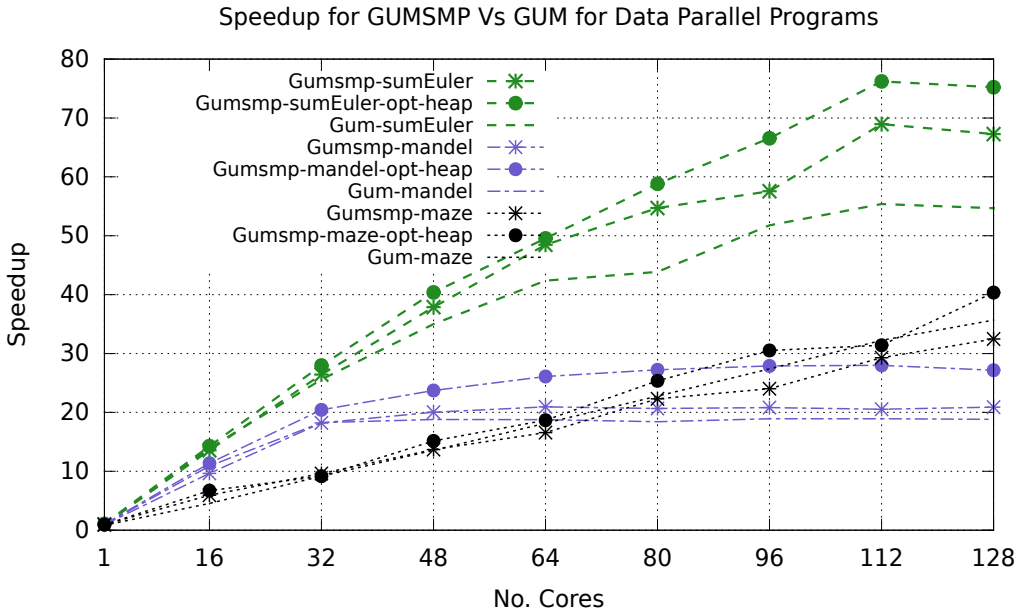

Figure 5.13: Speedup of GHC-GUM vs. GUMSMP on up to 128 cores for *data-parallel programs*

For all three data-parallel programs, the optimised heap setting (denoted **optimised** in Table 5.8) improves the performance by up to a factor of 1.4. This contributes to the better scalability of `mandelbrot` as the number of cores per PE increases up to 6 cores per PE, as demonstrated in Figure 5.12, at a point where memory management overhead prevents further improvement.

Maze exhibits speculative parallelism, and the program terminates based on when the solution is found, so there is more variation in the runtimes compared with other data-parallel programs (i.e. on 128 cores the runtimes of three executions are: 76.7s,88.4s,88.6s). It also represents one that suffers more from the memory management overhead, with the largest percentage GC time. For `maze`, the optimised heap setting is important to get better performance compared with GHC-GUM.

Table 5.8: GC overheads for GUMSMP and GHC-GUM for data-parallel programs on 128 cores

	% GC time			**Absolute Speedup**			$\frac{optimised}{standard}$	$\frac{optimised}{GUM}$
Programs	GUM	G-SMP Standard Heap	G-SMP Optimised Heap	GUM	G-SMP Standard Heap	G-SMP Optimised Heap		
`maze`	7.2	23.0	10.3	36.1	32.5	40.3	1.2	1.1
`mandelbrot`	3.5	13.1	6.6	18.8	20.9	28.9	1.4	1.5
`sumEuler`	1.4	5.4	3.9	54.7	67.3	75.2	1.1	1.4
Min.							**1.1**	**1.1**
Max.							**1.4**	**1.5**
Geom Mean.							**1.2**	**1.3**

5.3.4.2 Divide and Conquer Programs

For the divide-and-conquer programs, we fix the number of cores per PE to 3, and measure the performance on the cluster of up to 32 PEs, resulting in up to 96 cores. Figure 5.14 shows the speedups for all the divide-and-conquer programs under GUMSMP and GHC-GUM. Crucially, `parfib` is a very simple program that generates massive amounts of regular parallelism. It also generates the largest number of threads as discussed in Section 5.3.1.2. As a result, it delivers the largest speedup of up to 85 on 96 cores.

It is clear that the optimised heap setting (denoted **optimised** in Table 5.9) improves the performance for GUMSMP; however, GHC-GUM performs better with a difference of 1.3%. Moreover, for `coins`, the optimised heap setting is also required to get better performance, as it represents the program that uses larger data structures, resulting in large percentage GC time compared with the other divide-and-conquer programs. `worpitzky` represents the divide-and-conquer program that generates the lowest amount of parallelism, and percentage GC time; therefore, GUMSMP provides better performance than GHC-GUM. This performance is further improved with the optimised heap setting.

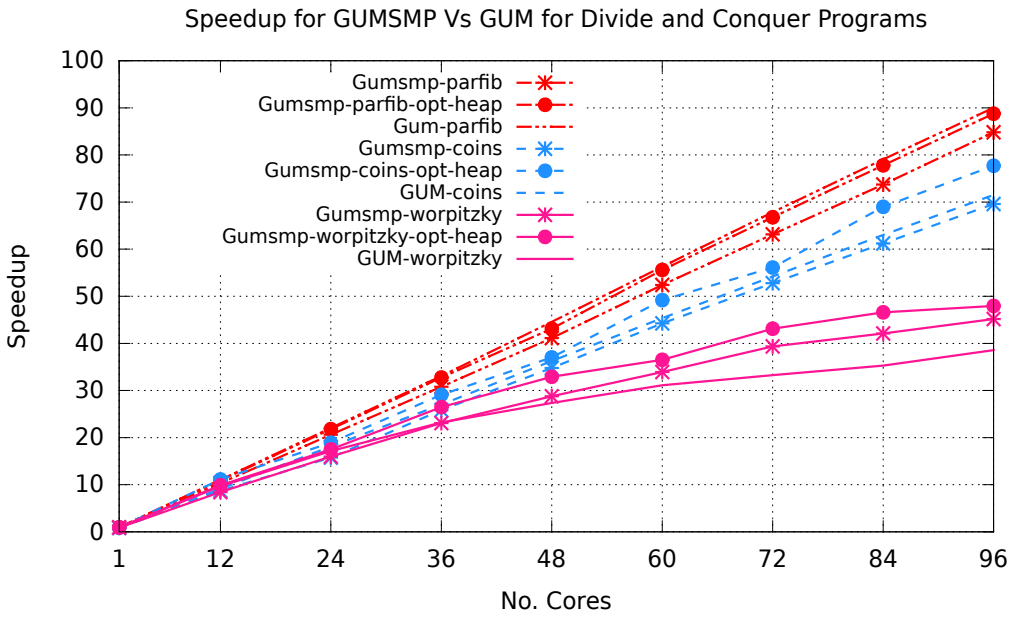

Figure 5.14: Speedup of GHC-GUM vs. GUMSMP on up to 96 cores for *divide-and-conquer programs*

Table 5.9: GC overheads for GUMSMP and GHC-GUM for divide-and-conquer programs on 96 cores

	% GC time			Absolute Speedup			$\frac{optimised}{standard}$	$\frac{optimised}{GUM}$
Programs	GUM	G-SMP Standard Heap	G-SMP Optimised Heap	GUM	G-SMP Standard Heap	G-SMP Optimised Heap		
`parfib`	3.5	9.1	5.5	90.0	84.6	88.8	1.0	0.9
`worpitzky`	2.6	4.7	1.4	38.6	45.2	47.9	1.0	1.2
`coins`	8.8	16.9	7.9	71.5	69.6	78.3	1.1	1.0
Min.							**1.0**	**0.9**
Max.							**1.1**	**1.2**
Geom Mean.							**1.0**	**1.0**

5.3.5 Summary

This section evaluated GUMSMP on a cluster of multi-cores, comparing its performance with GHC-GUM on up to 128 cores. The key findings are:

- GUMSMP aggressively exploits more parallelism compared to GHC-GUM by a factor of 1.5 on average (Table 5.5).

- GUMSMP tends to communicate fewer messages as the number of cores per PE increases, with an exception of `parfib` which produce and exploit massive amount of regular parallelism (Table 5.6).

- For `maze`, and `coins`, the two programs that exhibit the largest percentage GC, selecting the right heap configuration was important to exceed the performance of GHC-GUM.

- With the optimised heap setting, the performance of GUMSMP is up to a factor of 1.5 higher than GHC-GUM.

- With the optimised heap setting for GUMSMP, the percentage GC time is reduced, and the performance is improved for all programs tested by a factor of up to 1.4 compared with the standard heap setting (Tables 5.8 and 5.9).

5.3.6 More Active Load Management

This section investigates different load distribution policies for GUMSMP. In particular, the default setting of GUMSMP involves passive load distribution in terms of sparks, but active load distribution to share the threads. Section 4.3 has shown that the aggressive load balancing policy improves the performance of GUMSMP. Therefore, it is natural to consider active load distribution as another means to increase the performance (Section 4.3.7). In this section, we investigate an active load distribution policy for sharing sparks. In such setting, a combination of active and passive spark distribution is implemented, whereby each HEC can steal sparks from any other HECs local pool. Moreover, each HEC checks to see if it has extra sparks, and if any HEC is idle. Then, it will actively push sparks, onto the spark pools of the idle HECs. This process represents performing active load distribution of sparks at the intra-node level. We have tested and compared the new resource policy, for the two programs that generate the largest amount of parallelism, `maze`, and `parfib`, as presented in the speedup Figures 5.15 and 5.16. Moreover, we add the optimised heap setting from Section 5.3.4 to our comparison. Figure 5.15 shows that the combination of active and passive work distribution improved the performance of `maze`, by up

to 22% on 128 cores. However, `parfib` slows down by up to 11% on 96 cores, as demonstrated in Figure 5.16. In all settings, the optimised heap setting shows the best performance.

The spark pushing we discuss in this section represents the addition of active load distribution to the default of the passive load distribution mechanism at the intra-node level. This is equivalent to the use of the *high-watermark* mechanism discussed to add the active spark distribution in the inter-node level (Section 3.5.4.1). While a *high-watermark* mechanism is fully implemented in GUMSMP, we observe that the runtime is sensitive to the concrete setting of the watermark. If it is too high, excessive communication is incurred, if it is too low it is ineffective. Further work is necessary to determine effective settings, possibly based on previous monitoring of the parallel execution. This mechanism requires more evaluation, and we plan to analyse the impact of the combining the active and passive spark distribution in future work. In particular, for GUMSMP we have discussed around seven different policies to be tuned in order to improve the performance. The settings for all those mechanisms are statically tuned. We see in the future research directions (Section 6.2.1), that we plan to investigate the application of machine learning approaches, to adjust the settings for the different mechanisms in order to get the best possible performance.

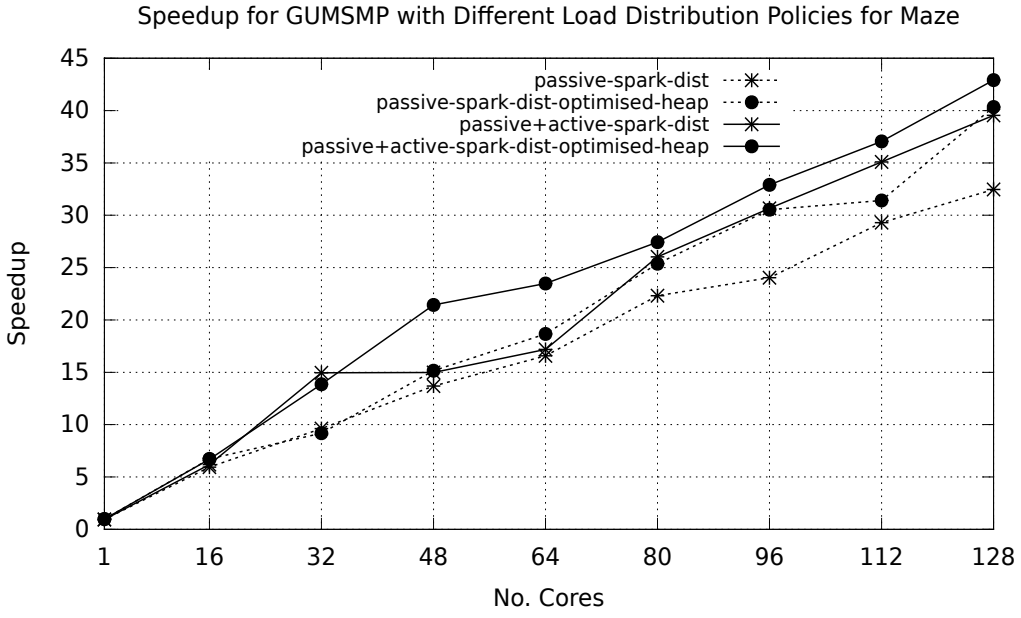

Figure 5.15: Speedup of GUMSMP (different load distribution policies) for `maze`

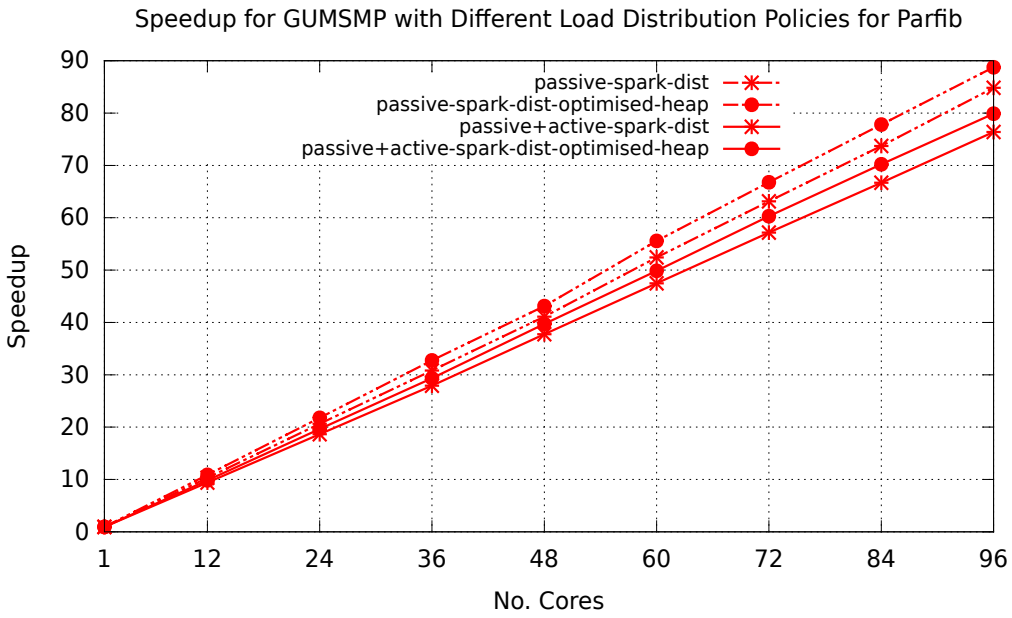

Figure 5.16: Speedup of GUMSMP (different load distribution policies) for `parfib`

5.4 Scalability Results

GUMSMP is designed for hierarchical architectures with homogeneous nodes, i.e. each PE is the same. Section 5.3.3 reports scaling on a homogeneous cluster architectures of up to 32 nodes with 128 cores. While there is no specific support for heterogeneity, in the form of different machines like multi-cores with GPUs, the design is flexible and distributes work such that the more powerful machine will complete its work faster, and therefore receive more work than other less powerful ones. This automatic adaptation is a consequence of the work-stealing load distribution with the *low-watermark*.

This section presents the scalability results when combining our 32 nodes of the Beowulf cluster with up to 36 Linux machines with different characteristics, all within Heriot-Watt university and connected with a relatively slow Ethernet connection (100 Mb/second), which represents an instance of hierarchical clusters. Table 5.10 shows the approximate network latency between the nodes in the same cluster, as well as between nodes in different clusters (measured using the bwtest program from PVM3.4.5 benchmarks). It is clear from the table that the latency is very small, and nearly the same, as both clusters share the same network and are located in the same building.

	Beowulf node	**Linux Machine**
Beowulf node	236	331
Linux Machine	348	370

Table 5.10: Approximate latency between nodes in Beowulf cluster and the Linux machines in μs

The correspondence of PE to the underlying physical host is intended to be compatible with the number of cores available on each machine. In particular, we use a setting of 3 HECs per PE, which are mapped as 2 PEs on all Beowulf machines, where each node has 8 cores. For each Linux machine, one PE is assigned to each machine, which has only 4 physical cores. Figure 5.17 demonstrates the mapping of the PEs to the underlying physical machines, and Table 5.11 presents the computation power of the machines.

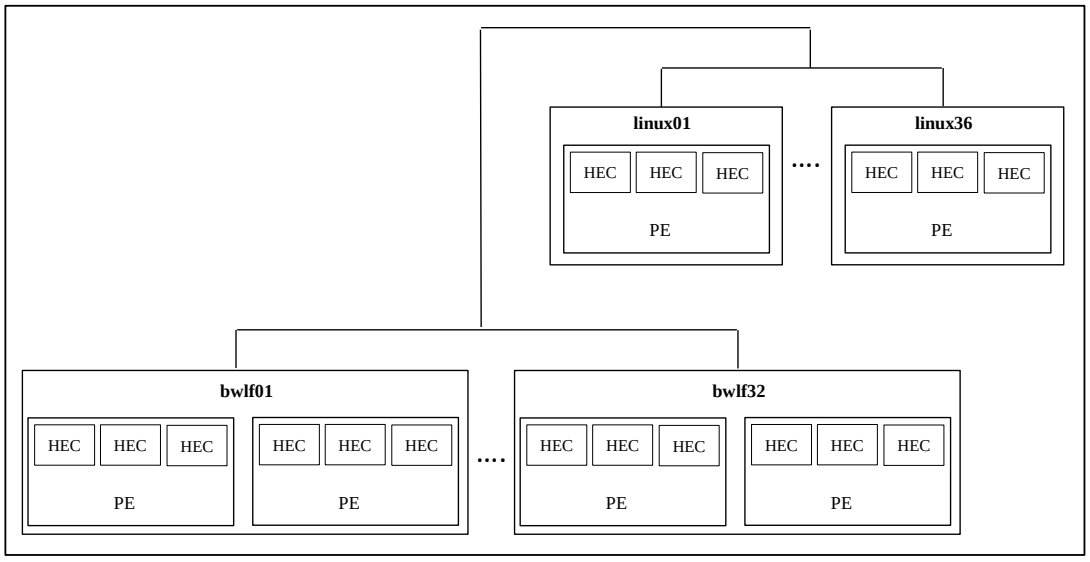

Figure 5.17: Levels of architectures used for scalable results

Machine	**CPU Speed (GHz)**	**Cache (Kb)**	**Memory (Gb)**	**Nodes used**	**Cores used**
Beowulf Cluster	2.00	4096	12	64	192
Linux Machines	2.80	8192	16	36	108

Table 5.11: Characteristics of the levels of architectures

The key results in this section demonstrate that GUMSMP scales well to hundreds of cores. Moreover, GUMSMP can adapt to a deeper hierarchy of machines, not only to a single flat cluster. In particular, we present in Figure 5.18 the scalability of 3 benchmarks with different parallelism paradigms, on up to 300 cores.

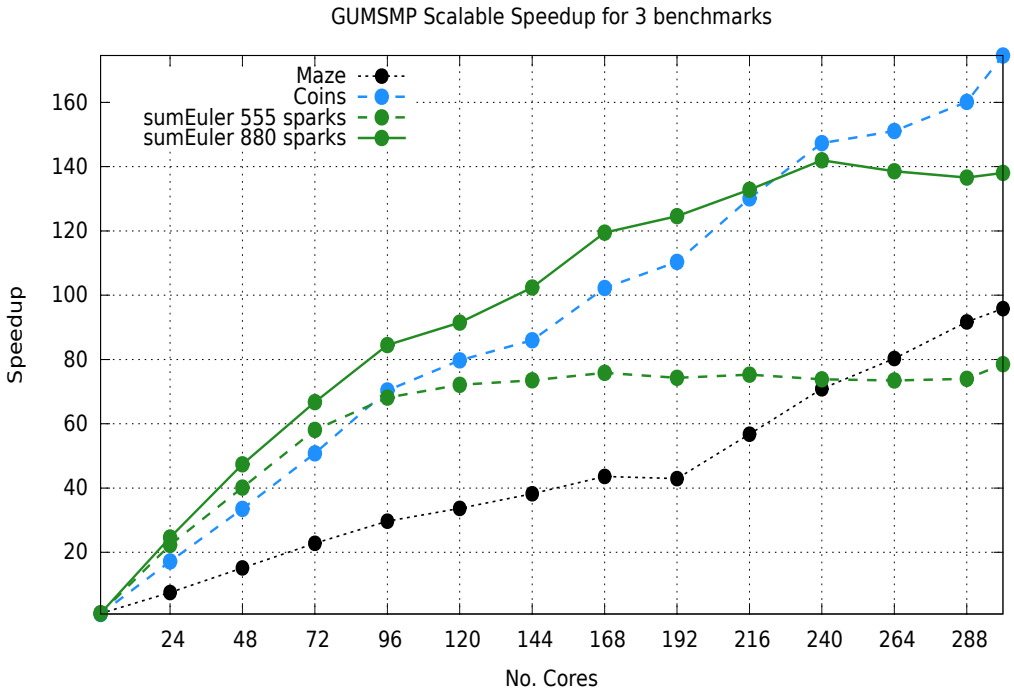

Figure 5.18: Speedup of GUMSMP on up to 300 cores for *3 benchmarks*

It is clear that due to the symmetric latency between nodes on different clusters, the latency effect on the overall performance is negligible, as all programs continue to scale well beyond the single cluster, as shown in Figure 5.18 from 192 cores onwards. The divide-and-conquer program `coins` exhibits irregular parallelism, with a moderate amount of parallelism; overall around 17K of sparks are generated. Nevertheless, it still shows very good scalability on up to 300 cores with a speedup of 175. In terms of the nested-data-parallel program, `maze`; with the use of a combination of active and passive load distribution, as discussed in the previous section, continues to scale and achieve a speedup of 100 on up to 300 cores, representing an excellent scalability for this large program. `sumEuler` on the other hand shows lower scalability as a result of having limited amount of parallelism, with 555 sparks generated in total. As a result it shows tail-off after around 120 cores where the communication-to-computation ratio increase

as a result of having low amount of parallelism to keep all the 300 cores busy with computation. When increasing the amount of parallelism for `sumEuler`, so that the total amount of spark generated is 880, it scales further beyond 200 cores with a speedup of 140. This scalability results represent the first systematic scalability results of GpH up to 300 cores.

Chapter 6

Conclusion

This thesis investigated whether a multi-level parallel implementation can effectively exploit the increasingly important hierarchical architectures. The aim is to achieve scalability while still using a single high-level programming model. We have designed and developed a novel runtime system implementation, GUMSMP, which manages parallelism on different levels of hierarchical architectures, employing different load distribution policies on the different levels of the hierarchy. The design combines the advantages of the shared and the distributed runtime system implementations. The novel combination of policies enhances the flexibility of the system, and the evaluation of our new runtime system demonstrates runtime and scalability improvements over the existing runtime systems for parallel Haskell on several hierarchical architectures.

6.1 Contributions and Achievements

The thesis presents the design, implementation, and evaluation of GUMSMP, a novel multi-level parallel Haskell RTS implementation. GUMSMP targets hierarchical parallel architectures like clusters of multi-cores, or large NUMAs, where the system can use a shared heap on each node, and distributed heaps across nodes. It achieves scalability as it efficiently exploits the specifics of such hierarchical platforms by combining distributed memory parallelism, using a virtual shared heap at the distributed memory level (e.g. clusters), with low-overhead shared memory parallelism on the small scale, physical shared memory level (e.g.

multi-cores). Moreover, unlike many other multi-level parallel paradigms like MPI+OpenMP, it provides a single programming paradigm, GpH. The thesis provided the following contributions.

6.1.1 Contribution 1: GUMSMP Design and Implementation

The design and implementation of a new and sophisticated shared and distributed memory parallel runtime system for a production functional language is presented. Accounting for the hierarchical nature of the modern architectures, like clusters of multi-cores, it provides a design for an improved load distribution mechanism [6] (Sections 3.5.3, 3.5.4).

The thesis presented the design and implementation details for the hierarchy-aware parallel Haskell RTS implementation GUMSMP. The GUMSMP design is based on the successful GHC-GUM and GHC-SMP technologies that already exist at inter-node (across the cluster) and intra-node (within a multi-core) levels. In particular, it combines a mechanism of work stealing for passive load distribution, with an adaptive, dynamic mechanism for automatically distributing work and data on a cluster (between the nodes of multi-cores), where communication is based on explicit message passing. Within the node (a multi-core), a combination of passive and active load distributions are employed, whereby communication between cores is carried out as direct access to the shared memory within the node. It also discussed a list of different design alternatives and motivated the decisions made (Section 3.5.4).

6.1.2 Contribution 2: GUMSMP Performance Tuning

The development and evaluation of the effectiveness of seven different policies to improve automatic hierarchical load distribution [6] (Section 4.3).

The thesis developed and assessed the effect of the following load performance tuning policies: the *low-watermark* mechanism for work pre-fetching, showing improvements of up to a factor of 3 (Section 4.3.1), and favouring inter-node work distribution, showing an improvement of up to 19% (Section 4.3.2). Addition-

ally, a novel spark segregation mechanism is studied to separate local and global sparks, identifying different policies to export sparks remotely and select sparks for local evaluation (Section 4.3.3). Moreover, the effect of using dedicated gateways, reserving one core for communication work is studied in (Section 4.3.4). In terms of reducing the bottlenecks of memory management overhead, we discuss further tuning by optimising the number of cores per PE (Sections 4.3.5, 5.3.3), as well as adjusting the heap settings by providing larger allocation area which consistently improve performance by a factor of up to 1.4 (Sections 4.3.6, 5.3.4). Furthermore, we show that combining active and passive load distribution for sparks at the intra-node level delivers improvement of up to 22% (Sections 4.3.7, 5.3.6).

6.1.3 Contribution 3: A Systematic Performance Evaluation of GUMSMP

Undertake a systematic performance evaluation of GUMSMP *in comparison to* GHC-SMP *and* GHC-GUM *using a set of benchmarks on both cluster and NUMA architectures.* The thesis showed that compared with GHC-SMP, our GUMSMP delivers performance improvement of a factor of 3.3 on average on a NUMA machine with 40 cores by balancing the shared and distributed heaps. Investigation of the scalability limits of GHC-SMP revealed that garbage collection is a primary source of overhead [8] (Section 5.2). Compared with GHC-GUM, GUMSMP provides an improvement of up to a factor of 1.5 on average on a cluster of multi-core architecture with up to 128 cores by exploiting the specifics of shared memory (Section 5.3). The thesis also showed that GUMSMP scales to deliver a speedup of up to 175 on a cluster with 100 nodes, comprised of 300 cores (Section 5.4).

6.2 Limitations and Future Research Directions

This thesis has the following limitations:

- **System Stability:** The work presented in this thesis is based on our implementation of GUMSMP, which is not fully stable. It occasionally fails during the execution especially with programs with large data structures.

More debugging work would be essential to debug the complex distributed RTS. Moreover, there is no fault tolerance mechanism implemented which resulted in a system failure if any PE failed during the computations. Further work would be needed to stablise the system, possibly adding a fault tolerance mechanism as an additional feature, and test large parallel Haskell applications.

- **Compiler Optimisation:** In the GUMSMP benchmarking we have not applied compiler optimisation (i.e. the GHC -02 compilation flag) which would be good practice, but because for a long time, both development and measurements did not use optimisation, we completed our measurements without it. We investigate the implications of the decision by comparing the performance of two programs; the small `worpitzky`, and the large `maze` with and without compiler optimisation on, and we report the results in Appendix A. The key observation for both programs is that while the runtimes for the programs are different with and without optimisation, the speedups are very similar, as these are ratios of runtimes, thus confirming the validity of our results.

6.2.1 Future Research Directions

6.2.1.1 NUMA-aware System

In future work, we plan to study ways to make the RTS NUMA-aware, initially by directly mapping an RTS heap to a particular NUMA region.

6.2.1.2 Auto Tuning

The development of GUMSMP discussed around seven different performance tuning policies with different combinations. We have made an extensive systematic study and evaluation of the effectiveness of those policies to improve the performance. However, the policies define a multi-dimensional space with a dimension for each policy. Some have a range of integer values, i.e. the setting of the *low-watermark*, and others have a smaller fixed space, like the binary selections when turning the policy on or off such as the use of passive load distribution

within a single multi-core. It would be a useful research direction to explore the application of the machine learning mechanism to adjust the settings of different policies for specific architectures automatically.

6.2.1.3 Dynamic Tuning

An interesting research direction would be to investigate dynamic tuning for the RTS, where it tunes itself during the execution. For example, the aim of the *low-watermark* is to distribute work eagerly, which might lead to starvation at the end of the execution where the amount of available work is low compared with the larger number of idle PEs. We could aim for further improvements to the system to adjust the settings of the *low-watermark* dynamically by extending the structure of FISH and SCHEDULE messages, in order to carry information about the size of spark pools for the PEs visited. This could be achieved by computing the average of the spark pool sizes for all the visited PEs, then including this information with the forwarded FISH, unsuccessful FISH, or with the SCHEDULE messages. The average spark pool size seen by a FISH(sp) can be computed as an integer based on the number of PEs visited so far (visitedPE), and the size of the spark pool on the current PE (sp_thispe):$sp = \frac{sp*visitedPE+sp_{thispe}}{visitedPE+1}$
The average spark pool size can be used to adjust the *low-watermark* setting for the PE that received the message. Different heuristics mechanisms can be applied to adjust the value of the *low-watermark*, depending on which message the information arrives with. If it comes with an unsuccessful FISH, then this might trigger disabling of the *low-watermark* completely. On the other hand, if it was part of the SCHEDULE, then it would be possible to adjust the setting of the *low-watermark*, according to the average sparks received with the message. Similar ideas were implemented by Al-Zain [4] for the Grid-level [3] on heterogeneous architectures, where there is large latency between different clusters of multi-cores, and the communication is very expensive. These ideas include recalling where the PE most recently found work, and then alternating between trying the PE, that had work with a random selection of the destination PE, or maintaining a table detailing the number of the visited PE, load, and the time-stamp. Then targeting the work request to the PE with the largest spark pool.

6.2.1.4 Spark Tagging

A further interesting research direction would be to apply and evaluate the use of pointer tagging for sparks in order to distinguish between local and global sparks, or the originating PE id number for the sparks. We expect such a policy to improve locality by grouping those sparks coming from the same source PE together. This might be combined with the our implementation of the import-spark-pool from Section 4.3.3.

6.2.1.5 Inter-cluster Performance Study

Our evaluation of the performance of GUMSMP covered hierarchical architectures with a two level hierarchy, shared-memory (multi-cores or NUMA regions) connected by distributed memory network. Current architecture trends are moving towards architectures with deeper communication/memory hierarchies, and it would be interesting to explore how our techniques work on such architectures. For example, by evaluating GUMSMP on inter-cluster architectures (i.e. by connecting the Heriot-Watt University cluster with the University of Glasgow cluster) and studying the scalability of the system, as well as the effect of the high latency of such a configuration on the performance. Such an integration of different clusters would be facilitated using a communication layer like MPICH-G2 (instead of PVM), which provides authentication to permit the communication between the two clusters.

Appendix A

Optimisation

This section compares the performance of 2 programs; the small `worpitzky`, and the large `maze`. We use two different settings: with and without compiler optimisation on. The key observation for both programs is that while the runtimes for the programs are different with and without optimisation, the speedups are very similar.

Worpitzky

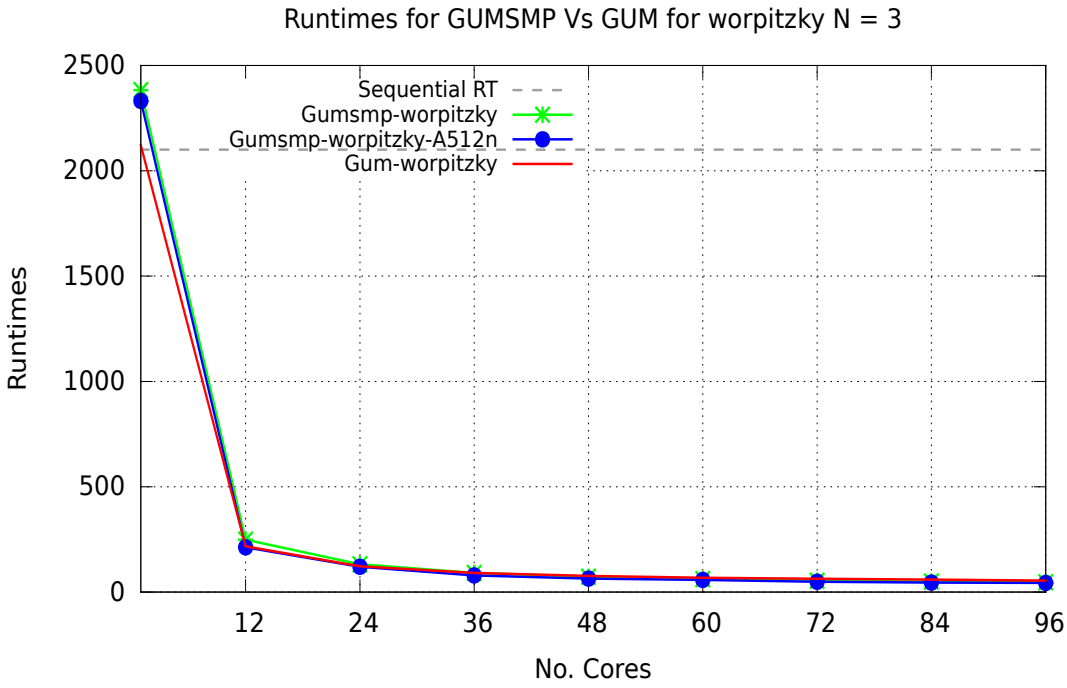

Figure A.1: Runtimes of GHC-GUM vs. GUMSMP with no optimisation

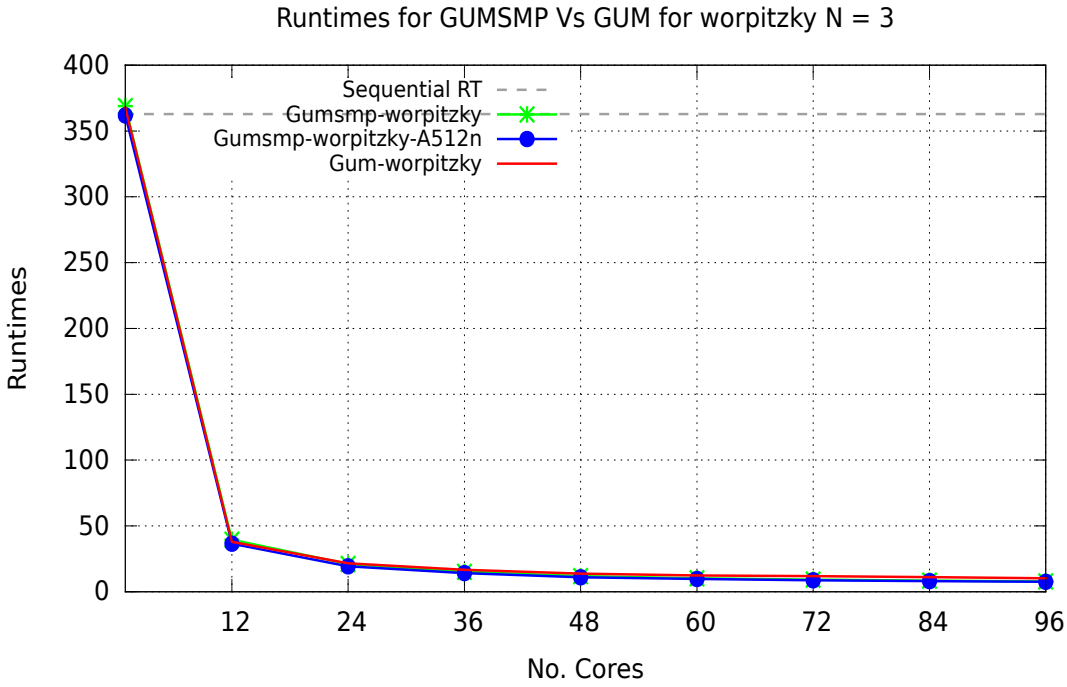

Figure A.2: Runtimes of GHC-GUM vs. GUMSMP with optimisation

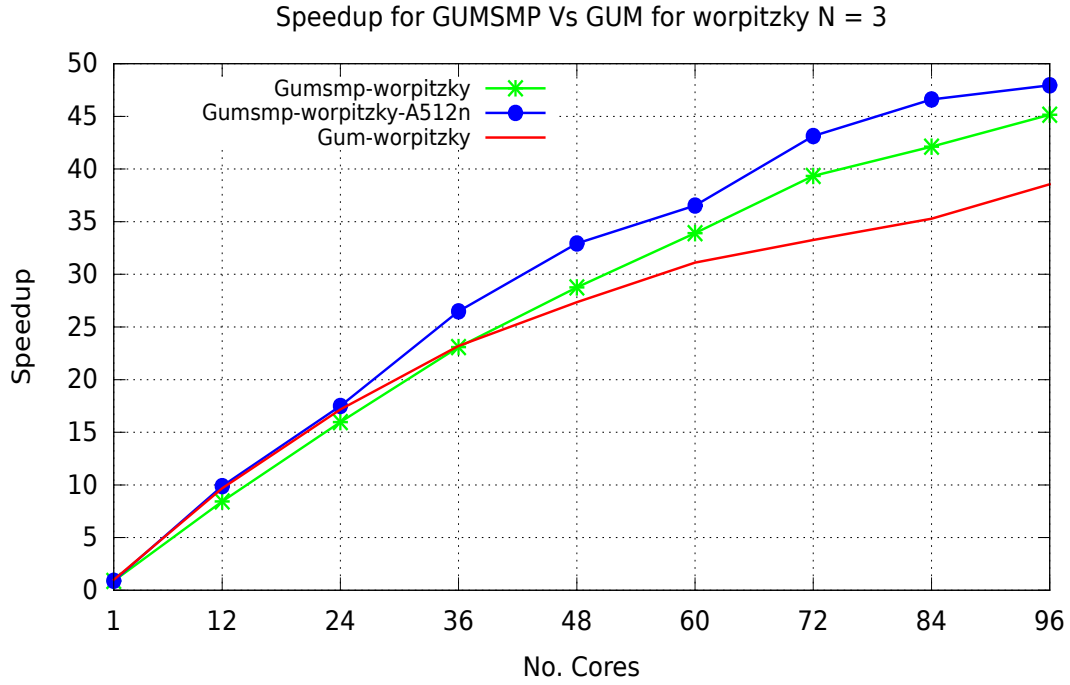

Figure A.3: Speedup of GHC-GUM vs. GUMSMP with no optimisation

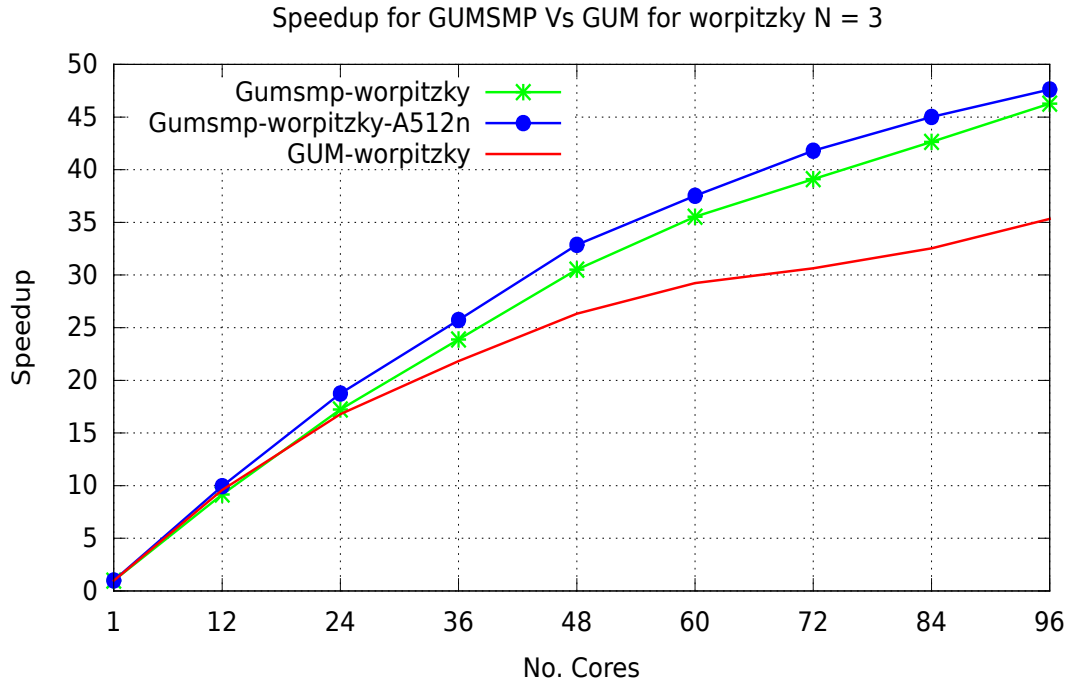

Figure A.4: Speedup of GHC-GUM vs. GUMSMP with optimisation

Maze

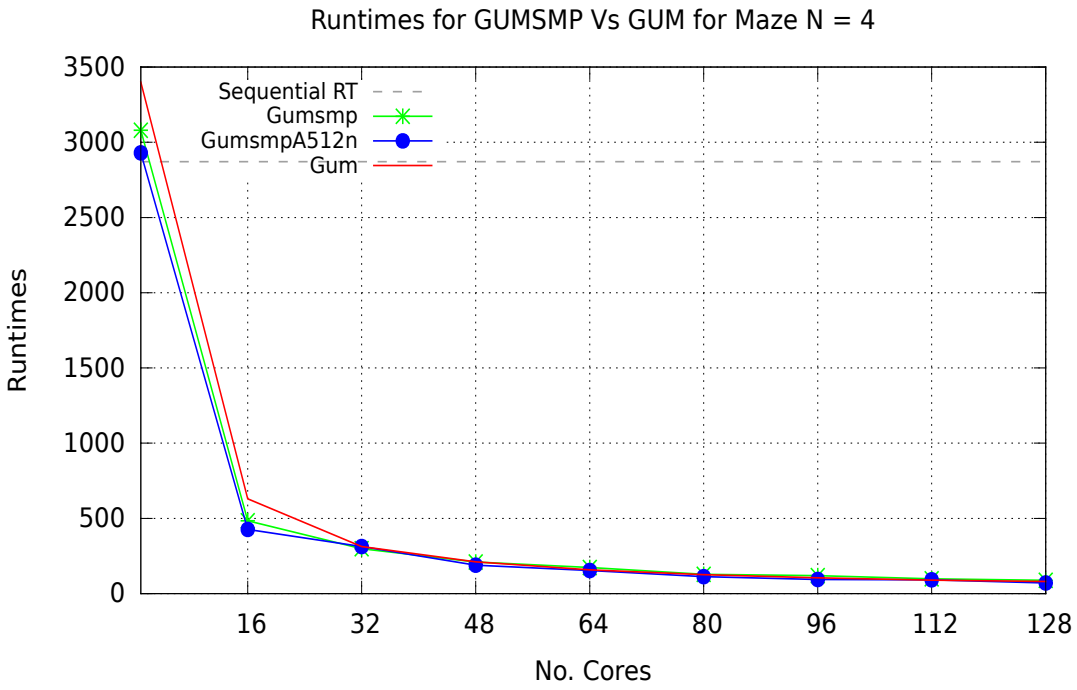

Figure A.5: Runtimes of GHC-GUM vs. GUMSMP with no optimisation

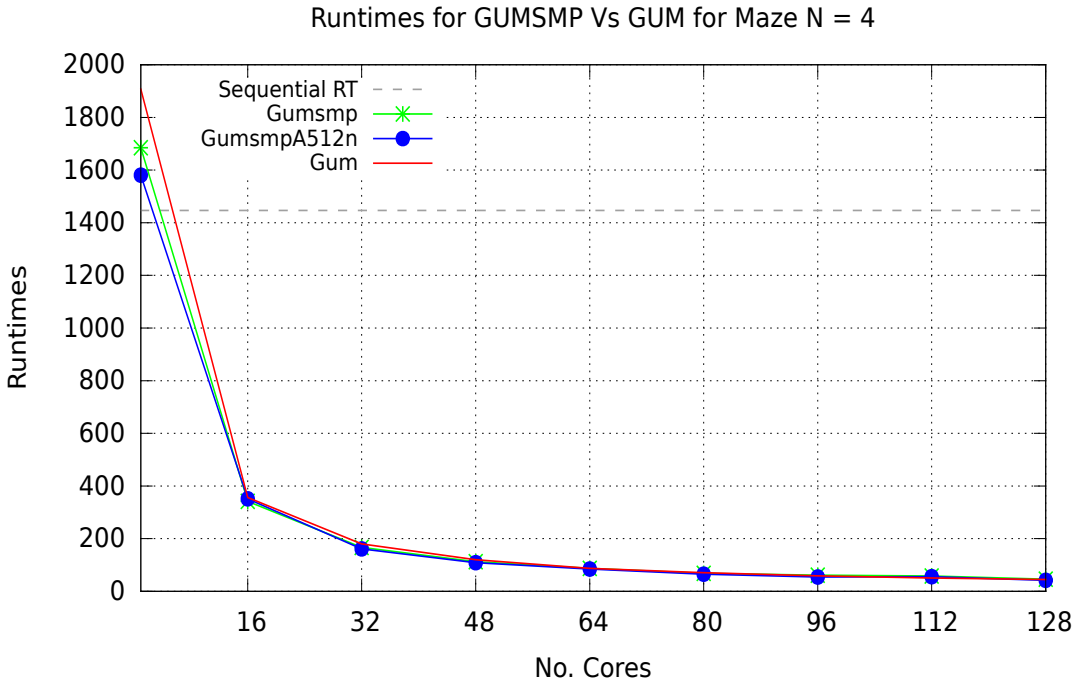

Figure A.6: Runtimes of GHC-GUM vs. GUMSMP with optimisation

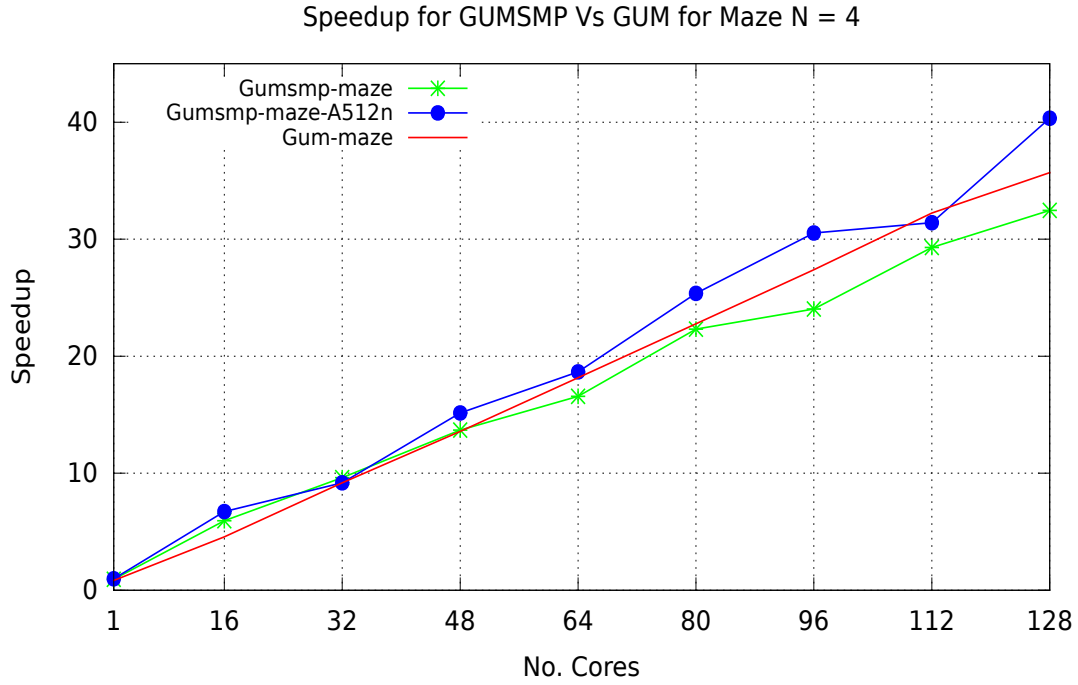

Figure A.7: Speedup of GHC-GUM vs. GUMSMP with no optimisation

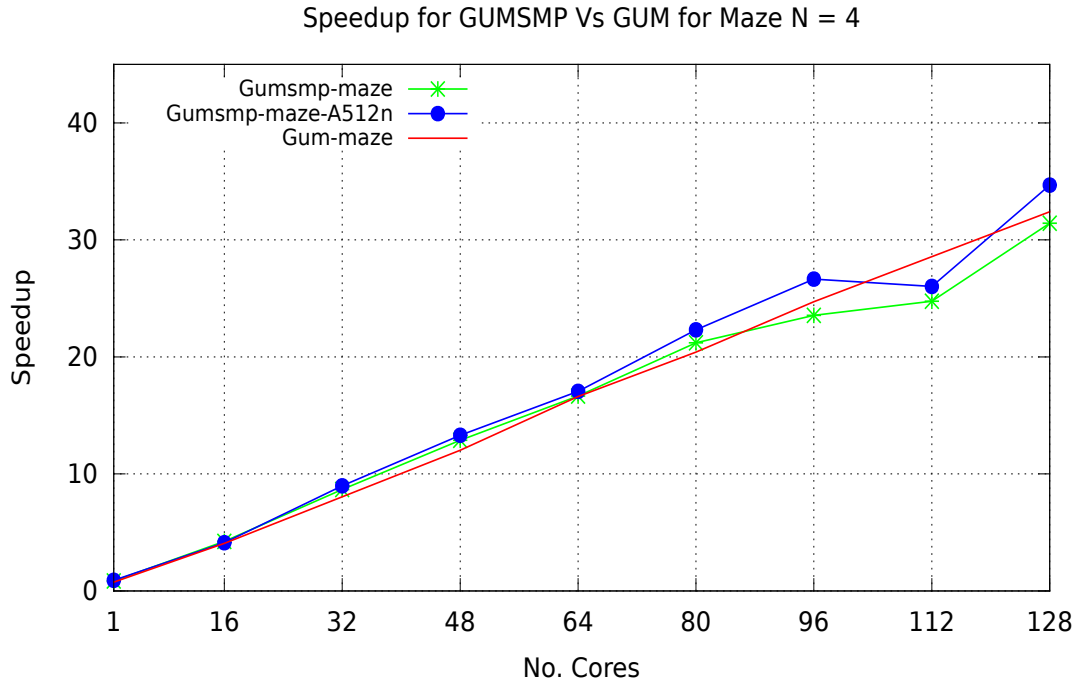

Figure A.8: Speedup of GHC-GUM vs. GUMSMP with optimisation

Appendix B

Benchmarks

These are the set of benchmarks used, which are available with the RTS in: http://www.macs.hw.ac.uk/~hwloidl/hackspace/GUMSMP

Parfib

```
module Main(main) where

import System.Environment (getArgs)
import Control.Parallel
import System.Time
import Text.Printf

main = do [arg1,arg2] <- getArgs

          let
            n = read arg1 :: Int  -- input for parfib
            t = read arg2 :: Int  -- threshold
          t1 <- getClockTime
          let res = parfib n t
          putStrLn ("parfib " ++ show n ++ " = " ++ show res)
          t2 <- getClockTime
          putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

-- parallel version of the code with thresholding
parfib :: Int -> Int -> Int
parfib n t | n <= t = nfib n
           | otherwise = n1 `par` (n2 `pseq` n1 + n2 + 1)
                 where n1 = parfib (n-1) t
          n2 = parfib (n-2) t

-- sequential version of the code
nfib :: Int -> Int
nfib 0 = 1
nfib 1 = 1
```

```
nfib x = nfib (x-2) + nfib (x-1) + 1

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

Parmap-of-parfib

```
import System (getArgs)
import Control.Monad
import Control.Parallel
import System.Time
import Text.Printf


main = do
   args <- getArgs
   unless (length args == 2) $
     error (unlines ["Usage: parfib <m> <n>","  generate <n>
        parallel task computing parfib <m> each"])
   putStrLn ("n = "++ args!!0)
   t1 <- getClockTime
   let m = (read (args!!0))::Integer -- input for parfib
       n = (read (args!!1))::Integer -- number of parfib
          computations
       res = spawnN m n
   putStrLn ("Result: "++(show res))
   t2 <- getClockTime
   putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

-- spawn n instances of parfib m

spawnN :: Integer -> Integer -> Integer
spawnN m 0 = 0
spawnN m n = this `par` (rest `pseq` (rest+this))
  where this = parfib m 23
        rest = spawnN m (n-1)

-- parallel implementation of fibonacci
parfib :: Integer -> Integer -> Integer
parfib n t | n <= t = nfib n
           | otherwise = n1 `par` (n2 `pseq` n1 + n2 + 1)
  where n1 = parfib (n-1) t
        n2 = parfib (n-2) t

nfib :: Integer -> Integer
nfib 0 = 1
nfib 1 = 1
nfib n = nfib (n-1) + nfib (n-2)
```

```
-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

Worpitzky

```
{-# OPTIONS -cpp -fglasgow-exts #-}

module Main(main) where

import System(getArgs)
import Debug.Trace(trace)
import Control.Monad
import GHC.Base(par#, parGlobal#, Int#)
import GHC.Conc(par,pseq)
import System.Time
import Text.Printf

main = do args <- getArgs
          unless (length args == 3) $
            error "Usage: worpitzky <x> <n> <t>  ... testing
               Worpitzky identity for x^n, using t as a threshold"
          let
            x :: Integer
            x = read (args!!0)  -- input for worpitzky
            n :: Integer
            n = read (args!!1)  -- input for worpitzky
            t :: Integer
            t = read (args!!2)  -- threshold
          t1 <- getClockTime
          let res = worpitzky x n t
          putStrLn ("Testing Worpitzky's identity for x^n ..." ++ (
              show x) ++ " " ++ (show n) ++ " " ++ (show t) ++ " = "
              ++ (show res))
          t2 <- getClockTime
          putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

worpitzky ::  Integer -> Integer -> Integer -> Bool
worpitzky x n t = x^n == sum [ (par_euler n k t) * (bin (x+k) n) | k
    <- [0,1..n] ]

-- euler numbers (Concrete Maths, p254)
euler :: Integer -> Integer -> Integer
euler 0 k = if k==0 then 1 else 0
euler n k = (k+1) * (euler (n-1) k) + (n-k)*(euler (n-1) (k-1))

-- binomials
bin :: Integer -> Integer -> Integer
bin n k = fact n `div` ( (fact k) * (fact (n-k)) )
```

```
fact :: Integer -> Integer
fact 0 = 1
fact n = product [1..n]

par_euler :: Integer -> Integer -> Integer -> Integer
par_euler 0 k t = if k==0 then 1 else 0
par_euler n k t | n<t = euler n k
par_euler n k t = e2 `par` (e1 `pseq` ((k+1) * e1 + (n-k)*e2))
                  where e1 = par_euler (n-1) k t
                        e2 = par_euler (n-1) (k-1) t

stir1 :: Integer -> Integer -> Integer
stir1 _ 0 = 1
stir1 0 _ = 1
stir1 n k = (n-1)*(stir1 (n-1) k)+(stir1 (n-1) (k-1))

par_stir1 :: Integer -> Integer -> Integer -> Integer
par_stir1 n k t | n<t || k<t = stir1 n k
                | otherwise  = s1 `par` (s2 `par` (n-1)*s1+s2)
                        where s1 = stir1 (n-1) k
                              s2 = stir1 (n-1) (k-1)

stir2 :: Integer -> Integer -> Integer
stir2 _ 1 = 1
stir2 0 _ = 1
stir2 n k = (stir2 (n-1) (k-1)) + k*(stir2 (n-1) k)

par_stir2 :: Integer -> Integer -> Integer -> Integer
par_stir2 n k t | n<t || k<t = stir2 n k
                | otherwise  = s1 `par` (s2 `pseq` s1+k*s2)
                        where s1 = par_stir2 (n-1) (k-1) t
                              s2 = k*(par_stir2 (n-1) k t)

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

SumEuler

```
{-# OPTIONS -fglasgow-exts #-}

module Main(main) where

import Data.List(transpose)
import System(getArgs)
import GHC.Base(par#, parGlobal#, Int#)
import GHC.Conc(par,pseq)
import Control.Monad
import System.Time
import Text.Printf
```

```
main = do args <- getArgs
          let
            n = read (args!!0) :: Int -- size of the interval
            c = read (args!!1) :: Int -- chunk size
          t1 <- getClockTime
          let res = sumEuler 1 n c
          putStrLn ("sumEuler over an interval from 1 to " ++ (show
              n) ++ " with chunk size " ++ (show c) ++ " = " ++ (show
               res))
          t2 <- getClockTime
          putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2



-- sumEuler over an interval from m to n (with chunk size c)
sumEuler ::  Int -> Int -> Int -> Int
sumEuler m n c | m>n = 0
        | otherwise = let
                               this = sum (map euler [m..m+c-1])
                               rest = sumEuler (min (m+c) (n+1)) n c
                             in
                               this `par` (rest `pseq` this+rest)

sumEuler_seq :: Int -> Int
sumEuler_seq = sum . map euler . enumFromTo 1

euler :: Int -> Int
euler n = let
            relPrimes = let
                          numbers = [1..(n-1)]
                        in
                          {- numbers `par` -} (filter (relprime n)
                             numbers)
          in
            {- (spine relPrimes) `par`-} (length relPrimes)

-- aux fcts
hcf :: Int -> Int -> Int
hcf x 0 = x
hcf x y = hcf y (rem x y)

relprime :: Int -> Int -> Bool
relprime x y = hcf x y == 1

-- strategic functions (could uses Strategies module instead)
parList :: [Int] -> ()
parList = foldr par ()

spine :: [Int] -> ()
spine []     = ()
```

```
spine (_:xs) = spine xs

chunk :: Int -> [a] -> [[a]]
chunk _n [] = []
chunk n  xs = ys:chunk n zs where (ys,zs) = splitAt n xs

slice :: Int ->  [a] -> [[a]]
slice n = transpose . chunk n

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

Coins

```
{-# LANGUAGE BangPatterns #-}

#define STRATEGIES 1

import Data.List
import System.Environment
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
import Control.Parallel.Strategies
import Control.Applicative (Applicative, pure, (<*>))
# elif defined(SM_STRATEGIES)
import Control.Parallel
import Control.DeepSeq
import Control.Parallel.Strategies
# else
import GHC.Conc(par,pseq)
# endif
#endif
import System.Time
import Text.Printf

-------------------------------------------------------------
-- Version 1: returns results as a list of list of coins

payL :: Int -> [(Int,Int)] -> [Int] -> [[Int]]
payL 0   coins     acc = [acc]
payL _   []        acc = []
payL val ((c,q):coins) acc
  | c > val   = payL val coins acc
  | otherwise = left ++ right
  where
    left  = payL (val - c) coins' (c:acc)
    right = payL val coins acc

    coins' | q == 1    = coins
```

```
          | otherwise = (c,q-1) : coins

-------------------------------------------------------
-- Version 2: uses a custom AList type to avoid repeated appends

-- The idea here is that by avoiding the append we might be able to
-- parallelise this more easily by just forcing evaluation to WHNF
   at
-- each level.  I haven't parallelised this version yet, though (V5
-- below is much easier) --SDM

data AList a = ANil | ASing a | Append (AList a) (AList a)

lenA :: AList a -> Int
lenA ANil          = 0
lenA (ASing _)     = 1
lenA (Append l r)  = lenA l + lenA r

append ANil r = r
append l ANil = l -- **
append l r    = Append l r

   -- making append less strict (omit ** above) can make the
      algorithm
   -- faster in sequential mode, because it runs in constant space.
   -- However, ** helps parallelism.

payA :: Int -> [(Int,Int)] -> [Int] -> AList [Int]
payA 0   coins     acc = ASing acc
payA _   []        acc = ANil
payA val ((c,q):coins) acc
  | c > val   = payA val coins acc
  | otherwise = append left right -- strict in l, maybe strict in r
  where
    left  = payA (val - c) coins' (c:acc)
    right = payA val coins acc
    coins' | q == 1    = coins
           | otherwise = (c,q-1) : coins

-------------------------------------------------------
-- Version 3: parallel version of V2

payA_par :: Int -> Int -> [(Int,Int)] -> [Int] -> AList [Int]
payA_par 0     val coins acc = payA val coins acc
payA_par _     0   coins acc = ASing acc
payA_par _     _   []    acc = ANil
payA_par depth val ((c,q):coins) acc
  | c > val    = payA_par depth val coins acc
  | otherwise  = res

  where
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
```

```
    -- res = right `par` left `pseq` append left right
    res = runEval $ pure append <*> rpar left <*> rseq right
    -- res = unEval $ do l <- rpar left; r <- rseq right; return (
       append l r)
# elif defined(SM_STRATEGIES)
    res = right `par` left `pseq` append left right
# else
    res = right `par` left `pseq` append left right
# endif
#else
    res = append left right
#endif

    left  = payA_par (if q == 1 then (depth-1) else depth) (val - c)
        coins' (c:acc)
    right = payA_par (depth-1) val coins acc

    coins' | q == 1    = coins
           | otherwise = (c,q-1) : coins

------------------------------------------------------------
-- Version 4: original list-of-list version (very slow)

pay :: Int -> Int -> [Int] -> [Int] -> [[Int]]
pay _   0 coins accum   = [accum]
pay _   val [] _        = []
pay pri val coins accum =
    res
    where --
          coins'  = dropWhile (>val) coins
          coin_vals = nub coins'
          res = concat ( map
                          ( \ c -> let
                                       new_coins =
                                           ((dropWhile (>c) coins')
                                              \\[c])
                                   in
                                     pay (pri-1)
              (val-c)
                                          new_coins
                                       (c:accum)
                          )
                          coin_vals )


------------------------------------------------------------
-- Version 5: assoc-list version (by HWL)

-- assoc-list-based version; still multiple list traversals
pay1 :: Int -> Int -> [(Int,Int)] -> [(Int,Int)] -> [[(Int,Int)]]
pay1 _   0 coins accum   = [accum]
pay1 _   val [] _        = []
pay1 pri val coins accum = res
```

```
    where --
          coins'  = dropWhile ((>val) . fst) coins
          res = concat (
                        map
                          ( \ (c,q) -> let
                                         -- several traversals
                                         new_coins =
                                           filter (not . (==0) .
                                              snd) $
                                            map (\ x'@(c',q') -> if
                                                c==c' then (c',q
                                                '-1) else x') $
                                             dropWhile ((>c) . fst)
                                                 $
                                              coins'
                                         new_accum =
                                           map (\ x'@(c',q') -> if
                                              c==c' then (c',q'+1)
                                              else x') accum
                                       in
                                        pay1 (pri-1)
                  (val-c)
                                           new_coins
                                         new_accum
                          )
                          coins' )

-----------------------------------------------------------
-- Version 6: just return the number of results, not the results
   themselves

payN :: Int -> [(Int,Int)] -> Int
payN 0   coins     = 1
payN _   []        = 0
payN val ((c,q):coins)
  | c > val   = payN val coins
  | otherwise = left + right
  where
    left  = payN (val - c) coins'
    right = payN val coins

    coins' | q == 1    = coins
           | otherwise = (c,q-1) : coins

-----------------------------------------------------------
-- Version 7: parallel version of payN

payN_par :: Int -> Int -> [(Int,Int)] -> Int
payN_par 0     val coins  = payN val coins
payN_par _     0   coins  = 1
payN_par _     _   []     = 0
payN_par depth val ((c,q):coins)
  | c > val    = payN_par depth val coins
```

```
  | otherwise  = res

  where
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
    -- res = right `par` left `pseq` left + right
    res = runEval $ do
                    r <- rpar right
                    l <- rseq left
                    return (l+r)
# elif defined(SM_STRATEGIES)
    res = right `par` left `pseq` left + right
# else
    res = right `par` left `pseq` left + right
# endif
#else
    res = left + right
#endif

    left  = payN_par (if q == 1 then (depth-1) else depth) (val - c)
        coins'
    right = payN_par (depth-1) val coins

    coins' | q == 1    = coins
           | otherwise = (c,q-1) : coins

------------------------------------------------------------
-- driver

main = do
         let vals = [250, 100, 25, 10, 5, 1]
         let quants = [55, 88, 88, 99, 122, 177]  -- large setup

         let coins  = concat (zipWith replicate quants vals)
             coins1 = zip vals quants
-- n is the version, arg is the value
         [n, arg, dep] <- fmap (fmap read) getArgs
         t1 <- getClockTime

         case n of
           -- sequential, list of results
           1 -> print $ length $ payL arg coins1 []
           -- sequential, append-list of results
           2 -> print $ lenA   $ payA arg coins1 []
           -- parallel, append-list of results
           3 -> print $ lenA   $ payA_par 4 arg coins1 []

           4 -> print $ length (pay 0 arg coins [])
           5 -> print $ length (pay1 0 arg coins1 (map (\(c,q) -> (c
              ,0)) coins1))
           6 -> print $ payN arg coins1
           7 -> print $ payN_par dep arg coins1
         t2 <- getClockTime
```

```
        putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

Minimax

```
module Main where

import System.Environment
import Prog
import Board
import System.Random
import System.Time
import Text.Printf

main = do
  [n, depth] <- fmap (map read) getArgs -- n is the bord size
  t1 <- getClockTime
  setStdGen (mkStdGen 99999)
  b <- randomBoard n
  putStrLn $ showBoard b
  putStrLn $ solve depth b
  t2 <- getClockTime
  putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

```
module Prog(prog,randomBoard,solve) where

import Board
import Wins
import Game
import Tree
import System.Random
import Data.List

-- First arg decaffinates game
prog :: Int -> String
prog decaf = showMove (head game)
        where
        game = if decaf == 0
                  then error "Decaffination error\n"
   else alternate decaf X maxE minE testBoard

-- X to play: find the best move
```

```
solve :: Int -> Board -> String
solve depth board
  = unlines
  . map showMove
  . take 1
  . alternate depth X maxE minE $ board

testBoard = [[Empty,O,Empty,Empty],[Empty,X,Empty,Empty],[Empty,
    Empty,Empty,Empty],[Empty,Empty,Empty,Empty]]

randomBoard :: Int -> IO Board
randomBoard moves =  do
  g <- newStdGen
  let (g1,g2) = split g
      xs = randomRs (1,boardDim) g1
      ys = randomRs (1,boardDim) g2

  let
    play 0 _ _ board = board
    play n (pos:poss) (p:ps) board
     | not (empty pos board) = play n poss (p:ps) board
     | otherwise             = play (n-1) poss ps (placePiece p
        board pos)

  return $ play moves (zip xs ys) (cycle [X,O]) initialBoard
```

```
module Wins where

type Win = [[Int]]

wins :: [Win]
wins = [win1,win2,win3,win4,win5,win6,win7,win8]

win1,win2,win3,win4,win5,win6,win7,win8 :: Win
win1 = [[1,1,1],
        [0,0,0],
        [0,0,0]]

win2 = [[0,0,0],
        [1,1,1],
        [0,0,0]]

win3 = [[0,0,0],
        [0,0,0],
        [1,1,1]]

win4 = [[1,0,0],
        [1,0,0],
        [1,0,0]]

win5 = [[0,1,0],
        [0,1,0],
        [0,1,0]]
```

```
win6 = [[0,0,1],
        [0,0,1],
        [0,0,1]]

win7 = [[1,0,0],
        [0,1,0],
        [0,0,1]]

win8 = [[0,0,1],
        [0,1,0],
        [1,0,0]]
```

```
{-# LANGUAGE CPP #-}
module Tree where

#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
import Strategies
# elif defined(SM_STRATEGIES)
import Control.Parallel
import Control.DeepSeq
import Control.Parallel.Strategies
# else
import Control.Parallel
import Control.Parallel.Strategies
# endif
#else
import GHC.Conc -- hiding (pseq,par)
#endif

data Tree a = Branch a [Tree a] deriving Show

repTree :: (a->[a]) -> (a->[a])-> a -> (Tree a)
repTree f g a = Branch a (map (repTree g f) (f a))

#define SEQ

#ifndef SEQ

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Branch a l)
   = fa `par` Branch fa (map (mapTree f) l `using` myParList)
   where fa = f a

#else  {- SEQ -}

mapTree :: (a -> b) -> (Tree a) -> (Tree b)
mapTree f (Branch a l) = Branch (f a) (map (mapTree f) l)

#endif

#ifndef SEQ
```

```
myParList [] = ()
myParList (x:xs) = x `par` myParList xs

mySeqList [] = ()
mySeqList (x:xs) = x `seq` mySeqList xs

parTree :: Int -> Tree a -> ()
parTree 0 (Branch a xs) = ()
parTree n (Branch a xs) = a `par` mySeqList (map (parTree (n-1)) xs)
#endif

prune :: Int -> (Tree a) -> (Tree a)
prune 0 (Branch a l) = Branch a []
prune (n+1) (Branch a l) = Branch a (map (prune n) l)
```

```
{-# LANGUAGE BangPatterns #-}
module Board where

import Wins

import Data.List
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
import Strategies
# elif defined(SM_STRATEGIES)
import Control.Parallel
import Control.DeepSeq
import Control.Parallel.Strategies
# else
import Control.Parallel
import Control.Parallel.Strategies
# endif
#else
import GHC.Conc
#endif

boardDim = 4

type Board = [Row]
type Row = [Piece]
data Piece = X | O | Empty deriving (Eq,Show)

isEmpty Empty = True
isEmpty _     = False

showBoard :: Board -> String
showBoard board = intercalate "\n--------\n" (map showRow board) ++
   "\n"
 where showRow r = intercalate "|" (map showPiece r)

showPiece :: Piece -> String
showPiece X = "X"
showPiece O = "O"
```

```
showPiece Empty = " "

placePiece :: Piece -> Board -> (Int,Int) -> Board
placePiece new board pos
  = [[ if (x,y) == pos then new else old
     | (x,old) <- zip [1..] row ]
     | (y,row) <- zip [1..] board ]

empty :: (Int,Int) -> Board -> Bool
empty (x,y) board = isEmpty ((board !! (y-1)) !! (x-1))

fullBoard b = all (not.isEmpty) (concat b)

newPositions :: Piece -> Board -> [Board]
newPositions piece board =
    goRows piece id board

goRows p rowsL [] = []
goRows p rowsL (row:rowsR)
  = goRow p rowsL id row rowsR ++ goRows p (rowsL . (row:)) rowsR

goRow p rowsL psL [] rowsR = []
goRow p rowsL psL (Empty:psR) rowsR
    = (rowsL $ (psL $ (p:psR)) : rowsR) : goRow p rowsL (psL . (
       Empty:)) psR rowsR
goRow p rowsL psL (p':psR) rowsR = goRow p rowsL (psL . (p':)) psR
   rowsR

empties board = [ (x,y) | (y,row)   <- zip [1..] board,
                          (x,Empty) <- zip [1..] row ]


initialBoard :: Board
initialBoard = replicate boardDim (replicate boardDim Empty)

data Evaluation = OWin | Score {-# UNPACK #-}!Int | XWin
  -- higher scores denote a board in X's favour
  deriving (Show,Eq)

maxE :: Evaluation -> Evaluation -> Evaluation
maxE XWin _ = XWin
maxE _ XWin = XWin
maxE b OWin = b
maxE OWin b = b
maxE a@(Score x) b@(Score y)  | x>y = a
     | otherwise = b

minE :: Evaluation -> Evaluation -> Evaluation
minE OWin _ = OWin
minE _ OWin = OWin
minE b XWin = b
minE XWin b = b
minE a@(Score x) b@(Score y)  | x<y = a
```

```
    | otherwise = b

eval n | n  == boardDim = XWin
       | -n == boardDim = OWin
       | otherwise      = Score n

static :: Board -> Evaluation
static board = interpret 0 (score board)

interpret :: Int -> [Evaluation] -> Evaluation
interpret x [] = (Score x)
interpret x (Score y:l) = interpret (x+y) l
interpret x (XWin:l) = XWin
interpret x (OWin:l) = OWin

scorePiece X     = 1
scorePiece O     = -1
scorePiece Empty = 0

scoreString !n [] = n
scoreString !n (X:ps)     = scoreString (n+1) ps
scoreString !n (O:ps)     = scoreString (n-1) ps
scoreString !n (Empty:ps) = scoreString n ps

score :: Board -> [Evaluation]
score board =
   [ eval (scoreString 0 row) | row <- board ] ++
   [ eval (scoreString 0 col) | col <- transpose board ] ++
   [ eval (scoreString 0 (zipWith (!!) board [0..])),
     eval (scoreString 0 (zipWith (!!) board [boardDim-1,boardDim-2
         ..])) ]
```

```
module Game where

import Board
import Tree

#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
import Strategies
# elif defined(SM_STRATEGIES)
import Control.Parallel
import Control.DeepSeq
import Control.Parallel.Strategies
# else
import Control.Parallel
import Control.Parallel.Strategies
# endif
#else
import GHC.Conc
#endif

type Player = Evaluation -> Evaluation -> Evaluation
```

```
type Move = (Board,Evaluation)

alternate :: Int -> Piece -> Player -> Player -> Board -> [Move]
alternate _ _ _ _ b | fullBoard b = []
alternate _ _ _ _ b | static b == XWin = []
alternate _ _ _ _ b | static b == OWin = []
alternate depth player f g board = move : alternate depth opponent g
    f board'
 where
 move@(board',eval) = best f possibles scores
        do_scores ps' = case ps' of
                          [] -> []
                          (p:ps) -> (bestMove depth opponent g f p):
                                    do_scores ps
 scores = do_scores possibles
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
                                                        `using` parList
                                                           rseq
# elif defined(SM_STRATEGIES)
                                                        `using` parList
                                                           rwhnf
# else
                                                        `using` parList
                                                           rwhnf
# endif
#endif
 possibles = newPositions player board
        opponent = opposite player

opposite :: Piece -> Piece
opposite X = O
opposite O = X


best :: Player -> [Board] -> [Evaluation] -> Move
best f (b:bs) (s:ss) = best' b s bs ss
 where
 best' b s [] [] = (b,s)
 best' b s (b':bs) (s':ss) | s==(f s s') = best' b s bs ss
      | otherwise  = best' b' s' bs ss

showMove :: Move -> String
showMove (b,e) = show e ++ "\n" ++ showBoard b

bestMove :: Int -> Piece -> Player -> Player -> Board -> Evaluation
bestMove depth p f g
  = parMise 2 f g
  . cropTree
  . mapTree static
  . prune depth
  . searchTree p
```

```
cropTree :: (Tree Evaluation) -> (Tree Evaluation)
cropTree (Branch a []) = (Branch a [])
cropTree (Branch (Score x) l) = Branch (Score x) (map cropTree l)
cropTree (Branch x l) = Branch x []

searchTree :: Piece -> Board -> (Tree Board)
searchTree p board = repTree (newPositions p) (newPositions (
   opposite p)) board

mise :: Player -> Player -> (Tree Evaluation) -> Evaluation
mise f g (Branch a []) = a
mise f g (Branch _ l) = foldr f (g OWin XWin) (map (mise g f) l)

parMise :: Int -> Player -> Player -> (Tree Evaluation) ->
   Evaluation
parMise 0 f g t = mise f g t
parMise n f g (Branch a []) = a
parMise n f g (Branch _ l) = foldr f (g OWin XWin) (map (parMise (n
   -1) g f) l
#if defined(STRATEGIES)
# if defined(EVAL_STRATEGIES)
                                                    `using` parList
                                                       rseq
# elif defined(SM_STRATEGIES)
                                                    `using` parList
                                                       rwhnf
# else
                                                    `using` parList
                                                       rwhnf
# endif
#endif
                                                    )
```

Maze

```
module Main where

import Maze
import System.Time
import Text.Printf
#if defined(STRATEGIES)
#if defined(EVAL_STRATEGIES)
--import Control.Parallel
import Control.Parallel.Strategies
import Control.DeepSeq
#elif defined(SM_STRATEGIES)
import Control.Parallel.Strategies
import Control.Parallel
#else
import Control.Parallel.Strategies
import Control.Parallel
```

```
#endif
#else
import Control.DeepSeq
#endif

main = do
          t1 <- getClockTime
          print (searchp 5 isExit continue sucessors [] node1)
          t2 <- getClockTime
          putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2


#if defined(STRATEGIES)
#if defined(EVAL_STRATEGIES)
parmap f list = map f list `using` parList rdeepseq
#elif defined(SM_STRATEGIES)
parmap f list = map f list `using` parList rdeepseq
#else
parmap f list = map f list  `using` parList rnf
#endif
#else
parmap f list = map f list
#endif

node1 :: Node
node1 = (True,1000,1000)


sucessors (True,1000,1000)=[(False,7,10),(False,6,10),(True,5,9),(
   False,4,10),(False,3,10),(False,2,10),(False,1,10)]
sucessors x = rsucessors 10 x

searchp :: (NFData a)=> Int -> (a -> Bool) -> (a->Bool) -> (a -> [a
   ]) -> [a] -> a -> [a]
searchp 0 isExit continue successors solutions node = search isExit
   continue successors solutions node
searchp t isExit continue successors solutions node
 | isExit node   = node:solutions
 | continue node  = let list = parmap (searchp (t-1) isExit continue
     successors (node:solutions)) (successors node)
       in list `seq` firstSolution list
 | otherwise   = []



search :: (NFData a)=> (a -> Bool) -> (a->Bool) -> (a -> [a]) -> [a]
    -> a -> [a]
search isExit continue successors solutions node
 | isExit node   = node:solutions
 | continue node  = firstSolution (map (search isExit continue
    successors (node:solutions)) (successors node))
 | otherwise   = []
```

```
firstSolution :: [[a]] -> [a]
firstSolution [] = []
firstSolution ([]:xs) = firstSolution xs
firstSolution (x:xs)  = x

-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

```
module Maze where

type Node = (Bool, Int, Int)

rsucessors :: Int -> Node -> [Node]
rsucessors degree (boolv,e,0)= [(boolv,e,0)]
rsucessors degree (True, number ,v) = listofnodes degree (v-1) True
rsucessors degree (False, number,v) = listofnodes degree (v-1) False


listofnodes 1 v vb = [(vb,1,v)]
listofnodes n v vb = (False,n,v) : listofnodes (n-1) v vb


isExit :: Node -> Bool
isExit (True,e,0) = True
isExit _  = False

continue :: Node -> Bool
continue (False,e,0) = False
continue _      = True
```

Mandel

```
module Main where

import Mandel
import PortablePixmap
import System.IO
import System.Environment
import System.Time
import Text.Printf

main =  do
  (minx_: miny_: maxx_: maxy_: screenX_ : screenY_ : limit_ : _) <-
     getArgs -- set window size and number of iterations
  t1 <- getClockTime
  let [minx,miny,maxx,maxy] = map read [minx_,miny_,maxx_,maxy_]
```

```
      [screenX,screenY]     = map read [screenX_,screenY_]
      limit                 = read limit_

  hSetBinaryMode stdout True
  putStr (show (mandelset minx miny maxx maxy screenX screenY limit)
     )
  t2 <- getClockTime
  writeFile "time.txt" (printf "time taken: %.2fs" $ secDiff t1 t2)

readNum::(Num a, Read a) => String -> [String] -> (a->[String]->IO
    ()) -> IO ()
readNum prompt inputLines succ
   = hPutStr stderr prompt >>
     case inputLines of
       (x:xs) -> case (reads x) of
     [(y,"")] -> succ y xs
     _      -> hPutStr stderr "Error - retype the number\n" >>
           readNum prompt xs succ
       _  -> hPutStr stderr "Early EOF"


secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
    - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

```
{-# LANGUAGE BangPatterns #-}
module Mandel where
import Complex -- 1.3
import PortablePixmap
import Control.Parallel
import Control.Parallel.Strategies (using)

default ()


mandel::(Num a) => a -> [a]
mandel c = infiniteMandel
           where
  infiniteMandel = c : (map (\z -> z*z +c) infiniteMandel)


whenDiverge::  Int -> Double -> Complex Double -> Int
whenDiverge limit radius c
  = walkIt (take limit (mandel c))
  where
     walkIt []     = 0      -- Converged
     walkIt (x:xs) | diverge x radius  = 0   -- Diverged
            | otherwise   = 1 + walkIt xs -- Keep walking

-- VERY IMPORTANT FUNCTION: sits in inner loop

diverge::Complex Double -> Double -> Bool
diverge cmplx radius =  magnitude cmplx > radius
```

```
parallelMandel:: [[Complex Double]] -> Int -> Double -> [Int]
parallelMandel mat limit radius
   = concat $

                parBuffer 70
                            [ let l = map (whenDiverge limit radius
                               ) xs
                              in seqList l `pseq` l
                            | xs <- mat ]


parBuffer :: Int -> [a] -> [a]
parBuffer n xs = return xs (start n xs)
  where
    return (x:xs) (y:ys) = y `par` (x : return xs ys)
    return xs [] = xs

    start !n [] = []
    start 0 ys = ys
    start !n (y:ys) = y `par` start (n-1) ys



lazyParList :: Int -> [a] -> [a]
lazyParList !n xs = go xs (parListN n xs)
   where
         go []     _ys    = []
         go (x:xs) []     = x : xs
         go (x:xs) (y:ys) = y `par` (x : go xs ys)

lazyParList1 :: Int -> [a] -> [a]
lazyParList1 !n xs = go xs (parListN1 n xs [])
   where
         go []     _ys    = []
         go (x:xs) []     = x : xs
         go (x:xs) (y:ys) = y `par` (x : go xs ys)


parList :: [a] -> ()
parList [] = ()
parList (x:xs) = x `par` parList xs


parListN :: Int -> [a] -> [a]
parListN 0  xs     = xs
parListN !n []     = []
parListN !n (x:xs) = x `par` parListN (n-1) xs


parListN1 :: Int -> [a] -> [a] -> [a]
parListN1 0  xs     ys = parList ys `pseq` xs
parListN1 !n []     ys = parList ys `pseq` []
```

```
parListN1 !n (x:xs) ys = parListN1 (n-1) xs (x:ys)

seqList :: [a] -> ()
seqList [] = ()
seqList (x:xs) = x `pseq` seqList xs



mandelset::Double ->     -- Minimum X viewport
    Double ->     -- Minimum Y viewport
    Double ->     -- Maximum X viewport
    Double ->    -- maximum Y viewport
    Integer ->     -- Window width
    Integer ->     -- Window height
    Int ->     -- Window depth
    PixMap    -- result pixmap
mandelset x y x' y' screenX screenY lIMIT
   = createPixmap screenX screenY lIMIT (map prettyRGB result)
   where

      windowToViewport s t
           = ((x + (((coerce s) * (x' - x)) / (fromInteger screenX))
              ) :+
       (y + (((coerce t) * (y' - y)) / (fromInteger screenY))))

      coerce::Integer -> Double
      coerce  s   = encodeFloat (toInteger s) 0

      result = parallelMandel
    [[windowToViewport s t | s<-[1..screenX]]
                  | t <- [1..screenY]]
     lIMIT
    ((max (x'-x) (y'-y)) / 2.0)

      prettyRGB::Int -> (Int,Int,Int)
      prettyRGB s = let t = (lIMIT - s) in (s,t,t)
```

```
module PortablePixmap where

data PixMap = Pixmap Integer Integer Int [(Int,Int,Int)]

createPixmap::Integer -> Integer -> Int -> [(Int,Int,Int)] -> PixMap
createPixmap width height max colours = Pixmap width height max
   colours

instance Show PixMap where
 showsPrec prec (Pixmap x y z rgbs) = showHeader x y z . showRGB
    rgbs


showHeader::Integer -> Integer -> Int -> ShowS
showHeader x y z  = showString "P6\n" . showBanner .
     shows x . showReturn .
```

```
     shows y . showReturn .
     shows z . showReturn

showRGB::[(Int,Int,Int)] ->  ShowS
showRGB []   = id
showRGB ((r,g,b):rest)  = showChar (toEnum r) .
     showChar (toEnum g) .
     showChar (toEnum b) .
     showRGB rest

showSpace  = showChar ' '
showReturn = showChar '\n'

showBanner = showString "# Portable pixmap created by Haskell
   Program :\n" .
      showString "#\tPortablePixmap.lhs (Jon.Hill 28/5/92)\n"
```

Blackscholes

```
import Control.Monad hiding (join)
import Data.Array.Unboxed
import System.Environment

import Control.Parallel.Strategies
import Control.DeepSeq
import Future

import System.Time
import Text.Printf

type FpType = Float


data ParameterSet = ParameterSet {
        sptprice :: FpType,
        strike :: FpType,
        rate :: FpType,
        volatility :: FpType ,
        otime :: FpType,
        otype :: Bool
} deriving Show


data_init :: Array Int ParameterSet

-- This defines some hard coded data as a big constant array:
#include "blackscholes_data.hs"
size_init = let (s,e) = bounds data_init in e - s + 1

inv_sqrt_2xPI = 0.3989422804014327028 6
```

```
cndf :: FpType -> FpType
cndf inputX = if sign then 1.0 - xLocal else xLocal
  where
    sign = inputX < 0.0
    inputX' = if sign then -inputX else inputX

    -- Compute NPrimeX term common to both four & six decimal
       accuracy calcs
    xNPrimeofX = inv_sqrt_2xPI * exp(-0.5 * inputX * inputX);

    xK2 = 1.0 / (0.2316419 * inputX + 1.0);
    xK2_2 = xK2 * xK2; -- Need all powers of xK2 from ^1 to ^5:
    xK2_3 = xK2_2 * xK2;
    xK2_4 = xK2_3 * xK2;
    xK2_5 = xK2_4 * xK2;

    xLocal = 1.0 - xLocal_1 * xNPrimeofX;
    xLocal_1 = xK2 * 0.319381530 + xLocal_2;
    xLocal_2 = xK2_2 * (-0.356563782) + xLocal_3 + xLocal_3' +
       xLocal_3'';
    xLocal_3 = xK2_3 * 1.781477937;
    xLocal_3' = xK2_4 * (-1.821255978);
    xLocal_3'' = xK2_5 * 1.330274429;


blkSchlsEqEuroNoDiv :: FpType -> FpType -> FpType ->
   FpType -> Bool -> Float -> FpType
blkSchlsEqEuroNoDiv sptprice strike rate volatility time otype timet
    =
   if not otype
   then (sptprice * nofXd1) - (futureValueX * nofXd2)
   else let negNofXd1 = 1.0 - nofXd1
            negNofXd2 = 1.0 - nofXd2
        in (futureValueX * negNofXd2) - (sptprice * negNofXd1)
 where
   logValues = log( sptprice / strike )
   xPowerTerm = 0.5 * volatility * volatility
   xDen = volatility * sqrt(time)
   xD1 = (((rate + xPowerTerm) * time) + logValues) / xDen
   xD2 = xD1 - xDen

   nofXd1 = cndf xD1
   nofXd2 = cndf xD1
   futureValueX = strike * exp ( -(rate) * (time) )

executeStep :: Int -> Int -> UArray Int FpType
executeStep t granularity =
       listArray (0, granularity-1) $
                 Prelude.map (\i ->
                              let ParameterSet { .. } = data_init !
                                  ((t+i) `mod` size_init)
                             in blkSchlsEqEuroNoDiv sptprice strike rate
                                 volatility otime otype 0)
```

```
                [0 .. granularity-1]

blackscholes :: Int -> Int -> Eval Float
blackscholes numOptions granularity =
   do
      fs <- forM [0, granularity .. numOptions-1] $ \t ->
              fork (return (executeStep t granularity))

      foldM (\ acc f ->
                 do x <- join f
                    return (acc + (x ! 0)))
    0 fs

main = do args <- getArgs
          let (numOptions, granularity) =
               case args of
            []      -> (10000, 1000)
            [b]     -> (10, read b)
          [b,n] -> (read n, read b) -- number of options and
              granularity

          if granularity > numOptions
    then error "Granularity must be bigger than numOptions!!"
    else return ()

          t1 <- getClockTime
   putStrLn$ "Running blackscholes, numOptions "++ show numOptions
      ++ " and block size " ++ show granularity
          let result = runEval $ blackscholes numOptions granularity
   putStrLn$ "Final result: "++ show result
   return result
          t2 <- getClockTime
          putStrLn $ printf "time taken: %.2fs" $ secDiff t1 t2


-- func to calc time taken between t0 and t1 in sec
secDiff::ClockTime -> ClockTime -> Float
secDiff (TOD secs1 psecs1) (TOD secs2 psecs2) = fromInteger (psecs2
   - psecs1) / 1e12 + fromInteger (secs2 - secs1)
```

```
module Future (Eval(..), Future, runEval, rseq, fork, join, deep)
   where

import Control.DeepSeq
import Control.Parallel
import Control.Parallel.Strategies

data Future a = Future a

fork :: Eval a -> Eval (Future a)
fork a = do a' <- rpar (runEval a); return (Future a')

join :: Future a -> Eval a
```

```
join (Future a) = a `pseq` return a

deep :: NFData a => Eval a -> Eval a
deep m = do a <- m; rnf a `pseq` return a
```

Bibliography

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[2] A. Al Zain. *Implementing High-Level Parallelism on Computational GRIDs*. PhD thesis, School of Mathematical and Computer Sciences, 2006.

[3] A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing heterogeneity in a grid parallel haskell. In V. Sunderam, G. van Albada, P. Sloot, and J. Dongarra, editors, *Computational Science ICCS 2005*, volume 3515 of *Lecture Notes in Computer Science*, pages 746–754. Springer Berlin Heidelberg, 2005.

[4] A. Al Zain, P. Trinder, G. Michaelson, and H.-W. Loidl. Evaluating a high-level parallel language (gph) for computational grids. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

[5] M. Aljabri, H.-W. Loidl, and P. Trinder. Overview of the Design of GUMSMP: a Multilevel Parallel Haskell Implementation. In *Proceedings of the 5th Saudi Scientific International Conference 2012*, SIC '12, page 25. Saudi Scientific International Conference, 2012.

[6] M. Aljabri, H.-W. Loidl, and P. Trinder. The Design and Implementation of GUMSMP: A Multilevel Parallel Haskell Implementation. In *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages*, IFL '13, pages 37–48, New York, NY, USA, 2014. ACM.

[7] M. Aljabri, H.-W. Loidl, and P. Trinder. Assessing the Scalability Issues on Many-Core NUMA machines. In *the Proceedings of the 8th Saudi Students Conference.*, SCC '15. Saudi Scientific International Conference, 2015.

[8] M. Aljabri, H.-W. Loidl, and P. Trinder. Balancing Shared and Distributed Heaps on NUMA Architectures. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, volume 8843 of *Lecture Notes in Computer Science*, pages 1–17. Springer International Publishing, 2015.

[9] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. Steele, and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, mar 2008.

[10] K. Alnowaiser. A study of connected object locality in numa heaps. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 1:1–1:9, New York, NY, USA, 2014. ACM.

[11] C. Amza, A. Cox, H. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, 1996.

[12] M. Armbrust, A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[13] 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. *Commun. ACM*, 52(10):56–67, Oct. 2009.

[14] M. Aswad, P. Trinder, A. Al Zain, G. Michaelson, and J. Berthold. Low pain vs no pain multi-core Haskells. In *TFP'09 — Symp. on Trends in Functional Programming*, volume 10 of *Trends in Functional Programming*, pages 49–64, Komarno, Slovakia, June 2009. Intellect.

[15] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 51–57, New York, NY, USA, 2011. ACM.

[16] B. Barney. Introduction to Parallel Computing, 2010. https://computing.llnl.gov/tutorials/parallel_comp/#ModelsData.

[17] O. Batchelor and R. Green. Cloud Haskell: First impressions and applications to processing large image datasets. In *Image and Vision Computing New Zealand (IVCNZ), 2013 28th International Conference of*, pages 412–417, Nov 2013.

[18] 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, Department of Computer Science, Heriot-Watt University, December 2013.

[19] R. Benner, V. Echeverria, U. Onunkwo, J. Patel, and D. Zage. Harnessing Manycore Processors for Scalable, Highly Efficient, and Adaptable Firewall Solutions. In *2013 International Conference on Computing, Networking and Communications (ICNC)*, pages 637–641, Jan 2013.

[20] L. Bergstrom. Measuring numa effects with the stream benchmark. *CoRR*, abs/1103.3225, 2011.

[21] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical Master-Worker Skeletons. In P. Hudak and D. S. Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2008.

[22] J. Berthold, H.-W. Loidl, and K. Hammond. PAEAN: Portable Runtime Support for Physically-Shared-Nothing Architectures in Parallel Haskell Dialects. *Journal of Functional Programming*, 2015. To appear.

[23] J. Berthold, S. Marlow, K. Hammond, and A. Al Zain. Comparing and Optimising Parallel Haskell Implementations for Multicore Machines. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 386–393, Washington, DC, USA, 2009. IEEE Computer Society.

[24] G. Blelloch. Programming Parallel Algorithms. *Commun. ACM*, 39(3):85–97, Mar. 1996.

[25] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[26] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[27] R. D. Blumofe and P. A. Lisiecki. Adaptive and reliable parallel computing on networks of workstations. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '97, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association.

[28] C. Boyd. Data-parallel computing. *Queue*, 6(2):30–39, Mar. 2008.

[29] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in Heterogeneous Computing. *Sci. Program.*, 18(1):1–33, Jan. 2010.

[30] R. Buchty, V. Heuveline, W. Karl, and J.-P. Weiss. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurrency and Computation: Practice and Experience*, 24(7):663–675, 2012.

[31] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM.

[32] D. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[33] M. Chakravarty and G. Keller. More Types for Nested Data Parallel Programming. In *In Proceedings ICFP 2000: International Conference on Functional Programming*, pages 94–105. ACM Press, 2000.

[34] M. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel. Nepal - Nested Data Parallelism in Haskell. In *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, Euro-Par '01, pages 524–534, London, UK, UK, 2001. Springer-Verlag.

[35] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Intl. J. High Perform. Comput. Appl.*, 21:291–312, Aug 2007.

[36] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[37] B. Chapman, G. Jost, and R. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[38] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, 40(10):519–538, Oct. 2005.

[39] D. Chavarria-Miranda, S. Krishnamoorthy, and A. Vishnu. Global Futures: A Multithreaded Execution Model for Global Arrays-based Applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 393–401, 2012.

[40] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and Its First Implementation: A Performance View. *IBM J. Res. Dev.*, 51(5):559–572, Sept. 2007.

[41] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[42] D. Culler, S. Goldstein, K. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, June 1993.

[43] M. Danelutto and M. Vanneschi. Parallel Programming Issues, Achievements and Trends in High-Performance and Adaptive Computing. In *1st Internal Conference on What is Going on and What is Next?*, number 2 in WiGoWiN, pages 16–18, Pisa, Italy, May 2010.

[44] R. Daniel, B. Donald, M. Phillip, and S. Thomas. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace Proceedings*, pages 79–91, 1997.

[45] J. Darlington, Y. Guo, H. To, and J. Yang. Parallel Skeletons for Structured Composition. In J. Ferrante, D. A. Padua, and R. L. Wexelblat, editors, *PPOPP*, pages 19–28. ACM, 1995.

[46] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, Jan 2008.

[47] F. Denneman. AMD Magny-Cours and ESX, 2011. http://frankdenneman.nl/2011/01/05/amd-magny-cours-and-esx/.

[48] F. Desprez, E. Fleury, A. Kalinov, and A. Lastovetsky. *Algorithms and Tools for Parallel Computing on Heterogeneous Clusters*. Nova Science Publishers, 2007.

[49] J. Diaz, C. Munoz-Caro, and A. Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 23(8):1369–1386, 2012.

[50] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable work stealing. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 53:1–53:11, New York, NY, USA, 2009. ACM.

[51] A. Du Bois, H.-W. Loidl, and P. Trinder. Thread migration in a parallel graph reducer. In *Proceedings of the 14th International Conference on Implementation of Functional Languages*, IFL'02, pages 199–214, Berlin, Heidelberg, 2003. Springer-Verlag.

[52] R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5–16, Feb. 1990.

[53] A. Duran, E. Ayguad, R. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, 2011.

[54] T. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[55] J. Epstein. Functional programming for the data centre. Master's thesis, University of Cambridge, 2011.

[56] J. Epstein, A. P. Black, and S. Peyton Jones. Towards Haskell in the Cloud. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

[57] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[58] J. Falcou. Parallel Programming with Skeletons. *Computing in Science and Engineering*, 11(3):58–63, 2009.

[59] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20:537–576, 11 2010.

[60] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: a Heterogeneous Parallel Language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, DAMP '07, pages 37–44, New York, NY, USA, 2007. ACM.

[61] M. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21(9):948–960, Sept. 1972.

[62] A. Foltzer, A. Kulkarni, R. Swords, S. Sasidharan, E. Jiang, and R. Newton. A Meta-scheduler for the Par-monad: Composable Scheduling for the Heterogeneous Cloud. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 235–246, New York, NY, USA, 2012. ACM.

[63] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering.* Addison-Wesley Longman, Boston, MA, USA, 1995.

[64] I. Foster and C. Kesselman. *The Grid2 : Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[65] T. Fountain and P. Kacsuk. *Advanced Computer Architectures: A Design Space Approach.* Pearson Education, 1997.

[66] R. Fowler and C. Greenough. Experiences with Globus and MPICH-G. Technical report, Rutherford Appleton Laboratory, 2001.

[67] V. Freeh. A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*, 34(1):50 – 65, 1996.

[68] V. Freeh, D. Lowenthal, and G. Andrews. Distributed filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[69] B. Freisleben and T. Kielmann. Approaches to support parallel programming on workstation clusters: A survey. *A Survey, Informatik Berichte, Fachgruppe Informatik, Universitat-GH Siegen*, 95, 1995.

[70] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[71] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Mass., 1994.

[72] D. Gelernter and N. Carriero. Coordination Languages and Their Significance. *Commun. ACM*, 35(2):97–107, Feb. 1992.

[73] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the Scalability of Garbage Collectors on Many Cores. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems*, PLOS '11, pages 7:1–7:5, New York, NY, USA, 2011. ACM.

[74] P. Glaskowsky. Tilera's Balancing Act: 100 Cores Vs. Market Realities, 2009. http://www.cnet.com/uk/news/tileras-balancing-act-100-cores-vs-market-realities.

[75] S. Goldstein, K. Schauser, and D. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[76] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[77] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Comput.*, 22(6):789–828, Sep 1996.

[78] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[79] K. Hammond. Why parallel functional programming matters: Panel statement. In A. Romanovsky and T. Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 201–205. Springer Berlin Heidelberg, 2011.

[80] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *Haskell '05: Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[81] J. Hennessy and D. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[82] S. Herb. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[83] L. Hertzberger. Trends in Parallel and Distributed Computing. *Future Generation Computer Systems*, 7(1):31–40, Oct 1991.

[84] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[85] Y. Hua, H. Lub, A. Coxc, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[86] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill Higher Education, 1st edition, 1992.

[87] V. Janjic. *Load balancing of irregular parallel applications on heterogeneous computing environments.* PhD thesis, Department of Computing Science, University of St Andrews, 2012.

[88] W. Jie, W. Cai, and S. Turner. Poems: A parallel object-oriented environment for multi-computer systems. *Comput. J.*, 45(5):540–560, 2002.

[89] L. Kale and S. Krishnan. CHARM++: a Portable Concurrent Object-oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, NY, USA, 1993. ACM.

[90] R. Karp and Y. Zhang. A randomized parallel branchandbound procedure. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 290–300, New York, NY, USA, 1988. ACM.

[91] H. Kasim, V. March, R. Zhang, and S. See. Survey on Parallel Programming Model. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 266–275, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] R. Khan and M. Ali. Current Trends in Parallel Computing. *International Journal of Computer Applications*, 59(2):19–25, December 2012.

[93] D. A. Kranz, R. H. Halstead, Jr., and E. Mohr. Mul-T: a High-Performance Parallel Lisp. *SIGPLAN Not.*, 24(7):81–90, July 1989.

[94] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.

[95] I. Kuon, R. Tessier, and J. Rose. FPGA Architecture: Survey and Challenges. *Found. Trends Electron. Des. Autom.*, 2(2):135–253, Feb. 2008.

[96] J. Labarta. Starss: a programming model for the multicore era - prace, 2010. http://www.prace-ri.eu/IMG/pdf/08_starss_jl.pdf.

[97] C. Lameter. NUMA (Non-Uniform Memory Access): An Overview. *Queue*, 11(7):40:40–40:51, Jul 2013.

[98] E. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[99] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming*

Systems Languages and Applications, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM.

[100] D. Lenoski and W. Weber. *Scalable Shared-Memory Multiprocessing*. Elsevier Science, 2014.

[101] D. Lester. An efficient distributed garbage collection algorithm. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89 Parallel Architectures and Languages Europe*, volume 365 of *Lecture Notes in Computer Science*, pages 207–223. Springer Berlin Heidelberg, 1989.

[102] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Work stealing and persistence-based load balancers for iterative overdecomposed applications. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 137–148, New York, NY, USA, 2012. ACM.

[103] B. Liu, D. Zydek, H. Selvaraj, and L. Gewali. Accelerating High Performance Computing Applications: Using CPUs, GPUs, Hybrid CPU/GPU, and FPGAs. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 337–342, Dec 2012.

[104] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, mar 1998.

[105] H.-W. Loidl. Load Balancing in a Parallel Graph Reducer. In K. Hammond and S. Curtis, editors, *SFP'01 — Scottish Functional Programming Workshop*, volume 3 of *Trends in Functional Programming*, pages 63–74, Bristol, UK, 2001. Intellect.

[106] H.-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *International Symposium on Cluster Computing and the Grid*, CCGrid 2002, pages 311–318, Berlin, Germany, May 2002. IEEE Computer Society.

[107] H.-W. Loidl, U. Klusik, K. Hammond, R. Loogen, and P. Trinder. GpH and Eden: Comparing two parallel functional languages on a beowulf cluster. In *SFP'00 — Scottish Functional Programming Workshop*, volume 2 of *Trends in Functional Programming*, pages 39–52, St Andrews, Scotland, July 2000. Intellect.

[108] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, A. J. Rebón, and P. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, Sept. 2003.

[109] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel Functional Programming in Eden. *J. Funct. Program.*, 15:431–475, May 2005.

[110] K. Louden. *Programming Languages: Principles and Practices*. Advanced Topics Series. Cengage Learning, 2011.

[111] D. Lowenthal, V. Freeh, and G. Andrews. Using Fine-grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1), 1996.

[112] D. Lowenthal, V. Freeh, and G. Andrews. Efficient support for fine-grain parallelism on shared-memory machines. *Concurrency - Practice and Experience*, 10(3):157–173, 1998.

[113] P. Maier and P. Trinder. Implementing a High-Level Distributed-Memory Parallel Haskell in Haskell. In A. Gill and J. Hage, editors, *23rd International Symposium on Implementation and Application of Functional Languages, IFL*, volume 7257 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2011.

[114] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel Generational-copying Garbage Collection with a Block-structured Heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 11–20, New York, NY, USA, 2008. ACM.

[115] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: Better Strategies for Parallel Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 91–102, New York, NY, USA, 2010. ACM.

[116] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82, New York, NY, USA, 2011. ACM.

[117] S. Marlow and S. Peyton Jones. Multicore Garbage Collection with Local Heaps. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 21–32, New York, NY, USA, 2011. ACM.

[118] S. Marlow and S. Peyton Jones. *The Architecture of Open Source Applications, Vol 2*, chapter The Glasgow Haskell Compiler. lulu.com, 2012.

[119] S. Marlow, S. Peyton Jones, and S. Singh. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 65–78, New York, NY, USA, 2009. ACM.

[120] M. Martin, M. Hill, and D. Sorin. Why On-chip Cache Coherence is Here to Stay. *Commun. ACM*, 55(7):78–89, July 2012.

[121] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[122] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin. Programming the Intel 80-core Network-on-a-chip Terascale Processor. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 38:1–38:11, Piscataway, NJ, USA, 2008. IEEE Press.

[123] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, 16(2-3):181–206, Aug 2001.

[124] E. Mohr, D. Kranz, and R. Halstead, Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Trans. Parallel Distrib. Syst.*, 2:264–280, July 1991.

[125] MPI Forum. *MPI: A Message-Passing Interface Standard version: 3.0*, sep 2012. http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[126] M. Nosrati and R. Karimi. Occam: A Primary Parallel Programming Language. *General Scientific Researches*, 1(1):1–3, 2013.

[127] R. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[128] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Sebastopol, CA, 3 edition, 2004.

[129] M. Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[130] A. Omicini and M. Viroli. Review: Coordination Models and Languages: From Parallel Computing to Self-organisation. *Knowl. Eng. Rev.*, 26(1):53–59, Feb. 2011.

[131] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[132] G.-R. Perrin and A. Darte, editors. *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, volume 1132 of *Lecture Notes in Computer Science*. Springer, 1996.

[133] S. Peyton Jones. Parallel Implementations of Functional Programming Languages. *Comput. J.*, 32:175–186, April 1989.

[134] S. Peyton Jones, C. Hall, K. Hammond, J. Cordy, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview, 1992.

[135] S. Peyton Jones, R. Leschinsky, G. Gabriele Keller, and M. Chakravarty. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *FSTTCS'08: Foundations of Software Technology and Theoretical Computer Science*, pages 383–414, Bangalore, India, 2008.

[136] M. Philippsen. JavaParty. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 992–997. Springer US, 2011.

[137] K. Pingali. Parallel Programming Languages. Technical report, Cornell University, 1998.

[138] M. Poldner and H. Kuchen. Skeletons for Divide and Conquer Algorithms. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2008)*, Innsbruck,Austria, 2008.

[139] J. Potter. *The Massively Parallel Processor*. The MIT Press,Cambridge, MA, United States, Jan 1985.

[140] C. Quammen. Introduction to Programming Shared-Memory and Distributed-Memory Parallel Computers. *Crossroads*, 12(1):2–2, Oct. 2005.

[141] F. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, UK, 2003.

[142] T. Rauber and G. Rnger. *Parallel Programming - for Multicore and Cluster Systems*. Springer, 2010.

[143] J. H. Reppy. Concurrent ML: Design, application and semantics. In P. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, LNCS 693, pages 165–198. Springer-Verlag, 1993.

[144] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the Third Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '91, pages 237–245, New York, NY, USA, 1991. ACM.

[145] ScaleMP, Inc. http://www.scalemp.com.

[146] D. Schmidl, C. Terboven, A. Wolf, D. a. Mey, and C. Bischof. How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 29–37, Washington, DC, USA, 2010. IEEE Computer Society.

[147] S.-B. Scholz. Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[148] M. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[149] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-core x86 Architecture for Visual Computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[150] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[151] L. Silva and R. Buyya. *Parallel Programming Models and Paradigms*, volume 2, chapter 1, pages 4–27. Prentice Hall PTR, NJ, USA, 1999.

[152] D. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[153] J. B. Smith. *Practical OCaml (Practical)*. Apress, Berkely, CA, USA, 2006.

[154] M. Sottile, T. G. Mattson, and C. E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC, 1st edition, 2009.

[155] R. Stewart, P. Trinder, and P. Maier. Supervised Workpools for Reliable Massively Parallel Computing. In H.-W. Loidl and R. Pea, editors, *13th International Symposium on Trends in Functional Programming, TFP*, volume 7829 of *Lecture Notes in Computer Science*, pages 247–262. Springer Berlin Heidelberg, 2013.

[156] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. de Supinski. Critical Path-based Thread Placement for NUMA Systems. *SIGMETRICS Perform. Eval. Rev.*, 40(2):106–112, Oct. 2012.

[157] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, Sept. 2005.

[158] D. Syme, A. Granicz, and A. Cisternino. *Expert F# 3.0*. Expert's voice in F#. Apress, 2012.

[159] G. Taboada, J. Tourio, and R. Doallo. High Performance Java Sockets for Parallel Computing on Clusters. In *IEEE International Symposium in Parallel and Distributed Processing*, IPDPS 2007, pages 1–8, March 2007.

[160] L. Tan, R. Yufei, Y. Dantong, J. Shudong, and T. Robertazzi. Characterization of Input/Output Bandwidth Performance Models in NUMA Architecture for Data Intensive Applications. In *42nd International Conference on Parallel Processing (ICPP)*, pages 369–378, Oct 2013.

[161] C. Terboven, D. Schmidl, T. Cramer, and D. an Mey. Assessing openmp tasking implementations on numa architectures. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 182–195, Berlin, Heidelberg, 2012. Springer-Verlag.

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

[163] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a portable implementation of Haskell. In *IFL'95 —International Workshop on the Implementation of Functional Languages*, Bastad, Sweden, Sep 1995.

[164] P. Trinder, K. Hammond, J. Mattson Jr., A. Partridge, and S. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Languages Design and Implementation*, pages 79–88, Philadelphia, PA, USA, May 1996.

[165] P. Trinder, H.-W. Loidl, and K. Hammond. Parallel functional languages. In D. Padua, editor, *Encyclopedia of Parallel Computing*, Springer Reference. Springer, New York, NY, USA, 2011.

[166] P. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed haskells. *J. Funct. Program.*, 12(5):469–510, July 2002.

[167] N. Tuan-Anh and K. Pierre. ParoC++: A Requirement-Driven Parallel Object-Oriented Programming Language. *The 8th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, 2003.

[168] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[169] P. Vabishchevich. *Computational Technologies: Advanced Topics*. De Gruyter Textbook. De Gruyter, 2015.

[170] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 34–43, New York, NY, USA, 2001. ACM.

[171] A. Varbanescu, P. Hijma, R. van Nieuwpoort, and H. Bal. Towards an effective unified programming model for many-cores. In *IPDPS Workshops*, pages 681–692. IEEE, 2011.

[172] R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1996.

[173] B. Wilson. Introduction to Parallel Programming Using Message-Passing. *J. Comput. Sci. Coll.*, 21(1):207–211, Oct 2005.

[174] X. Wu and V. Taylor. Using Processor Partitioning to Evaluate the Performance of MPI, OpenMP and Hybrid Parallel Applications on Dual- and Quad-core Cray XT4 Systems. In *Proceedings of the 2009 Cray Users' Group Meeting*, Atlanta, GA, May 2009.

[175] C. Yang, C. Huang, and C. Lin. Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters. *Computer Physics Communications*, 182(1):266 – 269, 2011.

[176] E. Z. Yang. The GHC Runtime System. http://ezyang.com/jfp-ghc-rts-draft.pdf, July 2013.

[177] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.

[178] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé. Periodic hierarchical load balancing for large supercomputers. *Int. J. High Perform. Comput. Appl.*, 25(4):371–385, Nov. 2011.

Glossary

ACK An acknowledgement message. 95

API Application Programming Interface. 38, 61

BH BlackHole, a closure that is under evaluation. 74, 76, 99–101 73, 74, 98.

BlockedFetch A closure that is under evaluation when fetch message received. 95, 100

closure A node in the graph structure. 73–75, 86, 88, 95–101

core An independent hardware unit for computation, closely related to the software defined notion of HEC. 2, 9, 11, 13–18, 20, 22–28, 32, 36, 39–41, 43, 44, 47, 48, 54–56, 59, 61, 62, 65, 67–72, 79, 80, 82, 84, 91, 104, 106–112, 116–121, 125, 127–135, 138–140, 142–148, 150–158, 160–165, 167, 168

CUDA Compute Unified Device Architecture. 40

DPH Data Parallel Haskell. 55

FETCH A data request message. 83, 95, 98–100

FetchMe A global indirections to remote closure. 74, 88, 95, 96, 98–100

FIFO First In First Out. 70–73, 75

FISH A work request message. 77, 78, 80–82, 84, 98, 100, 111, 112, 148, 150, 167

FPGA Field Programmable Gate Array. 16, 22, 27

GA Global Address, a globally unique identifier. 88, 95–98, 121, 123, 125

GC Garbage Collection. 11, 66, 79, 88–93, 114, 128, 139, 142–146, 148, 155–157

GHC Glasgow Haskell Compiler. 2, 16, 55, 56, 58–60, 68, 106–108, 132, 145

GHC-GUM Parallel Haskell implementation for Distributed Memory. 2, 8–13, 16–18, 42, 55, 56, 58–62, 65–70, 73, 74, 76, 77, 80, 81, 88, 89, 92, 93, 95, 103, 108–110, 121, 129, 131, 134, 135, 138, 147–151, 153–157, 164, 165, 170–173

GHC-SMP Parallel Haskell implementation for Shared Memory. 2, 8, 11–13, 16–18, 42, 59–62, 65–68, 70, 73, 76, 78–81, 89, 90, 92, 93, 103, 106, 109, 110, 128, 131–135, 138, 142, 144, 145, 148, 164, 165

GIT Global Indirection Table, a mapping of the GAs to LAs and vice versa. 59, 84, 88, 121

GpH Glasgow Parallel Haskell. 12, 15–17, 30, 31, 35, 38, 41, 46–48, 53, 55, 59, 65, 66, 119, 129, 133, 134, 144, 148, 162, 164

GPU Graphics Processing Unit. 16, 22, 24, 27, 40, 41, 60

GRIP Graph Reduction In Parallel, a dedicated graph reduction hardware. 71

GUMSMP Our parallel Haskell implementation that combines the shared and distributed memory implementations. 2, 8–18, 20, 26, 30, 41, 42, 60–62, 66–74, 76, 80–83, 85, 86, 89, 92–95, 98, 103–105, 108–111, 113, 121, 122, 126, 128, 129, 131, 133, 135–140, 142–144, 147–166, 168, 170–173

HdpH High-level Distributed Memory parallel Haskell in Haskell. 12, 46, 47, 52, 53, 55, 58, 59, 61

HEC Haskell Execution Context, An independent software unit for computation, closely related to the hardware defined notion of core. 8, 59, 72, 73, 75, 78–83, 86, 87, 89, 91–93, 109, 112, 114, 119, 121, 122, 126–129, 148–151, 154, 158, 160

HPC High Performance Computing. 20, 27, 40, 64

HW Hardware. 72

IND An indirection closure to result. 99, 100

IO Input/Output. 24–26, 28

LA Local Address, a locally unique identifier. 88

MIMD Multiple Instruction Multiple Data. 22, 23, 26

MISD Multiple Instruction Single Data. 22

MPI Message Passing Interface. 15, 20, 31, 35, 37, 38, 40, 48, 56

MPP Massively Parallel Processors. 26

node A multi-core machine, closely related to the software defined notion of PE. 2, 11, 13, 15, 17, 18, 34, 40, 41, 54, 58, 67, 68, 70, 72, 80, 82, 83, 103, 106, 109, 110, 112, 119–121, 126–130, 160, 161, 163–165

NUMA Non-Uniform Memory Access. 2, 7, 9, 11–13, 15–18, 24–26, 40, 62, 89, 104, 127, 128, 131–133, 138–140, 142–148, 151, 152, 163, 165, 166, 168

OpenMP Open Multiprocessing. 15, 31, 35, 37–41, 64, 147

OS Operating System. 71, 72

PE Processing Element, a computation engine with executor(s) (cores), memory, and other resources, e.g. file handlers etc., closely related to the hardware defined notion of node. 18, 28, 29, 31–34, 38, 46, 57–64, 71–75, 77, 78, 80–84, 86–88, 93, 95, 97–100, 104, 109–112, 114, 116, 119, 121–123, 126–129, 134, 135, 138, 142, 144, 148–155, 157, 160, 165–168

PGAS Partitioned Global Address Space. 31, 41, 64

PVM Parallel Virtual Machine. 31, 35, 37, 38, 56, 70, 106, 160

RBH Revertible BlackHole, a closure that has been exported to remote PE, but for which no location on the remote PE has been received. 75, 97–99, 101

RESUME A data delivery message. 95, 100

RTS Runtime System, used interchangeably with RTE(Runtime Environment). 2, 15, 30, 35, 41, 47, 48, 56, 60, 66, 68, 71, 72, 84, 90, 96, 106, 117, 125, 126, 132, 145, 147, 148, 154, 163, 164, 166, 167

SAC Single Assignment C. 45

SCHEDULE A reply to Fish message with work. 77, 84, 100, 167

SIMD Single Instruction Multiple Data. 22

SISD Single Instruction Single Data. 22

SMP Symmetric Multi-Processor systems. 24

SO Stack Object. 73

spark A potential parallelism as a pointer to un-evaluated graph structure "thunk". 2, 13, 18, 42, 50, 53, 58–60, 67, 71–73, 75, 77–87, 100, 112, 114, 119–123, 125–128, 130, 149, 150, 158, 161, 165, 167, 168

thunk An un-evaluated graph structure. 11, 71, 73, 74, 88, 96–101, 149

TSO Thread State Object, a representation of task in the RTS. 73, 74, 77, 79, 91

UMA Uniform Memory Access. 24

UPC Unified Parallel C. 41

WHNF Weak Head Normal Form. 48, 49, 79, 80