The Design and Implementation of GUMSMP:

a Multilevel Parallel Haskell Implementation

Malak Aljabri
School of Mathematical and Computer Sciences, Heriot-Watt University
ma767@hw.ac.uk

Hans-Wolfgang Loidl
School of Mathematical and Computer Sciences, Heriot-Watt University
H.W.Loidl@hw.ac.uk

Phil W. Trinder
School of Computing Science, College of Science and Engineering, Glasgow University
Phil.Trinder@glasgow.ac.uk

Abstract

The most widely available high performance platforms today are multi-level clusters of multi-cores. The Glasgow Haskell Compiler (GHC) provides a number of parallel Haskell implementations targeting different parallel architectures. In particular, GHC-SMP supports shared memory, and GHC-GUM supports distributed memory machines. Both implementations use different, but related, runtime-environment (RTE) mechanisms. Good performance results can be achieved on shared memory architectures and on networks individually. However, a combination of both, for networks of multi-cores, is lacking.

We present the design and implementation of a new parallel Haskell RTE implementation, GUMSMP, which exploits hierarchical platforms more effectively. It is designed to efficiently combine distributed memory parallelism, using a virtual shared heap over a cluster, with low-overhead shared memory parallelism on the multi-cores. Key design objectives for realising this system are: asymmetric load balance, effective latency hiding, and mostly passive load distribution.

We show that the automatic hierarchical load distribution policies must be carefully tuned to obtain good performance, showing the impact of several policies, including work pre-fetching and favouring inter-node work distribution.We present the initial performance results for this implementation, demonstrating the good scalability of a set of 8 benchmarks on up to 100 cores, and show performance gains of up to 20% compared to GHC-GUM.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Run-time environments; D.3.2 [*Programming languages*]: Applicative (functional) languages; C.1.4 [*Processor architectures*]: Distributed architectures

Keywords Parallel Haskell, Virtual Shared Memory, Distributed Architectures

1. Introduction

Multi- and many-core architectures have become the dominant general purpose hardware. Moreover, current trend in parallel architectures has shifted towards networks of multicores, in which several multicores or NUMA architectures sharing memory are connected via a network. In high-performance computing, a hybrid parallel programming model is frequently used in order to best exploit this architecture. This combination requires multilevel parallel programming in both a shared memory model and a distributed memory model. For example, using directive-based parallelism through OpenMP [6] on a physical shared-memory multicore node, combined with message passing coordination, through MPI [28], to be applied across the cluster of multicores. Thus, this combination achieves multilevel parallelism, combining the ease of programming of a shared memory model and the scalability of a distributed memory model. However, managing two abstractions is a burden for the programmer and increases the cost of porting to a new platform.

In contrast, our new runtime environment (RTE) 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 multilevel hierarchy, delegating control almost entirely to the RTE.

Glasgow Parallel Haskell (GpH) [37] is a widely-used parallel extension of Haskell, a lazy functional language. GPH is developed to facilitate parallel programming by limiting the programmer's work to a few key aspects of high-level coordination primitives supported internally by language implementation. In GPH, parallelism is expressed by two primitives added to the Haskell program, **par** and **pseq**. To provide a high level control on parallelism, evaluation strategies [25, 37] (polymorphic and higher order functions) are used to abstract over these primitives.

There are two different implementations of the same semi-explicit programming model, namely: GHC-SMP [24]; a low-overhead physical shared-memory implementation integrated in GHC, and GHC-GUM [36]; 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. While both implementations support a work-stealing approach, GHC-GUM distributes work in the form of sparks communicated by messages passing, preferring to communicate larger grain computations than GHC-SMP. In contrast, spark pools are shared in GHC-SMP and therefore idle processors directly access these pools to steal sparks.

In this paper, we present the design of GUMSMP, a multilevel parallel Haskell 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 a physical shared memory. GUMSMP is motivated by the work distribution of GHC-SMP, using a shared memory model within a single multicore, and by the work distribution of

GHC-GUM, using a distributed memory model across a hierarchy of multicores.

The main *benefits* of this multi-level design of GUMSMP over current implementations (GHC-GUM, GHC-SMP) are:

- GUMSMP provides a scalable model, which works on large distributed memory architectures.
- GUMSMP efficiently exploits the specifics of distributed and shared memory on different levels of the hierarchy.
- GUMSMP provides a single programming model, which renders programming easier and achieves performance portability.

The main *contributions* of this paper are as follows:

- We present the design of GUMSMP, focusing on the improved, hierarchy-aware load-distribution in Section 4.
- We present performance results over a set of 8 benchmarks for the implementation of GUMSMP in Section 5, demonstrating a speedup of up to 20% over GHC-GUM.
- We investigate two policies for improving the automatic hierarchical load distribution: pre-fetching of work, showing improvements between 28% and 57%, and favouring inter-node work distribution, showing an improvement between 4% and 16%, for two classes of benchmarks (Section 6).

2. Related Work

There is a diversity of languages and implementations for parallel Haskell. At the language level, diversity is based on the different supported abstractions. 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, multicores, etc.

Of the *parallel Haskell extensions*, GpH provides a model of semi-explicit parallelism, where potential parallelism is identified in the code, but all coordination aspects are managed automatically by the RTE. In contrast, the Par Monad [26] uses explicit monadic control of concurrency, providing a programming model for pure deterministic parallel computations. Cloud Haskell [10] is a domain-specific language for distributed memory systems, that emulates Erlang style, explicit message passing communication. HdpH [22] is a High-level Distributed-Memory Parallel Haskell extension, influenced by Cloud Haskell, but providing higher-level coordination and targeting hierarchical architectures. Eden [20] is another semi-explicit parallel Haskell extension, in which processes are explicit but communication is implicit. For a more detailed language comparison see [22].

The *implementations* of Cloud Haskell and HdpH are entirely at the Haskell level and are separate from the GHC RTE; thereby enhancing maintainability and facilitating the development of additional functionality. The GUMSMP, GHC-GUM, GHC-SMP, and Eden implementations are all GHC RTE extensions. The GHC-SMP implementation uses physical shared memory primitives; whereas, the implementations of GUMSMP, GHC-GUM, and Eden use message passing primitives to automatically manage synchronisation and communication between parallel threads. While GHC-GUM and GUMSMP implement a virtual shared heap model, Eden relies on distributed heaps and channel-based, implicit communication. A more detailed comparison of implementations is given in Section 3.

The advantage of using a low level language to implement the runtime environment is improved performance. However, implementation maintenance is challenging and needs to be continuously updated. A different approach is the use of Concurrent Haskell to implement functionality, instead of modifying the GHC runtime system; thereby trading performance with maintainability and ease of development. Examples include CloudHaskell, Par-Monad, HdpH and the light-weight concurrency substrate of GHC [35].

Another parallel Haskell dialect is Data Parallel Haskell [31]. This evolved out of earlier work on Nepal [4], which in itself was heavily influenced by the NESL [3] system. With the focus on data parallel applications, hierarchical networks are less of a concern in its design. In fact, one important aspect of the DPH implementation is flattening transformations. This aims to bring parallelism over nested data structures into a flat format, so that they can be more efficiently exploited by massively parallel hardware. SAC [34] is another functional, data-parallel language. Its syntax is based on C, but it uses a single assignment semantic, and is therefore referentially transparent. It also heavily utilises program transformations to improve the efficiency of the data parallelism generated. It mainly targets numerical applications and achieves excellent speedups on the NAS benchmark suite.

These most closely related languages are all dialects of Haskell. Other parallel functional languages with a shared design or implementation concerns are discussed below. Manticore [11] is a parallel implementation for ML and provides implicitly-threaded parallelism [12] that can be combined with Concurrent ML's [32] explicit synchronisation and coordination on a large scale. In 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. An early parallel Lisp implementation was MulT [17], which introduced the notion of lazy task creation [27]. This concept was also employed in the design of GHC-GUM. It allowed threads to subsume the evaluation of the data, for which potential parallelism was then generated. This concept is crucial for the efficient management of divide-and-conquer parallelism. The work on Lazy Threads [13] explores different methods 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 this spectrum, with a focus on low-overheads.

Several other systems have involved similar design decisions when producing systems for dynamic and adaptive management of parallelism. Filaments [21] was an early system focusing on lightweight 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 parallelism, rather than to restrict them at application level. The object-oriented Charm++ system [16] builds on top of C++ and provides an asynchronous message-driven orchestration, together with an adaptive runtime system. It has been used for numerous, large-scale applications; for example biomolecular simulations from the domain of molecular dynamics.

The current state-of-the-art in high-performance computing is to use different programming models at cluster and at node levels of a hierarchical architecture. A typical model might be MPI message passing at cluster level, together with OpenMP on each node, and [38] reports a performance case study. The complexity associated with managing two different parallel programming abstractions restricts this approach to parallel programming to high-performance computing areas. An early attempt to unify programming models in this setting was the implementation of an early definition of OpenMP on clusters of workstations [14], which were built on the TreadMarks distributed shared memory implementation [1]. More recently, a commercial virtualisation solution was provided by ScaleMP, 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 busi-

nesses rather than computing centres; reflecting the transition of parallel programming towards the mainstream. Using an Infiniband network, performance measurements in [33] 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 has been 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 class of partitioned global address space (PGAS) languages takes a data-centric view. It provides 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. Similar to our design, they manage syncronisation and communication automatically inside the RTE. Prominent examples of PGAS languages are Chapel [5] and X10 [7]. PGAS concepts that have been integrated into mainstream languages in the form of Unified Parallel C (UPC) [9] and Co-Array Fortran [29]. A library-based implementation of asynchronous PGAS, in the form of Global Futures [8] provides implicit synchronisation based on future abstraction.

3. GPH Implementations

This section gives an overview of the design of GHC-GUM and GHC-SMP, with special emphasis on thread management and load distribution.

3.1 The distributed memory GHC-GUM implementation

GHC-GUM (Graph Reduction for a Unified Machine Model) [36] is a portable implementation of an abstract graph reduction machine, based on explicit message passing and implementing a virtual shared heap. It implements the GpH parallel Haskell extension. GHC-GUM is built as an extension to the GHC (Glasgow Haskell Compiler) runtime environment [23]. Parallelism is introduced by the `par` primitive, indicating that the evaluation of an expression is potentially parallel, and exploited by reducing separate sub-graphs in parallel [30].

A key concept in GHC-GUM's design is the virtual shared heap, where the graph representing the program to be evaluated in parallel is stored, and is implemented on top of a distributed memory model. Another key characteristic is the dynamic and adaptive management of both work and data. This enables the RTE to adjust the dynamic behaviour of an application to the hardware characteristics and to the dynamic behaviour of the program.

Based on this design, several components of GHC-GUM can be identified:

1. **Initialisation and Termination:** responsible for controlling start up and termination.
2. **Thread Management:** responsible for deciding when to generate a new thread and how to schedule the threads.
3. **Load Distribution:** responsible for distributing the potential parallelism in the parallel system, so that the idle time of the processing elements is minimised.
4. **Memory Management:** responsible for controlling access to remote data and implementing a virtual shared heap.
5. **Communication:** responsible for transferring data and work between processing elements (PEs).

Shared closures (nodes in the graph structure) can be either normal-form closures, representing data, or thunks, representing work (un-

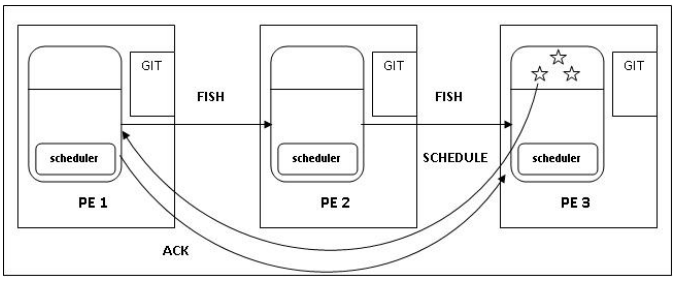

Figure 1: Passive Work Distribution in GHC-GUM

evaluated data). Access to shared closures is implicitly synchronised to avoid two Haskell threads from evaluating the same thunk simultaneously. If input data is under evaluation by another thread, the requesting thread will implicitly block and will be awoken when the evaluating thread updates the closure with the result. In cases where the input data has not yet been evaluated, the requesting thread will evaluate the data itself even if a spark has been created for it. This mechanism is called "thread subsumption", and its behaviour is similar to lazy task creation, which is crucial for throttling the level of parallelism in applications with massive amounts of dynamically created parallelism.

Load Distribution: The load distribution model is designed specifically to achieve an efficient and effective distribution of the available potential parallelism in the form of sparks, without generating an excessive number of messages. Spark generation in GHC-GUM is cheap. It simply consists of adding a pointer to a thunk, which is then added to the spark pool. Cheap "sparking" is essential in order to reduce the parallelism creation overhead, as well as to reduce the communication cost of sending sparks between PE. However, the cost of managing the thread pool is not as low as that for spark pool management. The reason for this is that additional information is required for a thread, such as a live thread priority, which is essential if more flexible scheduling is to be achieved.

Figure 1 presents the (passive) work distribution in GHC-GUM, in a scenario where PE 1 needs to search for work:

Searching for Local Work: In the current version of GHC-GUM, if there are no more threads to run in the thread pool, the scheduler searches for a spark in its spark pool. If a spark is found, it is then activated by turning it into a thread. A thread state object (TSO) is generated in order to hold essential information concerning the thread. If the running thread is blocked for unevaluated data, it will be placed in a queue. When the required data arrives, the blocked thread will be awoken and transferred back to the runnable pool. The data becomes available when it is either reduced by a local thread on the same PE, or its value is sent after being evaluated by another PE.

Searching for Remote Work: If there is no spark in the PE's spark pool (PE 1), the scheduler requests work by sending a FISH message. The FISH message swims randomly from one PE to another, searching for work. It includes the originating PE's id and age number, representing the maximum number of PEs to visit. If the recipient PE has no spark in its spark pool (PE 2), it forwards the message to another PE chosen at random (PE 3), after increasing its age. If the recipient has a spark (PE 3), it sends it to the requesting PE (PE 1) as a SCHEDULE message. If no spark is found, and the message limit is reached, the unsuccessful FISH is then returned to the originating PE. It then waits before sending another FISH message, in order to avoid swamping the

machine with FISH messages in cases where there are only a few busy PEs. For the same reason, each PE only ever has a limited number of outstanding FISH messages (the default number is 1). This mechanism is termed "work stealing", or "passive work distribution", since the work is requested by the idle PE.

3.2 The shared memory GHC-SMP implementation

GHC-SMP is an optimised shared memory implementation of GpH, integrated in GHC [15, 24]. 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 runtime system. In order to achieve such high thread management performance, the threads are multiplexed onto a handful of operating system threads, approximately one for each physical CPU. A thread is represented by a thread state object (TSO), a heap allocated structure holding the Haskell thread's state including its stack. The structure of the TSO is the same as in GHC-GUM.

A small set of operating system threads (worker threads, one worker thread per core) execute the Haskell threads. One Haskell Execution Context (HEC) is maintained for each core, owing to the fact that the worker thread may frequently vary.

The HEC is the data structure where the data required by an OS worker thread in order to execute Haskell threads is contained. Each HEC has a spark-, thread-, and black-hole-queues, which have the same structure as in GHC-GUM. The state required by a HEC to perform ordinary execution of Haskell threads is local to the HEC. This means that a HEC requires no synchronisation, locks, or atomic instructions. Synchronisation is only needed for global cooperation, such as load distribution, garbage collection, etc.

Load Distribution: An HEC's spark pool is implemented as a bounded work-stealing queue, in order to make spark distribution cheap and asynchronous. A work-stealing queue is a lock-free data structure where 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. In order 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 to be pushed is discarded. This means that potential parallelism may be lost.

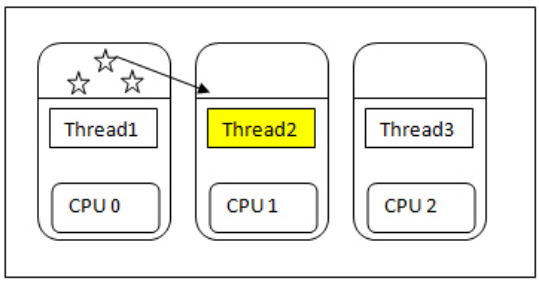

Figure 2: Work Distribution in GHC-SMP

As shown in Figure 2, when an HEC (CPU 1) has no assigned work, it searches for a spark, either in its spark pool or in any other HEC's spark pool. If a spark is found, then the HEC creates a "spark thread" in order to reduce the thread overhead, which in turn steals the spark and starts evaluating it. Once this process has finished, it will steal another spark. Thus, the spark thread will fully evaluate sparks to weak head normal form (WHNF) sequentially until no more sparks are found. At which point it exits, allowing the TSO to be recovered by the GC. Active load distribution in the form of work pushing is also available in GHC-SMP. In this mechanism, if any HEC has sparks and/or runnable threads available, and there are HECs without any work, those sparks and/or threads will be pushed to the pools of the idle HEC's.

4. GUMSMP Design

GUMSMP is designed to be multi-level, using different, tailored technologies on the small-scale, physical shared-memory level (multi-cores) and also on the large-scale, distributed memory level (clusters). We build on the successful technologies that already exist at both levels. In particular, we employ a mechanism of work-stealing for passive load distribution, combined with an adaptive, dynamic mechanism for automatically distributing work and data on a cluster. Technically, we achieve this design by integrating the functionalities of the existing GHC-SMP and GHC-GUM implementations of the RTE for GHC.

The main design objectives for GUMSMP are:

- *Asymmetric load distribution:* While striving for an even load balance, we employ different load distribution policies at inter-node and intra-node level, thus realising an asymmetric load balancing design. At the inter-node level (where communication is expensive) we accept a significant imbalance. On the intra-node level, within a multicore node (where communication is cheap) we aim to optimise for an even load balance, enabling GHC-SMP's mechanism for spark pushing. Compared to GHC-GUM, the load distribution mechanism in GUMSMP is more aggressive, accounting for the availability of several cores at each node in the network.
- *Mostly passive load distribution:* We refine the passive work distribution policy between multicore nodes, where work is only sent remotely when requested (work-stealing), enabling a node to *pre-fetch* work, using a low-watermark mechanism on the spark pool.
- *Gateway routing and distribution*: In our design, one HEC acts as a *gateway* to the rest of the cluster. It is responsible for communication and collects information concerning the remote processor's load. The advantage of this design is that only one processor has to pay the additional cost for maintaining a (partial) picture of the load across the network. The downside of this design is that this processor may then become a bottleneck for higher core numbers.
- *Effective latency hiding:* The system must be able to overlap inter-processor communication with useful computation. Thus, remote data lookup is implemented as a split-phase operation with implicit synchronisation.

Throughout the remainder of this section, we present the GUMSMP design, focusing on improved work distribution policies. In particular, we present the low-watermark mechanism for pre-fetching work and a mechanism of favouring inter-node spark distribution.

4.1 Work Distribution Mechanism

The main objective of the work distribution mechanism is to balance the load between the multicores. Naturally, we are interested in even load balancing to achieve the best utilisation of all computing resources. However, with a combination of multi-cores at the lower level (where several local CPUs 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), we need to use different policies in order to find a trade-off between even load distribution and communication costs. At the cluster level, we use explicit FISH messages (as in

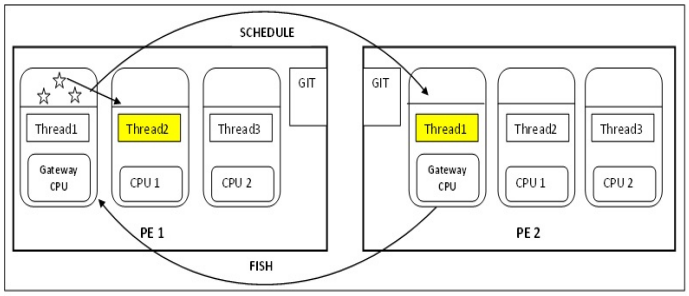

Figure 3: Work Distribution in GUMSMP

```
1  void ScheduleFindWork(Capability *cap , Task
   *task)
2  if emptyRunQueue(cap) then
3      // get local work; GHC-SMP-style
4      if anySpark(cap) then
5          for i ← 0 to num_capabilities do
6              if emptySparkPool(cap[i]) then
7                  continue;
8              end
9              spark = tryStealSpark(cap[i]);
10             if spark != NULL then
11                 break;
12             end
13         end
14         if spark != NULL then
15             tso = createSparkThread(cap,spark);
16             pushOnRunQueue(cap,tso);
17         end
18     else
19         // get remote work; GHC-GUM-style ;
20         pe = choosePE();
21         sendFISH(cap,pe);
22     end
23 end
```

Function ScheduleFindWork(Capability *cap, Task *task) in GUMSMP, combining GHC-SMP and GHC-GUM functionality

GHC-GUM), with a tunable delay to acquire sparks from remote processors. The delay needs to reflect the communication costs on the network, in order to avoid flooding the system with FISH messages, while being able to react sufficiently quickly when becoming idle. Within a multicore, the exchange of work is considerably cheaper and therefore can be undertaken far more aggressively: an idle HEC will directly access the spark pools of other HECs within the same physical shared memory machine, and pick-up work from there, if it has no sparks of its own. Additionally, an HEC with a filled spark pool may actively push work to idle HECs in an effort to more rapidly distribute work among all the HECs. This behaviour is summarised in Figure 3.

In contrast to a flat network of single-cores, an idle multicore represents several unused computation engines. We therefore provide a refinement to this pure model, in the form of *pre-fetching work*, controlled by a low-watermark associated with the spark pool, as shown in Figure 4 and discussed below.

In summary: we use a combination of active and passive load distribution on the intra-node level, and passive load distribution (including work pre-fetch) at the inter-node level. Most 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 of other HECs running on the same PE. Only if no local spark is available will a FISH message be sent to another processor in the system. The concrete work balancing algorithm for GUMSMP is presented in the Function ScheduleFindWork, distinguishing the components related to intra-node (GHC-SMP) and inter-node (GHC-GUM) interaction.

Obtaining a spark: In the current implementation of GUMSMP, when a FISH arrives from another PE, the HEC first searches the spark pool of the main HEC, in order to serve the work-requesting message.

This reflects our design using one dedicated *gateway*in charge of communication, but also identifying work to export.

The advantage of this is that this gateway has the most accurate picture of the current system's information, including the load on different machines. Furthermore, as such 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 available sparks.

We do not make such a finer distinction between sparks in the current implementation. Therefore, we have not yet profited from the advantages of this design. An analysis of our initial performance results in Section 5 will guide us in deciding whether the potential benefits of the current design for the gateway HEC outweighs its overheads.

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 HECs in order to identify the one with the largest spark pool, therefore imposing additional overheads in a common case, something that we wish to avoid.

Watermarks: One simple (but flexible) mechanism that gives better control of spark distribution is to use low- and high-watermarks for each spark pool. Using this approach, work offloading decisions are based on the size of each spark pool, as shown in Figure 4. The *low-watermark* specifies a minimum number of sparks that should be held in the local spark pool. If the number of sparks falls below this watermark, no sparks will be exported, and the PE will attempt to obtain additional sparks from other PEs. This mechanism is designed for high latency systems, with the aim of prefetching 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 a spark pool. If the number of sparks exceeds this limit, the instance will attempt to off-load sparks actively to other instances without receiving work requests.

It will use SCHEDULE messages, in the same manner in which spark pushing is performed between HECs within a PE. That is to say, the PE will temporarily and locally switch from lazy load distribution to eager load distribution, until the spark pool size again drops below the high-watermark. Where all instances have a large number of sparks, a back-off mechanism will be used to introduce a delay between each SCHEDULE message (as described above for FISH messages). While a high-watermark mechanism is implemented in GUMSMP, 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 find good settings, possibly based on previous monitoring of the parallel execution.

Spark placement: Once a stolen spark arrives at a node, the system should decide on a spark pool to place it in. The choice currently taken in GUMSMP is to assign it to the spark pool

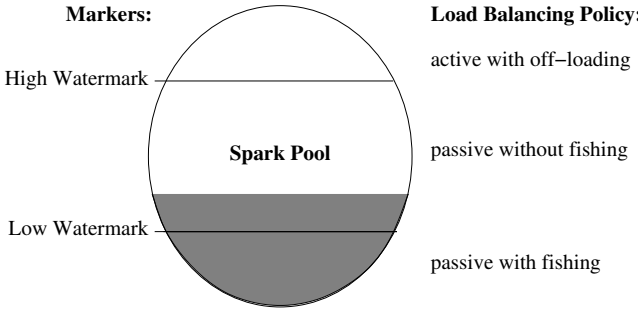

Figure 4: Low- and High-watermark mechanisms for spark distribution.

of the main HEC. Since HECs can exchange work cheaply in their spark pools, this indirection 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 of *heap fragmentation*. This can occur when data, that logically belongs together, is spread over several nodes, mainly due to work-stealing or a fetch request. One RTE parameter that is indicative of high heap fragmentation is the size of the Global Indirection Table (GIT).

A possible means of tackling heap fragmentation would be to use a separate spark pool, dedicated to imported sparks, from which other processors will steal work. This will keep related pieces of work together in one pool, but it does require additional stealing steps, in order to acquire external work. Such an additional spark pool would also be useful in situations in which none of the processors are idle at the time of the arrival of a new spark. This can arise if a thread, that has previously been blocked on remote data, has been awoken and then generated fresh sparks. Placing the imported spark into a dedicated spark pool would defer the placement decision to a later time, when idle processors would be available. Committing too early would not be the best use of the dynamic information of the system.

In the prototype GUMSMP implementation, we use the main HEC of each PE as a gateway, mediating communication and the distribution of work. The gateway can potentially use system information, such as load, to make decisions on work distribution. A pragmatic reason for this initial design is its simplicity, since basic communication operations are not required to be thread-safe. The following section will analyse the performance implications of this design. To improve load balance we primarily use a low-watermark mechanism, as discussed in this section and analysed in the following section.

5. Performance Results

5.1 Experimental Setup

To test the basic functionality and performance of the GUMSMP prototype, we use the following micro-benchmarks that exhibit a range of parallel patterns

- `parfib` is a divide-and-conquer program, which computes for a given `y` the Fibonacci number `fib y` using a depth threshold of `x`;
- `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);
- `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`.
- `parmap-of-parfib` is a data parallel program with nested divide-and-conquer parallelism, computing `x` instances of a parallel `fib 43` computation.
- `worpitzky` is a divide-and-conquer program, which checks the Worpitzky property over Stirling numbers `y z` (parameter `x` specifies the program's version).

Additionally, we measure the performance of the following, larger benchmarks[1]:

- `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 depth `y`;
- `maze` is a nested data-parallel AI application for finding the path through a fixed maze using a parallelism threshold of 5;
- `mandelbrot` is a data-parallel application for computing a mandelbrot set over a given window size (parameters 5 and 6) and number of iterations (parameter 7);

Program Name	Paradigm	Lines of Code	Input Parameters `[x,y,z]`
`parfib`	D&C	18	33 52
`coins`	D&C	29	7 5200 3
`sumEuler`	Data par	28	100000 180
`parmap-of-parfib`	Data par with nested D&C	30	20 43
`worpitzky`	D&C	33	2 27 20
`minimax`	D&C	218	4 9
`maze`	Nested Data par	41	
`mandelbrot`	Data par	106	-2.0 -2.0 2.0 2.0 4096 4096 3024

Our measurements are made on a Beowulf cluster of multicores, where each node is an 8-core CPU (2 quad-core Xeon E5506 2.13GHz, with 256kB L2 and 4MB shared L3 cache). All 32 nodes are connected via a non-specialised Gigabit ethernet connection. All machines are running Linux CentOS 6.4. The implementation of the GHC-SMP RTE is based on GHC 6.12.2, using GCC 4.4.7, and PVM 3.4.5 for message passing.

5.2 Scalability Results

Figure 5 summarises the absolute speedup results from the micro-benchmarks on up to 100 cores of the cluster. Each point in the measurement represents the median of three runtimes. All executions use PVM as a communication library across the network.

All micro-benchmarks scale well up to 100 cores, which significantly exceeds that of a single multi-core machine. Unsurprisingly the simplest micro-benchmark, `parfib`, exhibits a very good overall speedup of 81 on 100 cores, based on a sequential runtime of 6900 seconds. It is only beaten by an improved version of `parmapfib`, which achieves a speedup of almost 88 in this configuration (we will discuss the improvements for this program in Section 6.3). The `coins` benchmark, which is a less regular divide-and-conquer program, achieves a speedup of 68 on 100 cores. For the data-parallel `sumEuler` program, the speedup shows significant variations over an increasing number of cores. This is mostly due to the amount of parallelism being fixed (the number of blocks of data items being processed), which means that for higher core numbers there is a greater risk of load imbalance towards the end of the execution. In general, the RTE is designed to handle parallelism

[1] These are from [25] and the `nofib/parallel` suite, respectively.

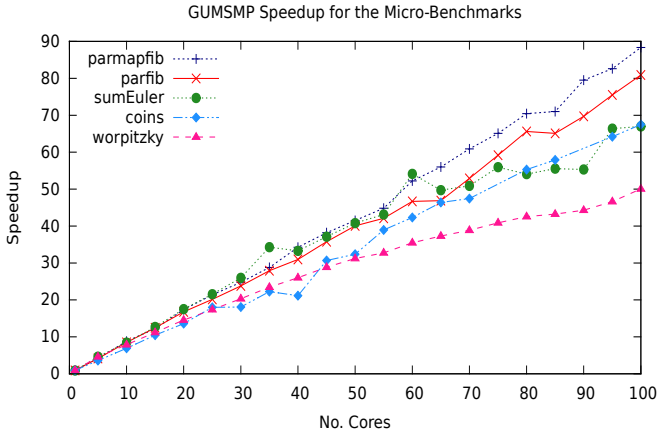

Figure 5: Speedup of *micro-benchmarks* using GUMSMP

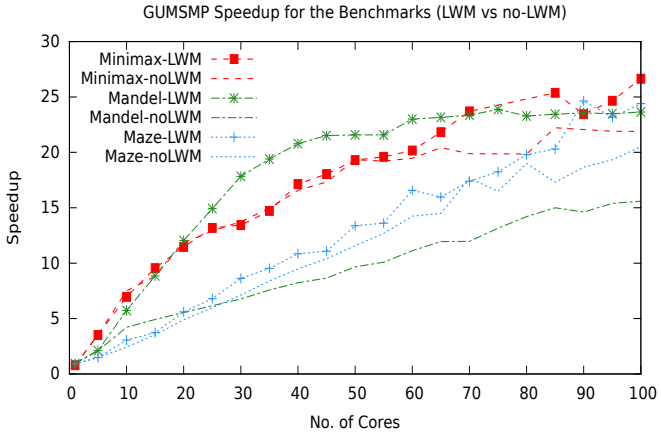

Figure 6: Speedup of *larger benchmarks* using GUMSMP with(out) low-watermarks

dynamically and adaptively, and therefore divide-and-conquer programs that generate a significant amount of parallelism throughout their execution are a better match than data-parallel programs. Furthermore, with `sumEuler` it is crucial to use a low-water-mark mechanism in order to achieve a speedup of 67 on 100 cores, as shown in Figure 11.

The poorest performance results were from the `worpitzky` program. This divide-and-conquer program uses several sources of parallelism, resulting in sparks from both sources being combined in the spark pool for a PE. This aspect hampers the effectiveness of the thread subsumption. A distribution mechanism that attaches a bound on the distance that a spark can travel in a network has been developed in [2] but has not yet been integrated into GUMSMP.

Figure 6 shows the speedups of the larger benchmarks, running with and with-out low-watermarks (this mechanism is analysed in the next section). As anticipated, the speedups for these three benchmarks are lower than for the micro-benchmarks, ranging between 24 for `mandelbrot` and 27 for `minimax`, in their final versions. The larger benchmarks involve significantly greater data transfer, resulting in higher communication overheads: compared to the `sumEuler` micro-benchmark, `mandelbrot` sends 15 times, `minimax` sends 44 times and `maze` sends 125 times the amount of data. Even more importantly, this data transfer also incurs higher virtual shared memory management overheads, due to the data structures being distributed. Evidence of this higher overhead is the average size of the GIT during the execution, which is typically between 10 and 30 times higher than that for the micro-benchmarks. In the case of the larger benchmarks, `minimax` performs best with a speedup of 27 (the speedup increases further to 31 when enabling inter-node sparks — see Section 6.3). With this program, however, the speedup tails off above ca. 35 cores, due to the aforementioned overhead. This is not an inherent limitation of the system, however, as the two other benchmarks show: they still scale, but instead have 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 such GpH benchmarks. The quantity of data exchange needed for these programs is substantially higher, and this the main factor limiting the speedup. The implementations 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 paper.

5.3 Single Multi-core performance

This section compares the performance of GUMSMP, GHC-GUM, and GUMSMP on a single multi-core node of the cluster used for the complete measurements. 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. By comparing the performance with the existing shared memory implementation, GHC-SMP, we can quantify the additional overheads of the GUMSMP design on a single multi-core.

Figure 7 compares the performance for all three systems: GHC-GUM, GUMSMP, and GHC-SMP. We observe that for all the programs GHC-SMP yields the best performance. GUMSMP is typically within 8% of GHC-SMP performance. While GHC-GUM is usually close to the GUMSMP results, in the case of `minimax` its performance is significantly lower. This is due to the overheads associated with the virtual shared heap management, which have to be paid in distributed memory GHC-GUM but not in GUMSMP, which in this setup uses one PE of up to 7 HECs. We observe that on one multi-core GUMSMP, outperforms GHC-GUM in all cases, except `parfib`, where the difference is less than 4%.

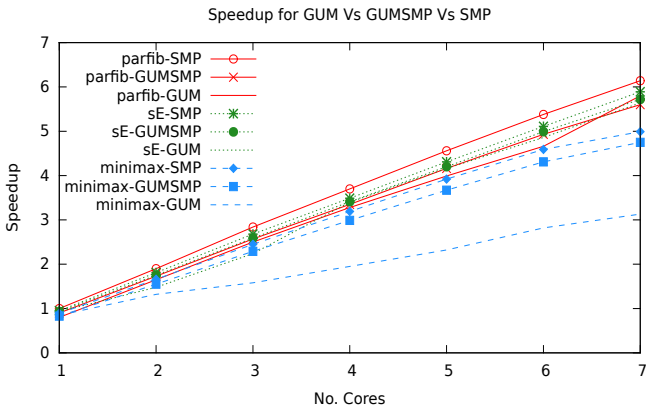

Figure 7: Speedup of all programs using GHC-GUM, GUMSMP, and GHC-SMP on a *single multi-core*

5.4 The Performance of GUMSMP vs GHC-GUM

The older GHC-GUM system can be configured to use a hierarchical network as a flat network; in essence running one instance of the RTE 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.

Figure 8 shows the speedups for all the test programs under GUMSMP and GHC-GUM[2]. Notably, GUMSMP delivers better speedups for all micro-benchmarks with an improvement of up to 20% for `sumEuler`. For the larger benchmarks the behaviour is similar with improvements of around 20% for `minimax` and `mandelbrot`, and a slight slow-down of 4% for `maze`. In general, GUMSMP performs better with data-parallel programs using the refined load-distribution mechanisms, as presented in this paper. In such single-source, data-parallel programs, with only one generator for parallelism, it is beneficial to send off a large computation early, with other capabilities collecting parallelism locally. This structure of parallelism is a natural match for the hierarchically structured RTE. Consequently, we observed a significant reduction in communication, compared to the GHC-GUM instance; the number of messages dropped by up to 41%.

Moreover, for all the benchmarks in Figure 8 we observe an increasing performance gain of GUMSMP over GHC-GUM with larger configurations. This provides evidence of the scalability benefit, due to GUMSMP's hierarchical design.

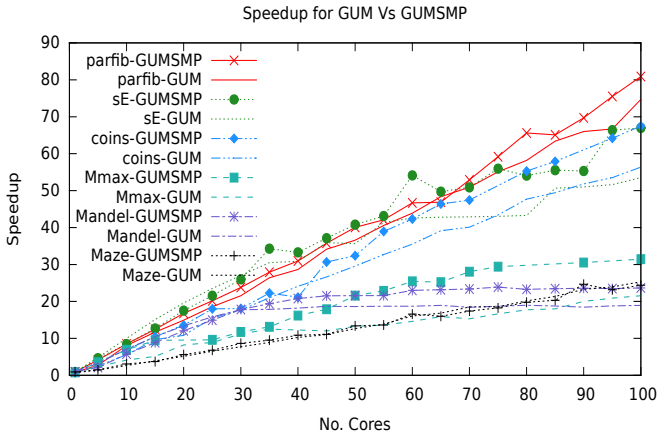

Figure 8: Speedups of benchmarks using GUMSMP *and* GHC-GUM

However, for nested data parallel programs, which generate massive amounts of parallelism, such as `maze`, the speedup of GUMSMP is by 4% smaller for GHC-GUM. Analysing the behaviour of `maze` on GUMSMP on 100 cores we observe an increase of 5% in the number of messages sent in GUMSMP compared to GHC-GUM. More importantly, over 30% of the received sparks were no longer required; typically, because a thread, blocked on remote data, has been awoken and then generated fresh sparks locally. This percentage can serve as an indicator of whether the load balancing policy is overly aggressive when distributing work, resulting in increased communication and higher heap fragmentation. In the concrete case of `maze`, the incurred overhead is still not to impede parallel performance seriously.

[2] Note that no changes to the programs are necessary, since both systems implement GpH

In general, however, the system needs to strike a balance between (pro-)actively distributing work and reducing communication costs in the execution; we discuss this aspect in more detail in the next section. However, the percentage of no-longer-required sparks can serve as an indication to the programmer that load distribution is overly aggressive, and that better results may be achieved by, for example, reducing the low-watermark.

6. Tuning Hierarchical Load Distribution

Load distribution on hierarchical architectures, like clusters of multi-cores, is necessary to account for both multiple cores (PEs) in each cluster node and large differences in latencies. For example, the latencies between cluster nodes are far greater than within the node. This requires adaptation of policies developed in GHC-GUM, and designed for flat clusters. In particular, we look at mechanisms to pre-fetch work to enable the multiple cores on a node to receive work as quickly as possible. Due to the referentially transparent nature of the execution, sharing global data structures is not problematic; all the data necessary for computation is integrated in the transferred graph structure. In addition, the RTE provides a mechanism for specifying how many thunks, i.e. unevaluated closures, should be transferred in a packet. Moreover, since GpH is least prescriptive in the way it defines potentially parallel executions, the RTE has a large degree of flexibility when defining its load distribution policy and, when determining whether to generate parallelism or not.

6.1 A pre-fetching load-distribution policy using a low-watermark mechanism

The low-watermark mechanism was designed to improve load distribution on hierarchical networks over the default load distribution policy; it was initially designed for flat, single-processor networks. In a flat network, passive load distribution, through sending a FISH message when a PE becomes idle (as described in Sec 3.1), works well for distributing work when needed. The danger with sending work (pro-)actively is that there can be a drastic increase in the total amount of communication, due to the unnecessary movement of work away from its input data. However, in a hierarchical system, comprised of multi-core nodes, this default mechanism acquires the amount of work necessary to feed all cores, only very slowly, as shown by the data below.

To visualise this behaviour, Figures 9 and 10 show the per-PE profiles of activities when running the `mandelbrot` program, with and without low-watermarks (note the different x-scales in both graphs). To clarify, dark green[3] is good utilisation, light green is low utilisation, and red is idle time. A per-PE profile shows PEs on the y- 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, amounting to 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 below 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 9, we observe that PE1 has considerably more work (dark green) and higher utilisation compared to the other PEs, which only have sufficient work to keep one HEC busy (light green). This behaviour is confirmed by the average utilisation on PEs 2-16, which ranges between 40% and 49%. Moreover, Figure 9 shows several periods of idle (red) time. The main reason for this behaviour is that `mandelbrot` is data parallel, where the main HEC of PE1 is the only one generating sparks

[3] In the monochrome version, gray is used instead of green, i.e. darker gray represents higher utilisation, and white for red, representing idle times.

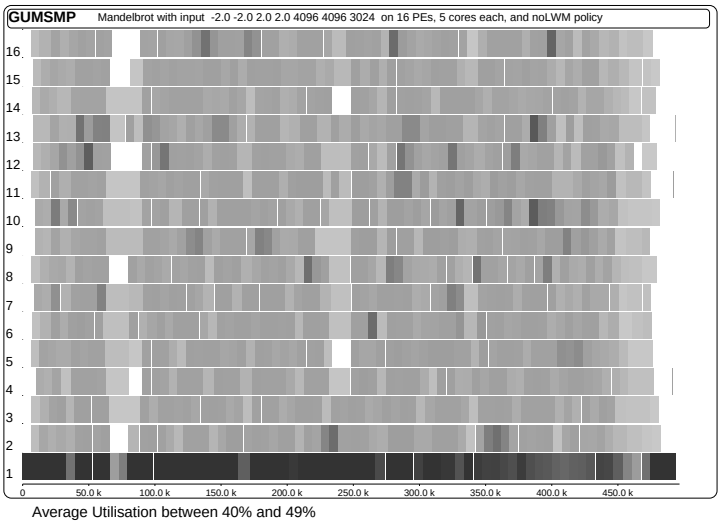

Figure 9: `mandelbrot` load distribution *without low-watermark* on GUMSMP

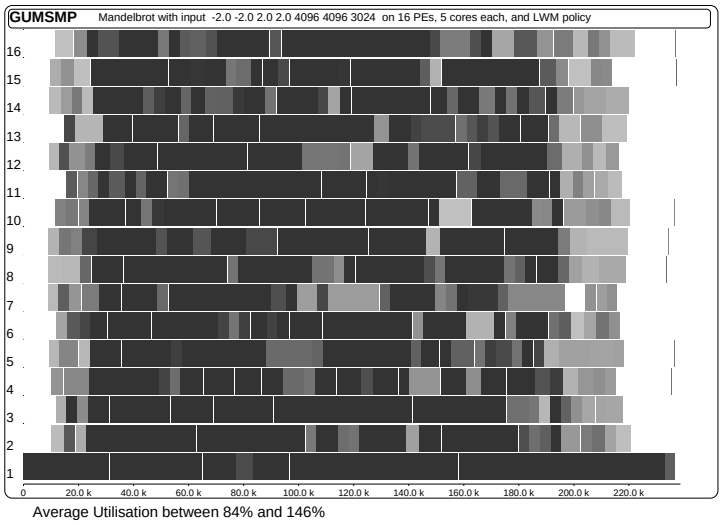

Figure 10: `mandelbrot` load distribution *with low-watermark* on GUMSMP

at the beginning of the execution. Other PEs send FISH messages asking for work from the PE 1. Using a pure work-stealing load distribution policy in Figure 9 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 main HEC, but the other HECs will mostly remain idle, resulting in a delay in 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 over 5 HECs. The utilisation numbers show percentages of mutation time over the total runtime, not including the time spent on garbage collection; therefore, they are not approaching the maximum of 500% for executions running on 5 cores.

In contrast, Figure 10 illustrates the behaviour when enabling low-watermarks; whereby the other PEs continue sending messages requesting work until the number of sparks in all local spark pools reaches the low watermark. Thus, the average utilisation on 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 a significant drop in runtime, from 496s to 238s (a drop by 52%).

In summary, the low-watermark policy enables the pre-fetching of work, so that spark pools reach the low-watermark, which is typically set to the number of HECs available on a single PE, and therefore only depends on a static parameter of the architecture. This results in swifter distribution of the available parallelism throughout the computation, and, in turn, leads to a higher than average utilisation on the other PEs. This is shown by the higher than average utilisation numbers, 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 in this phase the withholding of sparks starves other PEs of work. More desirable than entirely disabling the low-watermark mechanism on the main PE would be to adjust 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, but we plan to integrate such functionality together with more detailed per-thread statistics, to improve the dynamic adaptability of GUMSMP.

In order to assess the impact of the low-watermark mechanism on all programs; Figure 11 compares the speedups for the programs using a low-watermark tailored to the number of cores to improve the load-balance on this hierarchical network (ticked plots) with the speedups in a setting without low-watermarks, using the default passive load distribution mechanism (unticked plots). This comparison shows that the low-watermark mechanism consistently improves performance, by up to 57% in the case of `sumEuler`.

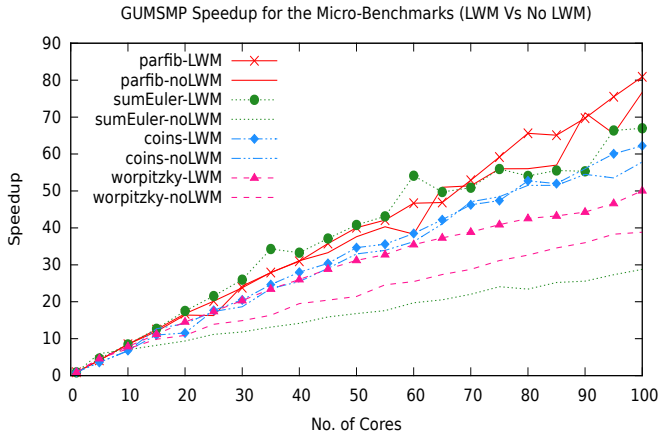

Figure 11: Speedup of GUMSMP *with(out) low-watermarks*

This behaviour is underlined by the results from the larger benchmarks, as shown in Figure 6. All three benchmarks exhibit consistently improved speedup when using low-watermarks, with improvements between 16% and 28% on 100 cores, higher than those for the micro-benchmarks. This reflects, 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 of Figure 10.

6.2 The impact of the FISH delay setting

Another important, tunable parameter for the RTE is the delay between receiving an own, unsuccessful FISH message and sending another FISH message. When running these tests on GUMSMP, we started using the default delay, established with GHC-GUM on flat networks. It transpired that reducing this delay improves performance on a hierarchical network using the GUMSMP RTE. A setting for the FISH delay that strikes 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, is required. Similarly, the low-watermark mechanism is advantageous in GUMSMP in order to send FISHes more quickly, to obtain sufficient parallelism to feed all cores on a multi-core. Both policies reflect the need for a more aggressive load distribution in GUMSMP.

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, even though 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 used small values for the fish delay. These were typically approximately half the value used by default in GHC-GUM.

6.3 Asymmetric load distribution policies

Using the same load distribution policy both on the inter- and intra-node level bears the danger that local HECs may steal large-grained parallelism, which would be more productively executed on another PE in a network.

The `parmapfib` micro-benchmark exhibits this behaviour, as well does the `minimax` benchmark. In both cases, the saturation with parallelism across PEs is poor, with other PEs only picking up small (nested) computations generated by large computations. In order to tackle 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 capabilities 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 other capabilities on the main PE are permitted to pick up work. This refinement of the default load distribution policy accounts for the multi-level structure of the architecture, favouring inter-node spark exchanges initially, to achieve large-scale distribution of large work, and causing significant improvements to the performance of (nested) data parallel programs.

In order to quantify the impact of this policy, Figure 12 shows the speedups for `parmapfib`, `coins` and `minimax` using low-watermarks (LWM), favouring inter-node spark distribution (inter-node sparks) and a combination of both; consistently delivering the best results. For the `parmapfib` micro-benchmark we observe an improvement in performance of up to 16% and for the `minimax` benchmark of up to 4%, all on a 100 core configuration. We expect this policy to be beneficial in general for nested data-parallel programs, where the outer parallelism should be off-loaded to another node. For non-nested data-parallelism, however, it is not necessary to disable the intra-node load distribution, which then risks increased idle time during the start-up phase of the parallel execution. Indeed, we did not observe any improvements for `sumEuler` in excess of the low-watermark mechanism discussed previously.

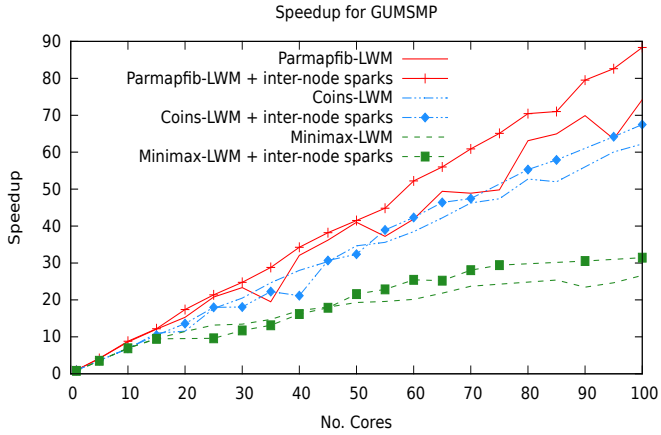

Figure 12: Speedups of GUMSMP using an *asymmetric load distribution policy*, enabling inter-node sparks

6.4 The role of spark placement

Currently an imported spark is added to the spark pool of the gateway HEC. Since the distribution of work between HECs on one multi-core is fairly cheap, it 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-processor pointers, and thus more communication. As yet we have been unable to quantify this aspect of the parallel execution based on these results. We plan to analyse the impact of the spark placement policy in future work, and then revisit the policy for the placement of imported sparks based on this assessment.

A possible design alternative would be to add an additional "import" spark pool to control the placement of sparks. Sparks in this pool might then be further annotated according to their originating PE, or possibly by user-provided information encapsulating information on desirable co-location. In most cases, the scheduler would prefer local sparks, but it can make use of the import spark pool in situations when no local sparks are available. 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.

Related to this separation of sparks into local and imported, is the decision as to whether to duplicate work when sending sparks to other PEs. The current design aims never to duplicate work because of the obvious danger of degrading performance. However, in view of the more aggressive load distribution policy that is necessary for GUMSMP, it might be acceptable to duplicate some work, if such work remains separate from the main pool of sparks and is activated only if other methods of obtaining work fail. The current implementation already provides a means 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. Combining an import spark pool, with a globalisation policy of controlled work duplication, would be an interesting direction for future work.

7. Conclusion

We have presented the design, implementation and early performance results for the new multi-level parallel Haskell implementation GUMSMP, designed for scalable, high-performance computations on networks of multi-cores. Our design focuses on flexi-

ble 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 use a less aggressive policy, which may produce some load imbalance but reduces the total amount of communication. Within a multi-core node we use a more aggressive load distribution policy which exploits the low communications overhead provided by physical shared-memory. Our system mostly uses a refined work-stealing policy, applying the concept of a low-watermark, allowing the system to *pre-fetch work* at the cluster level. This proved to be crucial for the performance of some of the test programs: for the micro-benchmarks runtime drops by up to 57%, for the larger benchmarks by up to 28%. 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 nested data parallel applications: on 100 cores the speedup improves by up to 16%.

More generally, we conclude that large, hierarchical architectures require a more aggressive work distribution policy than flat networks. We achieve such a policy by enabling pre-fetching of work using a *low-watermark* that is tailored to the number of cores per-node, and which yields a performance improvement for all programs, including the more communication intensive benchmarks. We also identified the potential problem of a single multi-core node monopolising all the parallelism, in particular in nested data-parallel programs. Favouring inter-node spark distribution in the start-up phase of such programs improves load distribution and avoids this problem. Thus, this policy should be enabled whenever running nested data-parallel programs.

Initial performance results on five micro-benchmarks and three more communication intensive benchmarks demonstrate the scalability of our multi-level design up to 100 cores, well beyond the size of individual multi-cores. Our implementation enables the execution of GpH programs on networks, extending our previous work on GHC-GUM, and these results represent the first systematic study of GpH performance on the 100 core scale. For the nested data parallel program, the multi-level load-distribution achieves a good match between program structure and hardware topology. The final speedups for the three larger benchmarks are between 24 and 31 on a 100 core machine. This is partly due to an increased amount of communication inherent in the applications, which in turn also increases the overhead for managing the virtual shared heap, but is also partly due to the main HEC becoming a bottleneck in the typically fine-grained, communication between nodes. We plan to address this issue next when tuning the GUMSMP system.

Comparing the performance of the current GUMSMP implementation with the distributed memory GHC-GUM implementation shows a largely positive picture. Only one of the eight benchmarks exhibits a lower performance; `maze` with a slowdown of only 4%. For the remaining programs GUMSMP performance exceeds that of GHC-GUM, with improvements of up to 20%. A direct comparison of the single multi-core performance of GUMSMP with GHC-GUM shows that it is within 8% of the latter's performance, and that it does not introduce a significant additional overhead to the existing shared-memory implementation.

We are continuing to improve GUMSMP, and our current objectives are as follows. We are undertaking performance measurements on a broader class of applications [19]. We plan to drop the restriction that only the main HEC performs communication, in order to eliminate this potential bottleneck. We are currently extending the monitoring support of GUMSMP to provide per-thread statistics. The intention is to use this information to further tune load distribution mechanisms in GUMSMP.

At a system level, we plan to explore some of the alternative design choices discussed in Section 4. Based on the previous work on the performance of the virtual shared memory abstraction [18], we anticipate that a separate import spark pool would reduce heap fragmentation, and thereby improve performance on large clusters.

Acknowledgments

We thank the anonymous referees for their detailed comments that helped to improve the paper. This work has been supported by the European Union grant IST-2011-287510 "RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software", and by the UK's EPSRC grant EP/G055181/1 "HPC-GAP:High Performance Computational Algebra and Discrete Mathematics", and by Saudi Arabian Ministry of Higher Education: Umm Al-Qura University.

References

[1] C. Amza, A. L. 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. doi: 10.1109/2.485843. URL http://dx.doi.org/10.1109/2.485843.

[2] M. Aswad, P. Trinder, and H.-W. Loidl. Architecture Aware Parallel Programming in Glasgow Parallel Haskell (GPH). In *ICCS12: International Conference on Computational Science*, pages 1807–1816, Omaha, Nebraska, June 2012. doi: 10.1016/j.procs.2012.04.199. URL http://doi.acm.org/10.1016/j.procs.2012.04.199.

[3] G. Blelloch, S. Chatterjee, J. C. 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. doi: 10.1006/jpdc.1994.1038. URL http://dx.doi.org/10.1006/jpdc.1994.1038.

[4] M. M. T. 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. doi: 10.1145/357766.351249. URL http://dx.doi.org/10.1145/357766.351249.

[5] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *Intl. J. High Perform. Comput. Appl.*, 21:291–312, Aug. 2007. URL http://portal.acm.org/citation.cfm?id=1286120.1286123.

[6] 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.

[7] 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. ISSN 0362-1340. doi: 10.1145/1103845.1094852. URL http://doi.acm.org/10.1145/1103845.1094852.

[8] D. Chavarria-Miranda, S. Krishnamoorthy, and A. Vishnu. Global Futures: A Multithreaded Execution Model for Global Arrays-based Applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 393–401, 2012. doi: 10.1109/CCGrid.2012.105.

[9] T. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, NY, 2006. URL http://doi.acm.org/10.1145/1188455.1188483.

[10] 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. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034690. URL http://doi.acm.org/10.1145/2034675.2034690.

[11] 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. ISBN 978-1-59593-690-

5. doi: 10.1145/1248648.1248656. URL http://doi.acm.org/10.1145/1248648.1248656.

[12] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20:537–576, 11 2010. ISSN 1469-7653. doi: 10.1017/S0956796810000201. URL http://journals.cambridge.org/article_S0956796810000201.

[13] 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. doi: 10.1006/jpdc.1996.0104. URL http://dx.doi.org/10.1006/jpdc.1996.0104.

[14] Y. Hu, H. Lu, A. Cox, and W. Zwaenepoel. OpenMP for Networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000. doi: 10.1.1.136.2705. URL http://dx.doi.org/10.1.1.136.2705.

[15] D. Jones, Jr., S. Marlow, and S. Singh. Parallel Performance Tuning for Haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: http://doi.acm.org/10.1145/1596638.1596649. URL http://doi.acm.org/10.1145/1596638.1596649.

[16] L. V. 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. ISBN 0-89791-587-9. doi: 10.1145/165854.165874. URL http://doi.acm.org/10.1145/165854.165874.

[17] 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. doi: 10.1145/74818.74825. URL http://dx.doi.org/10.1145/74818.74825.

[18] H.-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *CCGrid/DSM 2002 — Intl. Symp. on Cluster Computing and the Grid*, pages 311–318, Berlin, Germany, May 21–24, 2002. IEEE Press. URL http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/dsm02.ps.gz.

[19] H.-W. Loidl, P. Trinder, K. Hammond, S. Junaidu, R. Morgan, and S. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency and Computation: Practice and Experience*, 11:701–752, 1999. doi: 10.1002/(SICI)1096-9128(199910)11:12⟨701::AID-CPE443⟩3.0.CO;2-P. URL http://www.macs.hw.ac.uk/~dsg/gph/papers/ps/cpe.ps.gz.

[20] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel Functional Programming in Eden. *J. Funct. Program.*, 15:431–475, May 2005. ISSN 0956-7968. doi: 10.1017/S0956796805005526. URL http://portal.acm.org/citation.cfm?id=1067405.1067409.

[21] D. K. Lowenthal and V. W. F. G. R. Andrews. Using Fine-grain Threads and Run-time Decision Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1), 1996. doi: 10.1006/jpdc.1996.0106. URL http://dx.doi.org/10.1006/jpdc.1996.0106. Special issue on multithreading for multiprocessors.

[22] P. Maier and P. Trinder. Implementing a High-Level Distributed-Memory Parallel Haskell in Haskell. In A. Gill and J. Hage, editors, *IFL'12: Implementation and Application of Functional Languages*, LNCS 7257, pages 35–50. Springer Berlin Heidelberg, 2012. doi: 10.1007/978-3-642-34407-7_3. URL http://dx.doi.org/10.1007/978-3-642-34407-7_3.

[23] S. Marlow and S. Peyton Jones. *The Architecture of Open Source Applications, Vol 2*, chapter The Glasgow Haskell Compiler. lulu.com, 2012. URL http://www.aosabook.org/en/ghc.html.

[24] 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. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596563. URL http://doi.acm.org/10.1145/1596550.1596563.

[25] S. Marlow, P. Maier, H.-W. Loidl, M. K. 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. ISBN 978-1-4503-0252-4. doi: http://doi.acm.org/10.1145/1863523.1863535. URL http://doi.acm.org/10.1145/1863523.1863535.

[26] 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. ISBN 978-1-4503-0860-1. doi: 10.1145/2034675.2034685. URL http://doi.acm.org/10.1145/2034675.2034685.

[27] E. Mohr, D. Kranz, and J. Halstead, R.H. Lazy task creation: a technique for increasing the granularity of parallel programs. *Parallel and Distributed Systems, IEEE Transactions on*, 2(3):264–280, July 1991. doi: 10.1109/71.86103. URL http://dx.doi.org/10.1109/71.86103.

[28] MPI Forum. MPI 2: Extensions to the Message-Passing Interface. Technical report, University of Tennessee, Knoxville, 1997.

[29] R. W. Numrich and J. Reid. Co-array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998. ISSN 1061-7264. doi: 10.1145/289918.289920. URL http://doi.acm.org/10.1145/289918.289920.

[30] S. L. Peyton Jones. Parallel Implementations of Functional Programming Languages. *Comput. J.*, 32:175–186, April 1989. ISSN 0010-4620. doi: 10.1093/comjnl/32.2.175. URL http://portal.acm.org/citation.cfm?id=63410.63418.

[31] S. L. Peyton Jones, R. Leschinsky, G. Keller, and M. M. T. 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. doi: 10.1007/978-3-540-89330-1_10. URL http://dx.doi.org/10.1007/978-3-540-89330-1_10.

[32] 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. doi: 10.1007/3-540-56883-2_10. URL http://dx.doi.org/10.1007/3-540-56883-2_10.

[33] 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. ISBN 978-0-7695-4220-1. doi: 10.1109/CLUSTER.2010.38. URL http://dx.doi.org/10.1109/CLUSTER.2010.38.

[34] 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. doi: 10.1017/S0956796802004458. URL http://dx.doi.org/10.1017/S0956796802004458.

[35] K. Sivaramakrishnan, T. Harris, S. Marlow, and S. Peyton Jones. Composable Scheduler Activations for Haskell. Technical report, July 2013. URL http://research.microsoft.com/en-us/um/people/simonpj/papers/lw-conc/lwc-hs13.pdf.

[36] 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. doi: 10.1145/231379.231392. URL http://dx.doi.org/10.1145/231379.231392.

[37] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998. doi: 10.1017/S0956796897002967. URL http://dx.doi.org/10.1017/S0956796897002967.

[38] 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. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.3035&rep=rep1&type=pdf.