The Design and Implementation of GUMSMP:

a Multilevel Parallel Haskell Implementation

Malak Aljabri
Heriot-Watt University
ma767@hw.ac.uk

Hans-Wolfgang Loidl
Heriot-Watt University
H.W.Loidl@hw.ac.uk

Phil W. Trinder
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 several parallel Haskell implementations targeting different architectures. In particular, GHC-SMP supports shared memory, and GHC-GUM supports distributed memory. Both implementations use different but related runtime-environment (RTE) mechanisms. Good performance results can be achieved on shared memory architectures and on networks individually, but 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 to better exploit such hierarchical platforms. 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 in realising this system are asymmetric load balance, effective latency hiding, and mostly passive load distribution. We present initial performance results for this implementation, indicating that the new RTE outperforms the distributed memory implementation in certain scenarios.

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 architectures. Moreover, the current trend in parallel architectures has shifted towards networks of multicores, in which several multicore CPUs with nodes physically sharing memory are connected via a network. In high-performance computing a hybrid parallel programming model is often used to best exploit this architecture. Such a hybrid model uses multiple coordination abstractions, e.g. MPI + OpenMP, where OpenMP (directive-based parallelism) is applied within a multicore node and MPI (message passing interface) is applied across the cluster of multicores. Thus, this achieves a 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 on both levels of the hierarchy. Therefore, this model relieves the programmer from the burden of explicitly controlling coordination on a multi-level hierarchy, delegating such control completely to the RTE.

Glasgow Parallel Haskell (GpH) [22] 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 the language implementation. In GPH parallelism is expressed by two primitives added to the Haskell program, **par** and **pseq**. Evaluation strategies [16, 23] are polymorphic and higher order functions abstract over these primitives to provide a high level control of parallelism.

There are two different implementations of this semi-explicit programming model, namely GHC-SMP [15], a low-overhead physical shared-memory implementation integrated in GHC, GHC-SMP, and a virtual shared memory implementation on clusters built on top of explicit message passing, GUM [21]. A major difference between them lies in the work distribution model supported. While both implementations support a work-stealing approach, GUM distributes work in the form of sparks that are communicated by message passing and prefer coarse grain computations to be sent away. In contrast, spark pools are shared in GHC-SMP and therefore idle processors steal sparks from the spark pools of the busy ones.

In this paper we present the design of GUMSMP, a multilevel parallel Haskell implementation which integrates the advantages of these two implementations, thus providing a platform for scalable parallelism that is not bounded by the limitations of a physical shared memory. GUMSMP is motivated by the work distribution of the GHC-SMP as the shared memory model within a single multicore and by the work distribution of the GUM as the distributed memory model across a hierarchy of multicores.

The main *benefits* of this multi-level design of GUMSMP over the existing implementations (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 makes programming easier and achieves performance portability.

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

- We provide a detailed description of parallel Haskell languages and implementations in Section 2.
- We present the design of GUMSMP, focusing on the improved, hierarchy-aware scheduling and placement of light-weight threads in Section 4.
- We present preliminary performance results for GUMSMP in Section 5.

2. Related Work

There is a diversity of languages and implementations of parallel Haskell. At the language level, the diversity is based on the different abstractions supported which vary in how explicitly they control parallelism, e.g. implicit, semi-explicit, and fully explicit approaches. At the implementation level, the diversity is based on different classes of architectures with different characteristics, e.g. clusters, multicores, etc. In this section we briefly outline different parallel Haskell languages and implementations.

2.1 Parallel Haskell Languages

Some important parallel Haskell languages are classified according to the abstraction level supported as follows:

Explicit parallelism:

The Par Monad [17] is a new parallel Haskell programming model for pure deterministic parallel computations providing monadic control of concurrency.

CloudHaskell [6] is a domain-specific language for developing programs for distributed memory systems. It emulates Erlang style message passing communication yet still benefits from Haskell features such as purity, types, and monads. Cloud Haskell has been re-implemented recently to support multiple transport implementations in order to provide high performance and scalability. The new implementation provides more precise semantics in terms of message communications as well as node connection.

Semi-explicit parallelism:

Eden [13] is a semi-explicit approach to functional parallel programming which extends Haskell with constructs to support parallelism. Processes are defined explicitly in Eden, but the communications are implicit, thus achieving a high level of abstraction. Eden supports distributed memory parallelism with message passing as a communication model. The programmer has some control over the load balancing as well as the granularity in order to specify expressions that have to be evaluated as parallel processes. Eden provides high level parallelism abstractions (libraries of skeletons) and therefore simplifies the task of parallelizing a program substantially.

HdpH [14] a High-level Distributed-Memory Parallel Haskell is heavily influenced by the design and implementation of Cloud Haskell, targeting distributed memory architectures with multicore nodes. It supports high-level semi-explicit parallelism, dynamic load management, polymorphism, powerful coordination abstractions, and has the potential for fault-tolerance.

2.2 Parallel Haskell Implementations

Parallel Haskell implementations can be classified as distributed memory or shared memory implementations as follows:

Distributed Memory Implementation:

GUM [21] is the distributed memory implementation for GpH which is discussed in further detail in Section 3.1.

Dream/EDI [13] is the distributed memory Eden implementation, which extends GHC functionality by defining primitives for explicit remote task creation and channel based communication mechanisms and supports the eager work distribution model.

CloudHaskell [6] is implemented entirely in Haskell as processes with explicit message passing and function closures serialisation. The overhead associated from using Haskell as a system language is acceptable as demonstrated by the initial performance results.

HdpH [14] is implemented entirely in Concurrent Haskell. 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, etc., thus it preserves maintainability and facilitates development.

Shared Memory Implementations:

GHC-SMP [15] is the shared memory GpH implementations which is discussed in more detail in Section 3.2.

Par Monad [17] provides implementation of the system-level functionality (work-stealing scheduler) as a Haskell library. The Par-Monad associated overhead is still low, as indicated by the performance results.

Meta-Par [8] built on the Par Monad, takes Haskell-level programmability and abstractions of runtime-system functionality further, especially for scheduling on heterogeneous architectures (with a focus on CPU/GPU interaction). A work-stealing scheduler is described which reifies core data-structures like the thread pool in Haskell.

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

Having a runtime system implemented in low level language resulted in a high application performance. However, the implementation maintenance is challenging and needs to be continuously updated.

One current trend for parallel Haskell implementations is to use Concurrent Haskell to implement all 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 [20].

Table 1 has been reproduced from [14] which compares the key features of different parallel Haskells. Shared memory implementations have limited scalability as they only work on a multicore. On the other hand, distributed memory implementations work on shared memory architectures as well as on distributed memory architectures. They can still give good performance on multicores as long as the tasks to be communicated are large and the communication rate is low [1, 2]. GUMSMP is also scalable, but it provides improvements for two aspects of the parallel implementations, namely implicit task placement and automatic load balancing. This resulted from integrating GHC-SMP which is tuned for multicore architectures and GUM which is tuned for clusters. Thus, GUMSMP is designed to provide an architecture-aware system tuned for a cluster of multicore architectures. As a result, if the computation is small it will remain in the multicore, but if it is large, it can be sent to different nodes in the clusters, thus reducing the communication overheads associated with GUM.

Table 1: Parallel Haskell Comparison

	Distributed Memory				Shared Memory		
Property / Language	GUMSMP	GUM	Eden	CloudHaskell	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	++	+	+	-	+	+	+

All previously presented languages are dialects of Haskell. There are many other parallel functional languages. An example is Manticore [7], which is a heterogeneous, statically typed, strict language with support of multiple levels of parallelism. It combines three components: the sequential functional language is drawn from SML, explicit mechanisms for concurrency are drawn from CML [19] and implicit mechanisms for parallelism are drawn from NESL [4] and Nepal [5].

3. GPH Implementations

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

3.1 The distributed memory GUM implementation

GUM (Graph Reduction for a Unified Machine Model) [3, 21] is the portable, message basing virtual machine for the parallel Haskell Functional Language which has been released as an extension to the GHC (Glasgow Haskell Compiler) [10]. It is based on the parallel reduction of the graph representing the program, and the parallelism being exploited by the reduction of independent sub-graphs being carried out in parallel [18].

The key concepts in GUM's design are to support a virtual shared heap where the graph representing the program to be evaluated in parallel is stored and which is implemented on top of a distributed memory model as well as dynamically managing resources for work and data.

Based on its design, several components of GUM can be identified:

1. **Initialization 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 balancing:** responsible for distributing the load in the parallel system so that the processing elements' idle time is minimized.
4. **Memory Management:** responsible for controlling access to remote data and in GUM, implementing a virtual shared heap.
5. **Communication:** responsible for transferring data and work between PEs.

Shared closures (nodes in the graph structure) can be either normal-form closures, representing data, or thunks, representing work (unevaluated data). Access to shared closures is implicitly synchronised to avoid two Haskell threads from evaluating the same thunk simultaneously.

Load Balancing: The load balancing model is designed specifically to achieve an efficient and effective distribution of the available sparks without generating an excessive number of messages. Spark generation in GUM is cheap. It is simply the adding of a pointer to a thunk which is then added to the spark pool. This is essential 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 needed for a thread such as a live thread priority, which is essential if more flexible scheduling is to be achieved.

Figure 1 presents the work distribution in GUM which is explained as follows:

Searching for Local Work: In the current version of 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 activated by turning it into a thread and generates a TSO to hold essential information about the thread and starts evaluating it. If the running thread is blocked for unevaluated values, it will be put in a queue and when the required data arrives the blocked thread will be awakened and transferred back to the runnable pool. The data becomes available when it is either reduced by a local thread in 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, 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, it forwards the message to another PE chosen at random after increasing its age. If the recipient has a spark, it sends it to the requesting PE 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, which then waits before sending another FISH message in order to avoid swamping the machine with FISH messages when 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 called "work stealing", or passive work distribution, since the work is requested by the idle PE. Algorithm ScheduleFindWork presents the load balancing mechanism implemented in GUM.

3.2 The shared memory GHC-SMP implementation

GHC-SMP is an optimized shared memory implementation for functional parallel Haskell integrated in GHC [9, 15]. It assumes a physical shared memory and uses mutexes for synchronization between local threads. GHC-SMP excels at the efficient handling of lightweight threads. Millions of lightweight threads are supported by the GHC runtime system. To achieve this the threads are multiplexed onto a handful of operating system threads, approximately one for each physical CPU. A (TSO) thread state object is a heap

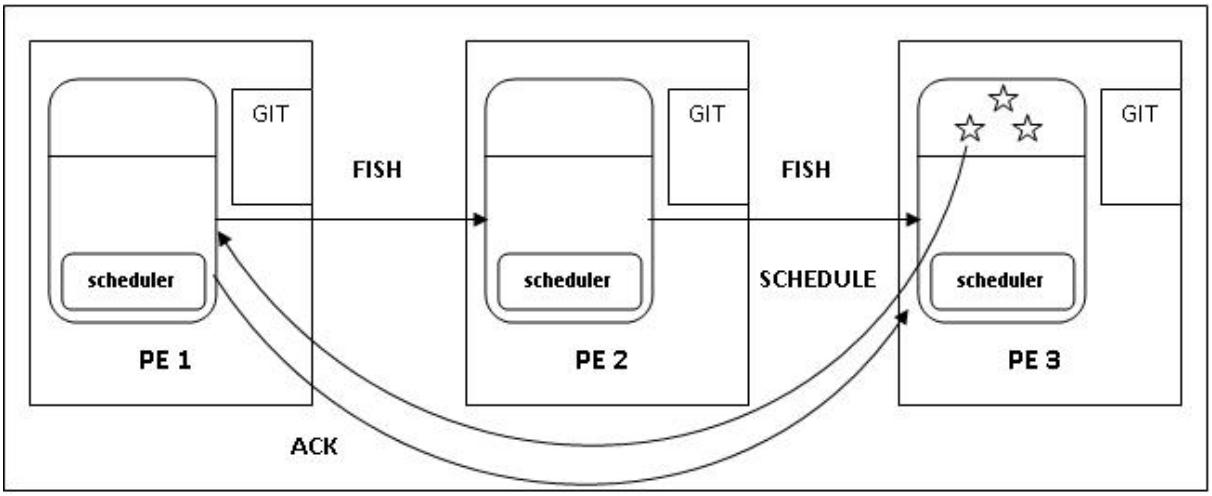

Figure 1: Work Distribution in GUM

```
1  Void ScheduleFindWork(Capability *cap , Task
   *task)
2  if emptyRunQueue(cap) then
3  |   //Call ScheduleActivateSpark(cap)to get
   |  local work
4  |  if anySpark(cap) then
5  |  |  spark = tryStealSpark(cap);
6  |  |  if spark != NULL then
7  |  |  |  tso = createSparkThread(cap,spark);
8  |  |  |  pushOnRunQueue(cap,tso);
9  |  |  end
10 |  else
11 |  |  //Call Function ScheduleGetRemoteWork(cap)to get
   |  |  remote work
12 |  |  pe = choosePE();
13 |  |  sendFISH(cap,pe);
14 |  end
15 end
```

Function ScheduleFindWork(Capability *cap, Task *task)in GUM

allocated structure used to keep the Haskell thread's state together with its stack where it runs (same TSO structure as in GUM).

A set of operating system threads (worker threads, one worker thread per CPU) execute the Haskell threads. One Haskell Execution Context (HEC) is maintained for each CPU 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, threads, and global black hole queues that are the same as those for 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 some situations such as load balancing, garbage collection, etc.

Load Balancing HEC's spark pool is implemented as a bounded work-stealing queue in order to make spark distribution cheaper and more 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 synchronization. Other threads can steal from the other end of the queue, meaning that only one atomic instruction is needed. 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, meaning potential parallelism may be lost.

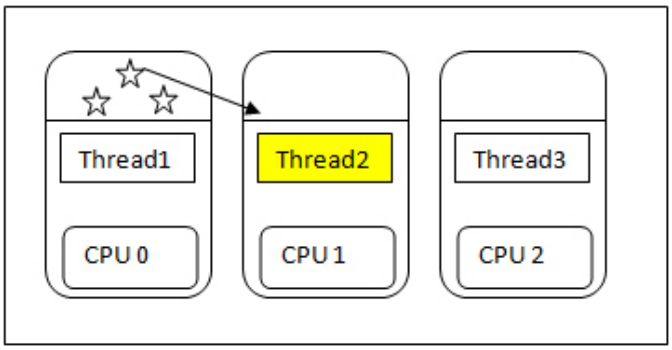

Figure 2: Work Distribution in GHC-SMP

As shown in Figure 2, when an HEC 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 evaluate sparks to WHNF sequentially until no more sparks are found, at which point it exits, allowing the TSO to be recovered by the GC.

It is necessary to create a spark thread in order to avoid creating a new thread and a fresh TSO for every spark and to discard it after completing the evaluation for recovery by the garbage collector. In this way there will only be one thread executing multiple sparks. This also fixes the problem of latency between creating the parallel tasks and being able to execute them in another CPU. The algorithm ScheduleFindWork presents the load balancing mechanism implemented in GHC-SMP.

```
1  Void ScheduleFindWork(Capability *cap , Task
   *task)
2  if emptyRunQueue(cap) then
3  |   //Call ScheduleActivateSpark(cap) to get
   |  local work;
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_shell);
17 |  |  end
18 |  end
19 end
```

Function ScheduleFindWork(Capability *cap, Task *task)in GHC-SMP

3.3 Main Scheduling Loop

For both implementations, the core of each PE's execution is the scheduling loop presented in Algorithm 1 which is executed by each PE. The main difference between GUM and GHC-SMP is in the load balancing mechanism presented in the function **ScheduleFindWork** for both.

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 on the large-scale, distributed memory level (clusters). We build on the successful technologies that exist for both levels already, 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 GUM implementations of the RTE for GHC.

The main design objectives for GUMSMP can be summarized as follows:

- *Asymmetric load balancing:* While striving for even load balance, we employ different concepts for the different levels, thus realising an asymmetric design of load balancing. On the large scale, where communication is expensive, we accept significant imbalance, whereas within a multicore node, where communication is cheap, we aim to optimise for an even load balance.
- *Mostly passive load distribution:* It is essential to maintain a passive work distribution between multicore nodes, so work is only sent remotely when requested (work-stealing). On the other hand, within a multicore, it is preferred to maintain active work distribution as the communication is carried out locally within the same multicore.
- *Gateway routing and distribution*: In our design, one processor at the lower level acts as a gateway to the rest of the cluster. It is in charge of communication and collects information about the load of remote processors. The advantage of this design is that only one processor has to pay the extra cost for maintaining a (partial) picture of the load across the network. The downside in this design is that this processor may become a bottleneck for higher core numbers.
- *Effective latency hiding:* the system must be designed in such a way that communication costs are not on the critical path of cooperative computations. Conceptually this is achieved by implementing the access to some data that resides on another processor as a split-phase operation with implicit synchronisation. While the data is being fetched, another thread can be executed. To be effective, this mechanism relies on a large pool of runnable threads, to overlap communication and computation.

```
1  while True do
2  |  switch sched_ state do
3  |  |  case SCHED_ RUNNING
4  |  |  |  continue;
5  |  |  case SCHED_ INTERRUPTING
6  |  |  |  performGC ;
7  |  |  |  shut_ down;
8  |  |  case SCHED_ SHUTTING_ DOWN
9  |  |  |  Exit;
10 |  endsw
11 |  ScheduleCheckBlackHole(cap);
12 |  ScheduleSendPendingMessages(); //Send any
   |  messages
13 |  ScheduleFindWork(cap);
14 |  processMessages(cap);
15 |  ScheduleYield(cap);
16 |  if emptyRunQueue(cap) then
17 |  |  continue;
18 |  end
19 |  tso = popRunQueue(cap);
20 |  result = stgRun(tso);
21 |  switch result do
22 |  |  case out_of_heap
23 |  |  |  pushOnRunQueue(cap,tso); performGC;
24 |  |  case out_of_stack
25 |  |  |  enlargeStack(tso); pushOnRunQueue(cap,tso);
26 |  |  case time_expired
27 |  |  |  pushOnRunQueue(cap,tso);
28 |  |  case finished
29 |  |  |  if bound then
30 |  |  |  |  return
31 |  |  |  else
32 |  |  |  |  continue;
33 |  |  |  end
34 |  endsw
35 end
```

Algorithm 1: Main Scheduling Loop for GUM and GHC_SMP

In the remainder of this section we present the GUMSMP design, focusing on the work distribution algorithm.

4.1 Work Distribution Mechanism

The main objective of the work distribution mechanism is to balance the load between the multicores, and of course we are interested in an even load balance to make best use of all the 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 between nodes at the higher level, which makes the transfer of work and data expensive, we need to use different policies in order to balance overhead with load distribution. At the cluster level, we use explicit FISH messages (as in GUM), with a tunable delay between them, in order to acquire sparks from remote processors. Choosing a suitable delay is important to avoid flooding the system with FISH messages, while being able to react sufficiently quickly to becoming idle. Within a multicore the exchange of work is much cheaper and can therefore be done much more aggressively: an idle HEC will directly access the spark pools of other HECs within the same physical shared memory machine, and pick-up the work from there, if it has no sparks of its own. This behaviour is summarised in Figure 3.

An alternative to this work-stealing-based load balancing policy would be to more aggressively send away work, which in turn would increase the total amount of communication and, even more severely, increase the heap fragmentation in the virtual shared heap, resulting in a considerable increase in runtime overhead. In general this is undesirable. However, in high-latency clusters, the delay between sending a FISH and receiving work, using a pure work-stealing model, might be considerable. We therefore provide a refinement to this pure model in the form of a low-watermark.

Most notably, the GUMSMP design for load distribution is hierarchy-aware. In looking for work, each HEC prefers local sparks, from its own spark-pool, directly steals sparks from the pools of other HECs running on the same PE, and only if no local spark is available it will send a FISH message to another processor in the system. The concrete work balancing algorithm for GUMSMP is presented in the Function ScheduleFindWork.

```
1  Void ScheduleFindWork(Capability *cap , Task
   *task)
2  if emptyRunQueue(cap) then
3  |   //Call ScheduleActivateSpark(cap)to get
   |  local work
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,spark);
17 |  |  end
18 |  else
19 |  |   //Call Function
   |  |  ScheduleGetRemoteWork(cap)remote work ;
20 |  |  pe = choosePE();
21 |  |  sendFISH(cap,pe);
22 |  end
23 end
```

Function ScheduleFindWork(Capability *cap, Task *task) in GUMSMP

Obtaining a spark: In the current implementation of GUMSMP, when a FISH arrives from another PE, the HEC will first search the spark pool of the main HEC in order to serve the work-requesting message. This reflects our design of using one dedicated gateway to other processors in charge of communication but also of identifying work to export. The advantage is that this gateway has the most accurate picture of the current system 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 the available sparks.

In the current implementation, however, we don't make such a finer distinction between sparks and therefore don't profit 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 of gateway HEC outweigh its overhead.

Another 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 find out the one with the largest spark pool and therefore impose additional overhead.

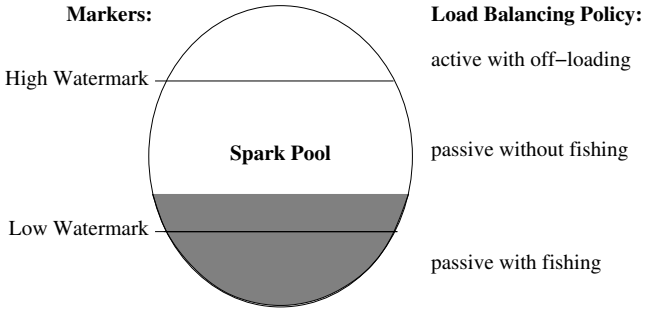

Figure 4: Low- and High-watermark mechanisms for spark distribution.

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 sizes 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 ever be exported, and the instance will try to obtain additional sparks from other instances. This mechanism is designed for high latency systems, aiming to pre-emptively acquire work and thus support effective latency hiding, 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 actively off-load sparks to other instances without being asked for work, using SCHEDULE messages. In other words, the instance will temporarily and locally switch from lazy load distribution to eager load distribution, until the spark pool size drops below the high-watermark again. Where all instances have large numbers of sparks, a back-off mechanism is used to introduce a delay between each SCHEDULE message, as described above for FISH messages. This high-watermark mechanism is currently not used in GUMSMP, but we plan to use it on programs that have aggressive generator threads that create a lot of sparks in a short time period.

Spark placement: Once a stolen spark arrives at a node, the system has to decide in which spark pool to put it. The choice currently taken in GUMSMP is to assign it to the spark pool of the main HEC. Since HECs can cheaply exchange work in their spark pools, this indirection of retrieving work should not

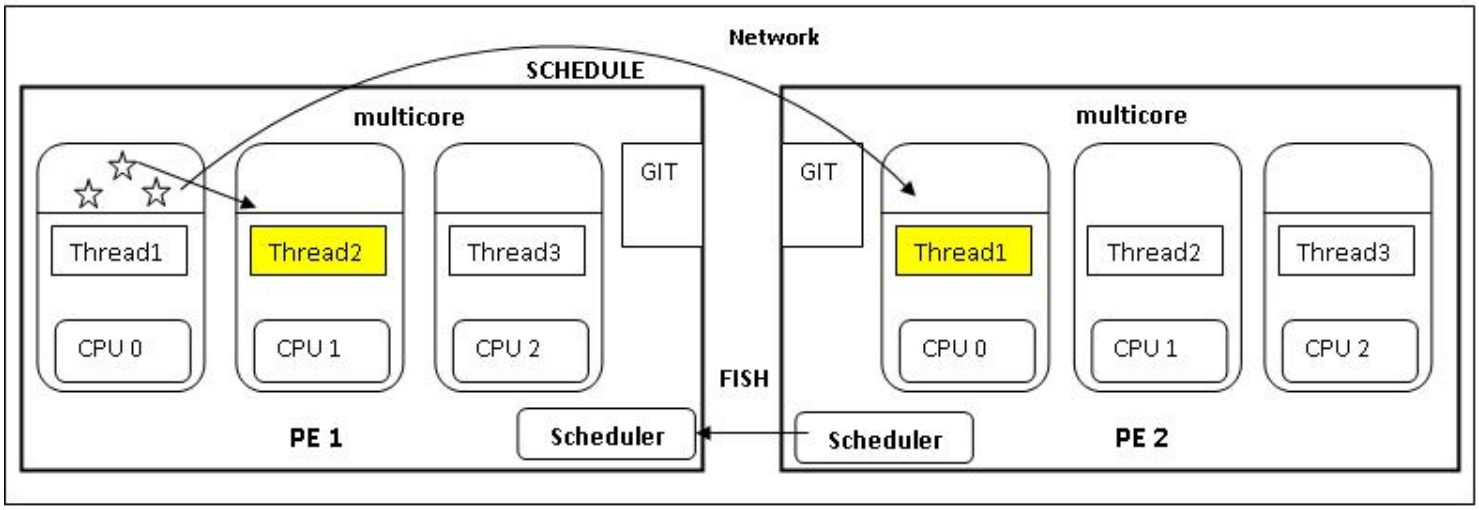

Figure 3: Work Distribution in GUMSMP

incur any significant delay. However, one general problem with work distribution in a virtual shared heap model is the danger of *heap fragmentation*. This occurs when data, logically belonging together, is spread over several nodes, mainly due to the work-stealing or due to a fetch request. One RTE parameter that is indicative of high heap fragmentation is the size of the GIT table.

One possibility to tackle heap fragmentation would be to use a separate spark pool, dedicated to imported sparks, from which other processors will steal work. This keeps related pieces of work together in one pool, but requires additional stealing steps in order to acquire external work. Such an additional spark pool would also be useful in situations where none of the processors are idle at the time of the arrival of a new spark (the processor originally requesting work, might have found new work locally in the meantime). Putting the imported spark into a dedicated spark pool would defer the placement decision to a later point, when idle processors are available. Committing too early would not make 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 distribution of work. The gateway can potentially use system information, such as load, to make decisions on work distribution. One pragmatic reason for this initial design is its simplicity, since basic communication operations don't have 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. Preliminary Performance Results and Implementation Issues

5.1 Experimental Setup

To test the basic functionality and performance of the GUMSMP prototype, we use the several micro-becnhmarks that exhibit different parallel patterns:

- `parfib` is a divide-and-conquer program, which computes the Fibonacci number;
- `coins` is a divide-and-conquer program, which computes the number of ways to pay a given value from a given set of coins;
- `sumEuler` is a data parallel program, which computes the sum of the Euler totient function on each list interval.
- `parmap-of-parfib` is a data parallel program with nested divide-and-conquer parallelism, combining both patterns.

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), and all 32 nodes are connected through 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 Preliminary Performance Results

Table 2 summarises our results in terms of runtimes for the four micro-benchmarks, using GUM and GUMSMP respectively. These preliminary results do not represent systematic performance results and we therefore refrain from giving concrete speedup figures (for one thing, these represent results from individual rather than several runs). The numbers reported in the first column is the sum of cores used across nodes by GUMSMP. A current implementation limitation is that we cannot use the single bound task of a non-main PE, which results in this unusual sequence of core numbers. In the full version of the paper we plan to expand on these measurements, and additionally produce speedup results based on several runs.

These preliminary results are mainly a sanity check of the existing GUMSMP implementation, and thus one main result is that in most of the cases GUMSMP is competitive with the GUM RTE. We do observe some additional overhead of GUMSMP over GUM in the 1 PE cases especially for `parfib` and `coins` with slow-downs of 5% and 8%, respectively. We expect this overhead to be due to the additional management of HECs in the RTE, whereas in the flat design of GUM no such runtime mechanism is used.

In terms of runtimes for the large configurations, we observe that GUMSMP outperforms GUM for `parfib` and for `parmap of parfib`, the former being a pure divide-and-conquer program

Table 2: Runtimes for `parfib`, `coins`, `sumEuler`, and `parmap of parfib` on GUM and GUMSMP

	`parfib`		`coins`		`sumEuler` *without LWM*		`sumEuler` *with LWM*		`parmap of parfib`	
No. Cores	GUM	GUMSMP	GUM	GUMSMP	GUM	GUMSMP	GUM	GUMSMP	GUM	GUMSMP
1	1304.1	1372.19	1118.5	1213.7	1043.2	1119.2	1043.2	1119.2	477.0	479.0
4	331.0	361.7	257.6	341.7	266.7	278.1	242.1	279.1	133.1	125.40
7	188.5	169.2	148.1	196.3	144.5	188.3	140.3	134.4	70.86	59.0
10	122.1	111.7	108.9	179.2	105.2	191.4	103.1	90.6	48.9	41.2
13	94.7	84.1	82.2	131.3	86.6	167.4	79.8	67.3	38.6	30.85
16	78.4	70.5	77.2	165.1	81.6	151.0	71.1	56.8	31.2	25.31
19	65.5	59.1	65.8	119.4	65.9	156.8	61.0	48.1	26.5	20.59
22	60.6	50.9	60.1	147.8	61.8	131.4	57.1	41.0	22.7	18.21
25	55.4	45.2	51.6	170.0	54.8	113.5	54.3	37.3	20.8	16.1

generating massive amounts of parallelism, the latter being a combination of data-parallelism with (nested) divide-and-conquer. For the largest configurations of these two programs the performance improvements are 18% and 23%, respectively. Moreover, for both of these programs this performance gain increases with larger configurations, which suggests benefits in terms of scalability for GUMSMP, which is one of our main design goals.

However, the picture is significantly more negative for the `coins` and `sumEuler` examples, where GUMSMP is significantly slower than GUM by a factor of 3.3 and 2.1, respectively. This indicates a serious bottleneck in the current implementation and requires further investigation below.

5.3 The Impact of the Load Distribution Mechansims

The impact of the low-watermark mechanism: To identify the problem in `sumEuler`, Table 3 shows the average utilisation on each of the 3 PEs involved in a sample execution, using 4 HECs on each PE. The second column shows the utilisations without a low-watermark, the third column shows utilisations with low-watermarks. From this comparison it is clear that for `sumEuler` the low water mark mechanism plays a significant role in load balancing. In the first case, we use a pure work-stealing load distribution policy, where a PE asks for remote work only if all local work has been exhausted. In `sumEuler` most of the parallelism is generated at the start, and therefore this policy might lead to a delay that is too high for picking up sufficient work to keep all 4 HECs on each PE busy. Thus, the average utilisation of PEs 2 and 3 is only about 58% of the 400% possible in this execution on 4 HECs.

In contrast, when a low-watermark policy is used, all PEs will aim to fill up their spark pools to the low-watermark from the start. This results in a more speedy distribution of the available parallelism, and in turn to a higher average utilisation on the other PEs, shown in the third column of Table 3.

Table 3: Average utilisation on each PE for `sumEuler`

PE	`sumEuler` *without LWM*	`sumEuler` *with LWM*
1	257.95%	254.68%
2	58.88%	292.19%
3	58.87%	276.99%

To visualise this behaviour, Figure 5 and Figure 6 show the per-PE profiles of the activities when running the `sumEuler` program with and without low-watermarks. A per-PE profile shows PEs on the y- and time on the x-axis. In this configuration we see 3 bars, representing the 3 PEs used in the run. The darkness of the green value at each point in time shows the activity, i.e. the number of running HECs, as an average over a fixed time window. The statistics in Table 3 give these averages across the entire runtime of the program. Thus, per-PE profiles give an indication of the load-balance across PEs over time.

In the concrete per-PE activity profile in Figure 5 we observe that PE1 has more work (dark green) and a long active time compared to other PEs, which only have enough work to keep one HEC busy (light green), which is confirmed by the average utilisation in Table 3. The main reason for this behaviour is that `sumEuler` is data parallel, where the main HEC of PE1 is the only one generating sparks at the beginning of the execution. Other PEs will send FISH messages asking for work from PE1. In Figure 5, since there is no low-watermark applied, it will only receive one spark each time it sends a work requesting message. Therefore, the imported spark is executed by the main HEC, but the other HECs will remain idle all the time. In contrast, Figure 6 shows the behaviour when enabling low-watermarks, where the other PEs will keep sending messages requesting work until the number of sparks in all local spark pools reaches the low watermark. As a result, the utilisation on the other PEs is significantly higher, shown as darker green, and matches the utilisation statistics in Table 3.

The impact of the fish delay setting: The delay between receiving an own, unsuccessful FISH message and sending another FISH message is an important, tunable parameter for cluster-level execution. The setting needs to strike a balance between getting work as quickly as possible, and avoiding swamping the machine with FISH messages, which endangers the scalability of the system as such.

In all of the presented results we used small values for fish delay, to optimise quick work distribution. In GUMSMP the role of the fish delay value, which is inherited from GUM, is even more aggravated, because of 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 coming from a different HEC. This is reflected by longer idle times with moderate fish delay values, compared to the GUM executions. One immediate improvement would be to make the fish delay value dependent on the number of HECs in the PE. This however will again risk swamping the cluster with FISH messages. We therefore prefer a more aggressive use of watermarks over such an ad-hoc change to the fish delay.

5.4 Performance Bottlenecks

In analysing these performance results for the GUMSMP prototype, we identified several performance bottlenecks that we need to address in order to realise a truly scalable system.

The role of the gateway HEC: Our initial performance results reveal some bottlenecks in the current GUMSMP implementation, especially with higher numbers of cores. The main restriction at the moment is that the main HEC on each PE acts as a gateway for communication, which leads to a heavy system load and therefore a longer delay in requesting either data or work from remote PEs. On the other hand, we currently do not record load or other system

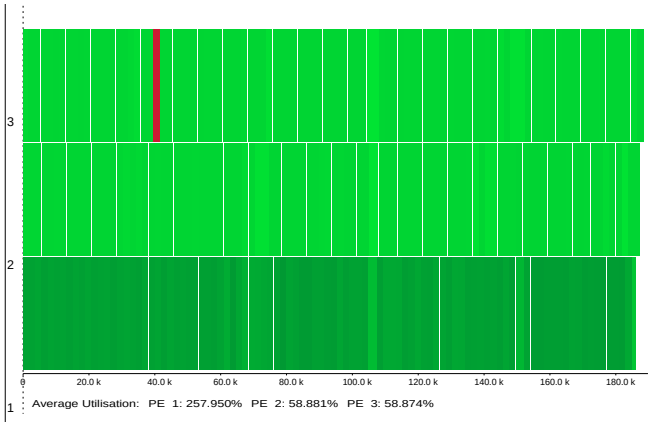

Figure 5: `sumEuler` load distribution *without low-watermark*

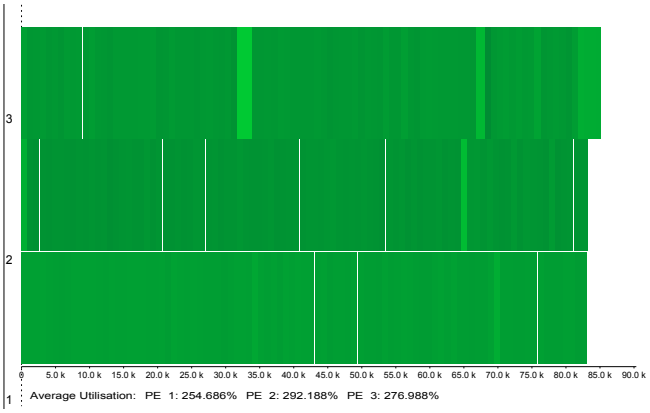

Figure 6: `sumEuler` load distribution *with low-watermark*

information in the gateway HEC and therefore cannot profit from the potential advantage of this design. In any case, to ensure rapid processing of pending messages by the main HEC, it would be advisable to throttle the actual computation being performed on it. This would reduce the communication and response bottleneck, however, we would potentially lose compute power for the parallel execution.

An alternative design would be to enable every HEC to send or receive messages and in this way distribute the communication over all cores. This might well be the most scalable solution in the GUMSMP design, but it might also lead to swamping the cluster with messages, especially FISH messages, despite only very little work being available. Therefore, a combination of distributing the communication across cores, together with active monitoring of the current system load seems to be the most promising direction, which we want to explore in the future.

The role of spark placement: At the moment an imported spark is added to the gateway HEC's spark pool. Since the distribution of work between HECs on one multi-core is fairly cheap, this does not seem to be problematic from a load balancing point of view. However, mixing local and imported sparks in the same pool might be problematic in terms of heap fragmentation, leading to more inter-processor pointers, and thus more communication. We have not yet been able to quantify this aspect of the parallel execution based on these results. Based on such an assessment we will revisit the policy for the placement of imported sparks.

To tackle potential heap fragmentation, we are considering adding an additional "import" spark pool to control the placement of sparks. Sparks in this pool might be further annotated by their PE of origin or other information that is useful for the scheduler. In most cases, the scheduler would prefer local sparks, but makes use of the import spark pool when no local ones are available. Once imported sparks have been turned into threads, the scheduler might prefer other imported sparks that have some affinity with the previous one, e.g. coming from the same PE.

6. Conclusion

We have presented the design and preliminary results of the new multi-level parallel Haskell implementation GUMSMP, designed for scalable, high-performance computation on networks of multi-cores. Our design focuses on flexible work distribution policies in hierarchical architectures. In particular, we aim for even but asymmetric load balancing, using different load distribution policies for the different levels in the hierarchy, accepting that on a large scale clusters will exhibit significant differences in the relative loads on multicores, but assuring that work can be cheaply stolen between processors on one multi-core. Our system mostly uses passive load distribution, employing work stealing to obtain either local or remote work. However, we refine this pure work-stealing policy with the concept of a low-watermark, which allows the system to prefetch work. This proves to be crucial for the performance of some of the test programs.

Our initial performance results on a set of micro-benchmarks are in the first instance a sanity check and functionality test of our implementation. Comparing the performance of the current GUMSMP implementation with the distributed memory GUM implementation shows a mixed picture. In some cases the hierarchical GUMSMP RTE outperforms the flat GUM RTE by up to 23%. However, we also observe significant bottlenecks in the current implementation, with a slow-down of up to 8% in the 1 PE configuration. A comparison of the load distribution between these executions reveals that the main HEC on the main PE becomes the bottleneck, having to mediate all the communication of the main PE, which in this case holds all the available parallelism of this flat, data-parallel example. Using a low-watermark, to pre-emptively acquire work, is crucial in order to avoid a massive slow-down in this case.

The implementation of GUMSMP is still under development. Our next goals are to extend the performance measurements to a broader class of applications and to focus on the scalability of some concrete applications that we have previously examined [12]. We are currently extending the monitoring support of GUMSMP, to integrate per-thread statistics as well and plan to use this information to develop a classification of parallel applications based on their dynamic behaviour. On a system level, we plan to explore some of the alternative design issues mentioned in Section 4. Based on previous work on the performance of the virtual shared memory abstraction [11], we expect that an import spark pool should reduce heap fragmentation and thus improve performance on large clusters. In the longer term we plan to modularise these key RTE policies in such a way that they can be easily modified and combined to deliver a customised RTE, without having to change the actual C implementation underneath. Such modularisation continues our previous efforts on a micro-kernel structured runtime-system, and has more recently been described in [3].

Acknowledgments

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 UKs Engineer-

ing and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP:High Performance Computational Algebra and Discrete Mathematics.

References

[1] M. Aswad, P. Trinder, A. A. Zain, G. Michaelson, and J. Berthold. Low Pain vs No Pain Multi-core Haskells. pages 49–64, Selye Janos University, Komarno, Slovakia, May 2009. Intellect.

[2] J. Berthold, S. Marlow, K. Hammond, and A. 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. ISBN 978-0-7695-3803-7. doi: 10.1109/ICPPW.2009.10. URL http://dx.doi.org/10.1109/ICPPW.2009.10.

[3] J. Berthold, H. Loidl, and K. Hammond. PAEAN: Portable Runtime Support for Physically-Shared-Nothing Architectures in Parallel Haskell Dialects. *Journal of Functional Programming*, 2013. Special issue on Run-Time Systems and Target Platforms for Functional Languages. Submitted.

[4] 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*, 21:102–111, 1994.

[5] 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.

[6] 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.

[7] 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.

[8] 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. ISBN 978-1-4503-1054-3. doi: 10.1145/2364527.2364562. URL http://doi.acm.org/10.1145/2364527.2364562.

[9] 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.

[10] S. L. P. Jones, C. Hall, K. Hammond, J. Cordy, H. Kevin, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a Technical Overview, 1992.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] N. Ramsey. Concurrent Programming in ML. Technical report, 1990.

[20] 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.

[21] 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://www.macs.hw.ac.uk/~dsg/gph/papers/abstracts/gum.html.

[22] P. W. Trinder, E. Barry Jr., M. K. Davis, K. Hammond, S. B. Junaidu, U. Klusik, H.-W. Loidl, and S. L. Peyton Jones. GpH: An Architecture-Independent Functional Language. Submitted to IEEE Transactions on Software Engineering, special issue on Architecture-Independent Languages and Software Tools for Parallel Processing, July 1998.

[23] 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.