JFP **26**, e10, 39 pages, 2016. © Cambridge University Press 2016
doi:10.1017/S0956796816000010

PAEAN: Portable and scalable runtime support for parallel Haskell dialects

JOST BERTHOLD*

Dept. of Computer Science (DIKU), University of Copenhagen, Commonwealth Bank of Australia, Sydney
(*e-mail:* jberthold@acm.org)

HANS-WOLFGANG LOIDL

School of Mathematical and Computer Sciences, Heriot-Watt University
(*e-mail:* hwloidl@macs.hw.ac.uk)

KEVIN HAMMOND

School of Computer Science, University of St.Andrews
(*e-mail:* kevin@kevinhammond.net)

Abstract

Over time, several competing approaches to parallel Haskell programming have emerged. Different approaches support parallelism at various different scales, ranging from small multicores to massively parallel high-performance computing systems. They also provide varying degrees of control, ranging from completely implicit approaches to ones providing full programmer control. Most current designs assume a shared memory model at the programmer, implementation and hardware levels. This is, however, becoming increasingly divorced from the reality at the hardware level. It also imposes significant unwanted runtime overheads in the form of garbage collection synchronisation etc. What is needed is an easy way to abstract over the implementation and hardware levels, while presenting a simple parallelism model to the programmer. The PArallEl shAred Nothing runtime system design aims to provide a portable and high-level *shared-nothing* implementation platform for parallel Haskell dialects. It abstracts over major issues such as work distribution and data serialisation, consolidating existing, successful designs into a single framework. It also provides an optional *virtual* shared-memory programming abstraction for (possibly) shared-nothing parallel machines, such as modern multicore/manycore architectures or cluster/cloud computing systems. It builds on, unifies and extends, existing well-developed support for shared-memory parallelism that is provided by the widely used GHC Haskell compiler. This paper summarises the state-of-the-art in shared-nothing parallel Haskell implementations, introduces the PArallEl shAred Nothing abstractions, shows how they can be used to implement three distinct parallel Haskell dialects, and demonstrates that good scalability can be obtained on recent parallel machines.

1 Introduction

This paper studies how to implement parallelism portably and abstractly for different dialects of parallel Haskell targeting modern multicore/manycore/cluster/cloud

* Corresponding author. Reported work performed while at the University of Copenhagen (DIKU).

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

systems. In order to improve energy efficiency and performance, there is a strong trend towards reducing sharing in the underlying parallel hardware (Lameter, 2013). However, exposing sharing concerns to the programmer moves away from the traditional functional programming strengths of increasing programmability and maintaining very high levels of abstraction. In this paper, we describe a runtime system (RTS) approach that enables us to separate sharing at the language, implementation and hardware levels. In this way, we can provide a possibly *shared-nothing implementation* for a possibly *logically shared language design*, for example. We can thus obtain the benefits of reducing or eliminating sharing at the hardware level, without losing the benefits of high-level abstraction at the language level. By providing a coherent framework of sophisticated RTS operations, it becomes possible to develop implementations of new language extensions quickly and robustly. The same approach also enables extensibility of the RTS. This helps overcome two of the key obstacles to developing new parallel RTSs, while maintaining functional programming properties of abstraction and ease of use at the language level. Our approach is evaluated against three existing and contrasting dialects of Parallel Haskell: Glasgow parallel Haskell (GpH) (Hammond & Peyton Jones, 1990; Trinder *et al.*, 1995), Eden (Loogen *et al.*, 2005) and EdI (Berthold & Loogen, 2007; Berthold, 2008).

1.1 Contributions

This paper makes the following *contributions:*

- We expose and evaluate the key design decisions underlying the GpH, Eden and EdI Parallel Haskell dialects, focusing particularly on sharing, parallelism control and load distribution;
- We describe a high-level, portable RTS framework, PAEAN (PArallEl shAred-Nothing), that builds on the widely used GHC[1] implementation and that can be used to implement a variety of parallel dialects of Haskell, for *possibly shared-nothing* parallel systems;
- We demonstrate how PAEAN can be used to implement different dialects of Parallel Haskell, in particular GpH and Eden, capturing all of the requirements of the previously-constructed specialised GRAPH on a Unified Memory Model (GUM) and DREAM RTSs;
- We evaluate the performance and scalability of the parallel Haskell implementations that follow PAEAN ideas and principles and
- We discuss the state-of-the-art for distributed-memory parallel Haskell implementations, relating different programming models to the respective runtime support and assessing the merits of RTS versus library support for coordinating parallel programs.

[1] The Glorious Haskell Compiler, `https://www.haskell.org/ghc/`

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 1. *Library-based and runtime-system-based parallel Haskell*

	Library	RTS
Code bloat	High	Low
Abstraction level	Low	High
Language integration	Poor	Good
Communication abstraction	Pervasive	Localised
Performance	Hard to tune	Easily tunable
Implementation cost	Low	*High*
Extensibility	Easy	*Hard*

1.2 Library versus runtime system

A number of recent designs have chosen to implement parallel Haskell constructs exclusively at the library level, e.g. (Foltzer *et al*., 2012; Maier & Trinder, 2012). In contrast, we have deliberately taken a RTS approach. As summarised in Table 1, there are many advantages to our approach. First, less coordination code is required to manage parallelism and load distribution because an underlying RTS will take care of all the details without programmer intervention. This significantly improves programmability and productivity. RTS-based approaches can abstract most of the coordination into language constructs, which simplifies programming, reduces code clutter and gives better integration with the programming language. This is especially true for communication abstractions, which are pervasive in any library-based approach for shared-nothing approaches. Second, performance-critical code is much easier to fine-tune and avoids the functionality limitations that are inherent to a library approach (for instance, custom memory management is simple and direct). The key disadvantages to an RTS approach are: first, that the cost of implementation can be significantly higher, since features must be considered throughout the language implementation; and second, that it can be difficult to incorporate new features. By taking a modular approach, PAEAN addresses both of these problems, making it easier to design and build new parallel RTSs.

1.3 Why sharing matters

In Haskell, the sharing of the program and its data through *graph reduction* is fundamental to achieving lazy evaluation. In a parallel setting, this creates a tension since inter-thread sharing can impose unwanted synchronisation. At a hardware level, shared memory is increasingly a performance bottleneck. In order to improve time and energy efficiency, there is therefore a strong trend towards *non-uniform memory architectures* (NUMA) in multicore/manycore processors (Lameter, 2013), where memory is not treated as a single homogeneous, easily addressable array. At a software level, there is a correspondingly strong trend to avoid locks, wherever possible, and to minimise expensive cache coherency by exploiting non-shared memory. We can obtain a more efficient *and more scalable* implementation by not assuming the existence of shared memory at a hardware or implementation

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

level. Scalability is enhanced by reducing shared hotspots and memory bottlenecks. Such a *shared-nothing* approach (Gray, 1985) is also easier to deploy on large-scale systems, including clusters and distributed systems. We would also argue that it is easier to implement such an approach, even on a relatively small scale, since issues of cache coherency, synchronisation etc. are exposed in the implementation, and not hidden through shared memory accesses.

If memory cannot be assumed to be shared, then a number of implementation issues need to be addressed. These include:

1. how to serialise data;
2. how to logically share values between physically disjoint heaps;
3. how to manage global garbage collection (GC) while minimising synchronisation costs;
4. how to migrate data, potential parallelism and threads.

Some recent approaches such as Cloud Haskell (Epstein *et al.*, 2011) and HdPH (Maier & Trinder, 2012) delegate much of this work to the applications programmer, making parallelism control fully explicit in the program. While this minimises the work of the systems implementor, such an approach is likely to suffer from replication, errors and unexpected feature interaction, as well as to cause incompatibilities between applications. In our opinion, fully explicit parallelism control entangled with the application flies in the face of the usual functional programming philosophy of good abstraction, effectively "throwing out the baby with the bathwater". This paper takes the opposite approach, by focusing on a *mostly implicit* model of parallelism at the source level, where parallelism is specified in an abstract, declarative way and by supporting this with a sophisticated, well-engineered and flexible RTS.

1.4 Paper structure

The remainder of the paper is structured as follows. Section 2 describes the high-level programming models for Parallel Haskell that provide the requirements for the PAEAN implementation. Section 3 describes the PAEAN design and implementation. Section 4 evaluates the utility and scalability of the PAEAN approach. Section 5 covers related work, including extensions to PAEAN. Finally, Section 6 concludes.

2 Parallel programming models for Haskell

A wide variety of parallel extensions to Haskell have been proposed. A key design question is how explicit to make the coordination of the parallel execution. One extreme is a purely implicit approach, as taken by pH (Aditya *et al.*, 1995), where there are no extensions to the language and all parallelisation is performed by the compiler. Data parallel extensions to Haskell, such as DpH (Chakravarty *et al.*, 2007), take a similar approach, but are limited to data-parallelism on specific data structures. While implicit parallelism is appealing to the programmer, it also limits the scope for performance tuning. The other extreme is to make most aspects of the

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
sumEuler   :: Int → Int
sumEuler n = sum [ euler i | i ← [n, n−1..1] ]

euler   :: Int → Int                    -- Euler phi function
euler n = length ( filter  (relprime n) [1..n−1])

relprime  :: Int → Int → Bool    -- are x and y coprime
relprime x y = hcf x y == 1

hcf    :: Int → Int → Int          -- highest common factor of x and y
hcf x 0 = x
hcf x y = hcf y (rem x y)
```

Fig. 1. The basic Euler totient function, `sumEuler`.

coordination explicit in the program and thus give the programmer more control on how the parallelism is used. In the Haskell world, examples of this approach include the Par-Monad (Marlow *et al.*, 2011) and Cloud Haskell (Epstein *et al.*, 2011). While they are easier to implement and tune than implicit approaches, such approaches have the disadvantage of introducing all the complexities and lack of abstraction from mainstream parallel programming models. In such an approach, *applications programmers* must become *systems programmers*. The approach that we will mainly focus on is therefore a middle one of *semi-explicit* parallelism, where the programmer only needs to identify (potential) parallelism, but where synchronisation and communication is managed by the RTS. This greatly simplifies the parallel programming task, but still gives the programmer a concrete handle on how to improve parallel performance. Supporting such an approach requires an elaborate RTS, however. The focus of this paper is on how to design such a RTS in a generic, flexible and reusable way. A more detailed discussion of related systems, including ones taking both more explicit and more implicit approaches, is given in Section 5.

In this paper, we will consider three Haskell dialects: GpH, Eden and EdI (the implementation language for Eden). The approach taken by GpH uses programmer specified annotations (`par`) to create "sparks" that may, or may not, be subsequently transformed into parallel threads. In contrast, in Eden and EdI, parallelism is introduced by `process` constructs that always create new threads. While Eden provides a purely functional interface, EdI uses monadic constructs in a similar fashion to the Par-Monad. We will use the `sumEuler` function to characterise these three dialects. For a given n, `sumEuler` computes the Euler φ-function[2] for values up to n, and sums the results. The sequential Haskell code for this is shown in Figure 1.

[2] $\varphi(i)$ counts how many numbers smaller than i are coprime to i. It can of course be computed more efficiently using a prime factorisation of i. The naïve version shown here is just for benchmarking.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
type Strategy a = ...                          -- evaluation strategy abstraction

rseq :: Strategy a                             -- normal sequential evaluation
rpar :: Strategy a                             -- spark a closure

rdeepseq :: NFData a => Strategy a             -- full evaluation

parList :: Strategy a -> Strategy [a]          -- parallel evaluation of a list

using :: a -> Strategy a -> a                  -- strategy application
```

Fig. 2. Basic evaluation strategies.

2.1 GpH

GpH (Hammond & Peyton Jones, 1990; Trinder *et al.*, 1995) uses a semi-explicit parallelism model, where the programmer simply indicates potentially parallel closures and all synchronisation, coordination *etc.* issues are delegated to the RTS. It uses two basic primitives: `par` and `pseq`.

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

The `par` primitive identifies potential parallelism using *lazy futures* (Mohr *et al.*, 1991). Its first argument is a closure that is marked for possible parallel execution (*sparked*). It returns the value of its second argument. So, `x ‘par‘ e` marks `x` for parallel execution, and then returns `e`. The `pseq` primitive sequences the evaluation of its arguments, returning the value of its second argument. So, `x ‘pseq‘ e` first evaluates `x`, and then returns `e`.

Experience has shown that unstructured use of `par` and `pseq` can quickly obscure programs. A higher level of abstraction is provided by *evaluation strategies* (Trinder *et al.*, 1998; Marlow *et al.*, 2010), which cleanly separate coordination from computation. Evaluation strategies are lazy, polymorphic, higher order functions that control the evaluation degree and parallelism of a Haskell expression. Some primitive strategies are shown in Figure 2. The `rseq` and `rpar` strategies are analogous to `pseq` and `par`. The `rdeepseq` strategy is similar to `rseq`, except that it fully evaluates its argument[3]. The `parList` strategy is an example of a higher order strategy. This takes another, possibly parallel, strategy as its argument, and applies it in parallel to all elements of a list. So, for example, `parList rdeepseq` is the strategy that evaluates a list in parallel, forcing all the results to be completely evaluated. Finally, the `using` function applies a strategy to a Haskell expression. So, `e ‘using‘ parList rdeepseq` will evaluate `e` using the `parList rdeepseq` strategy.

Our first GpH version of `sumEuler`, `sumEulerGpH0` (Figure 3), simply uses `parList rdeepseq` to compute every application of `euler` in parallel. However, the *granularity* of the parallelism is too fine, since each spark only describes a single function application. `sumEulerGpH` therefore uses *chunking* to overcome this.

[3] i.e. it evaluates to *full normal form* rather than the Haskell default of *weak head normal form*. The `NFData` class defines the recursive evaluation strategy to implement this full evaluation.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
sumEulerGpH0 :: Int → Int
sumEulerGpH0 n = sum (map euler [n, n−1..1]
                        `using` parList rdeepseq)

sumEulerGpH :: Int → Int → Int
sumEulerGpH z n = sum (map workF (unshuffle z [n, n−1..1])
                        `using` parList rdeepseq)
          where workF = sum ∘ map euler
```

Fig. 3. GpH versions of `sumEuler`.

```
splitAtN    :: Int → [a] → [[a]]
splitAtN n [] = []
splitAtN n xs = ys : splitAtN n zs
                where (ys,zs) = splitAt n xs

unshuffle :: Int → [a] → [[a]]
unshuffle n xs = map (takeEachN) [ drop i xs | i ← [0.. n−1]]
    where takeEachN []     = []
          takeEachN (x:xs) = x : takeEachN (drop (n−1) xs)
```

Fig. 4. Chunking functions.

We therefore introduce an additional parameter `z`, the desired number of chunks (i.e. sparks). The n list elements are then distributed into z sub-lists. Each sub-list becomes a spark that may then potentially be evaluated in parallel. Alternative chunking functions can be easily defined as standard Haskell definitions[4]. For example, as shown in Figure 4, `splitAtN` splits a list into sub-lists of size n, and `unshuffle` distributes elements of a list into exactly n sub-lists, in a round-robin fashion. For `sumEulerGpH`, we have chosen to use `unshuffle` rather than `splitAtN` because the computation time for the φ-function increases with its argument. If `splitAtN` was used instead, then the largest argument values would be gathered into the first sparks that were created, creating load-imbalance between the sparks.

2.2 Eden

In contrast to GpH, Eden provides explicit process abstraction and process instantiation operations (Figure 5). The expression `process ( λx → e )` of type `Process a b` denotes a *process abstraction* over a function `λx → e` of type `a → b`. The (#) operator allows a new *child* process to be *instantiated*, i.e. evaluated in parallel with the calling *parent* process. `process (\ x -> map factorial x) # [1..100]` creates a process that maps the factorial function over the list `[1..100]`. This list is evaluated on the parent, and passed in a fully evaluated form to the child. Once computed, the result list is then returned to the parent. Eden's `spawn` function

[4] Chunking functions can also be provided abstractly for lists and other data types via a `Cluster` type class (Totoo & Loidl, 2014).

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
-- Process abstraction and instantiation
process :: (Trans a, Trans b) ⇒ (a → b) → Process a b
( # )   :: (Trans a, Trans b) ⇒ (Process a b) → a → b
spawn   :: (Trans a, Trans b) ⇒ [Process a b] → [a] → [b]
```

Fig. 5. Eden core constructs.

```
sumEulerEden :: Int → Int
sumEulerEden n = sum (spawn (repeat childProc) (unshuffle noPe [1..n]))
                 where childProc = process (sum ∘ map euler)
```

Fig. 6. Eden version of `sumEuler`.

lifts `process` to a list of processes and inputs. All processes are created eagerly in parallel. This avoids the sequential demand-driven evaluation that the lazy Haskell list would imply. Once instantiated, the new child process will be executed in parallel with the parent process. The parent and child processes have independent heaps and will only communicate by exchanging (implicit) messages. Data that is exchanged between Eden processes is always in normal form (i.e. fully evaluated). Therefore, process instantiation is roughly equivalent to hyper-strict function application. Lists are communicated as streams, element by element. Tuple components are evaluated by concurrent threads. This allows for circular programs and infinite data streams.

Figure 6 shows how `sumEuler` can be implemented in Eden by spawning a number of independent processes that each apply `euler` to part of the original input list. Having created these processes, the main process will block until the results become available, and then compute the final sum. Since, unlike GpH, Eden will create threads for every `process` construct that is specified by the programmer, we introduce a use of the `unshuffle` function to balance the load between child processes. This will evenly divide the inputs among the number of available processors, `noPe` (provided as a constant by the Eden RTS).

2.3 EdI

The Eden implementation language EdI (Berthold & Loogen, 2007; Berthold, 2008) (Figure 7) takes the approach of explicit process control even further, requiring the programmer to take full control of, and responsibility for, all data transfers. As suggested by its name, EdI was designed to implement Eden, but can also be used in its own right as a monadic language for parallelism. As in Eden, a new parallel process can be spawned on a specific instance of a running system but, unlike Eden, an EdI process is simply an `IO` action. EdI processes communicate explicitly using typed one-to-one channels. Channels are created using `createC`, which returns a *channel name*, of type `ChanName' a` and a placeholder result of type `a`. A channel name can be explicitly communicated to another process, using `sendWith`, or implicitly passed to a new process when it is created using `spawnProcessAt`. The other process can use the channel to return data (of a suitable type) to the original process. Note that the `sendWith` operation takes an evaluation strategy argument. This controls where the evaluation happens and is applied to the data argument before it is

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
spawnProcessAt :: Int → IO () → IO ()

data ChanName' a
createC :: IO (ChanName' a, a)
createCs :: Int → IO ([ChanName' a], [a])

sendWith :: Strategy a → ChanName' a → a → IO ()
sendStreamWith :: Strategy a → ChanName' [a] → [a] → IO ()

noPe, selfPe :: IO Int
```

Fig. 7. The complete API for the Eden implementation language, EdI.

```
sumEulerEdi :: Int → IO Int
sumEulerEdi n = do pes ← noPe
                   (cs, rs) ← createCs pes
                   zipWithM_ (spawnEulerW pes) [1..pes] cs
                   return (sum rs)
  where spawnEulerW :: Int → Int → ChanName' Int → IO ()
        spawnEulerW stride k c = spawnProcessAt k
                                 (sendWith rseq c
                                   (sum (map euler [n−k+1, n−k+1−stride..1]))
```

Fig. 8. EdI implementation for `sumEuler`.

transmitted. Finally, the `sendStreamWith` operation extends `sendWith` to streams. This enables the definition of circular process networks and pipelining.

Figure 8 shows how `sumEuler` can be implemented in EdI. A set of channels, `cs`, and placeholders, `rs`, are created using `createCs`. One channel and one placeholder are created for each available processor (`noPe`). Each channel is used in one call to `spawnEulerW`, which creates a child process on processor k using `spawnProcessAt`. The new process will generate its own part of the input list, apply the `euler` function to this partial input, and sum the resulting list. This partial sum is returned to the parent by calling `sendWith` on the given channel. EdI represents the lowest level of abstraction that we will consider in this paper, exposing the full control, *and burden*, of parallel evaluation and explicit communication to the programmer. It constitutes a minimal set of RTS support functionality for sending and receiving data and computations, exploiting the underlying GHC RTS to achieve synchronisation.

3 Haskell on shared-nothing systems: The PAEAN runtime framework

The language characteristics of Eden, GpH and EdI can be classified into the four key concepts of *parallelism*, *binding*, *synchronisation* and *workload management* (Table 2). Each of these will lead to a number of important *implementation requirements*.

Parallelism. As we have seen, in Eden and EdI, all threads that are specified by the programmer *must* be created (parallelism is *mandatory*). In contrast, in GpH, the RTS decides both when and whether to instantiate the potential parallelism that

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 2. *Language characteristics of GpH, Eden and EdI, and implementation concepts*

Concept Realisation	Eden	EdI	GpH
Parallelism	*Explicit **process** constructs*		*Implicit **par** annotations*
Threads	implicit threads *mandatory* parallelism explicit task distribution	explicit threads *mandatory* parallelism explicit task distribution	implicit threads *advisory* parallelism implicit task distribution
Binding	*data exchange through process application*		*variables act as futures*
Memory Management	distributed heaps (per process)		virtual-shared memory
Synchronisation	*Implicit through process arguments and results*		*Implicit variable sharing*
Communication	implicit and hyper-strict (on process arguments)	explicit, including evaluation control	implicit and lazy (on shared data)
Workload Management	*Explicit logical processor ids*		*Implicit*
Workload distribution	default (round-robin) or explicit process placement		automatic spark distribution (work-stealing)

has been indicated by the programmer (parallelism is *advisory*). The corresponding *requirements* for *thread management* are:

- to provide mechanisms to support the creation of parallelism;
- to decide on whether or not to instantiate potential parallelism;
- to decide on when to instantiate potential parallelism.

Binding. The concept of variable *binding* is realised in Eden and EdI via data transfer, initiated by a process creation, and in GpH via *lazy futures* and implicit synchronisation. This concept is implemented by the memory management component. For Eden, this links remote data access with the language concept of process abstraction, requiring explicit data transfer. For GpH, the main requirements are completely transparent access to bindings, which leads to automatic and distributed memory management over several physical heaps, and hence to a *virtual-shared memory* abstraction. More specifically, the requirement on the RTS is to ensure

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

that all data is available at the location where the computation is to be executed. This implicitly invokes communication, so establishing a tighter link between these components in the GpH implementation. In contrast, the RTS for Eden and EdI must implement a model of distributed heaps that are connected through communication channels. The *requirements* for *parallel memory management* are:

- to identify points of data exchange;
- to implement a logically shared address space on top of distributed memory;
- to interact with the communication component based on the need for remote data.

Synchronisation. *Synchronisation* is required to (i) avoid evaluating a value multiple times; (ii) communicate values between threads and (iii) ensure that results are shared between the evaluating thread and the threads that use those results. In EdI, the programmer is responsible for all communication and for ensuring that evaluation is performed appropriately. In Eden, the argument to a process is fully evaluated to full normal form by the parent process and communicated incrementally. The child process will *block* if it requires some part of the data that has not yet been evaluated and communicated to it. In contrast, in GpH, some data may be passed to a child thread when it is created, but the majority will be fetched on a demand-driven basis. This increases sharing and reduces startup costs, but may add some delay if a significant volume of data is fetched during execution. Having created one or more child processes, in Eden, the parent process will typically *block* until the results become available. In GpH, however, execution continues normally. If any GpH thread requires a value that is being evaluated by another thread, it will block until the result is produced. When the value of a virtually shared node becomes available, it will be returned to all blocked threads. This reduces synchronisation delays, but once again, shared data may be fetched incrementally. This uses an extension of the standard "black-holing" mechanism that is normally used to detect cycles in sequential code. Rather than raising an exception, as would normally happen, the parallel implementation treats an attempt to evaluate a node that is currently under evaluation as a synchronisation request[5]. The virtual shared graph model used in GpH will ensure that the results of computations will be shared as they become available, so avoiding repeated evaluation. In contrast, Eden and EdI make no attempt to avoid repeated evaluation other than through transmitting only normal forms. The *requirements* for *communication* are therefore:

- to check whether necessary data is available;
- to realise data exchange when necessary;
- to update values with results as they are produced;
- to notify blocked threads of the availability of a result;
- to interact with the thread management component, based on data availability.

[5] A full description of this mechanism can be found in Hammond (2011)

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

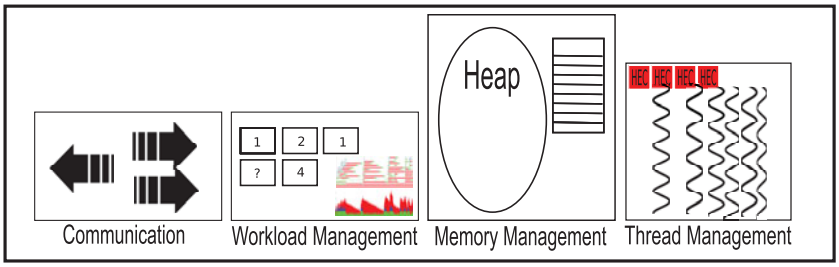


Fig. 9. The PAEAN runtime system components.

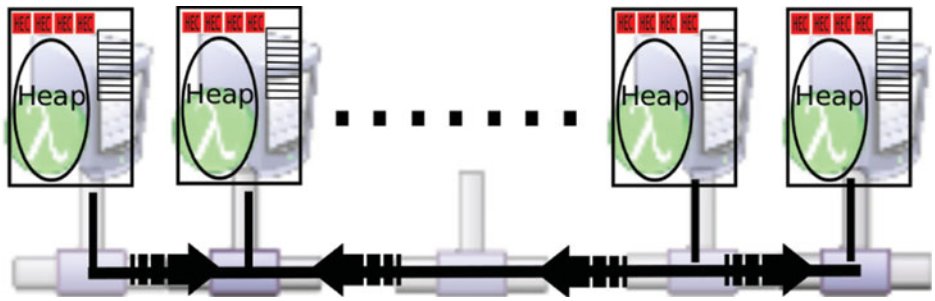


Fig. 10. Parallel system built as a collection of PAEAN instances.

Workload management. A key runtime decision is when and where to execute parallelism. At the language level, Eden and EdI both provide a notion of virtual processor, which can be used by the programmer to offload processes to specific processors. GpH abstracts this even further: threads are instantiated from sparks and then mapped to the available processors by the RTS. In order to get a good load balance, it may be necessary to migrate sparks and/or threads between processors. Finally, where multiple threads exist on a single processor, it is necessary to schedule them appropriately both for efficiency and, in the case of Eden and EdI, to ensure *fairness*. The *requirements* for *workload distribution* are:

- to decide on the order of execution of available threads (scheduling);
- to determine how to distribute the available parallelism (load balancing);
- to decide whether and when to offload parallelism.

As shown in Figure 9, PAEAN realises each of these issues as its own modular *component*. This makes PAEAN much more generally applicable. Although they have formed our starting point, the PAEAN system design is not restricted to the GpH and Eden parallelism models. Neither is it necessarily limited to Haskell: the concepts and realisations that we have shown in Table 2 are characteristic of a much wider class of languages, including some parallel dialects of Lisp, Prolog and C++. Any language that is more prescriptive in terms of thread management can easily be mapped to the underlying PAEAN realisations, though it may not make full use of the PAEAN mechanisms.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 3. *PAEAN runtime system routines for sparks and threads*

PAEAN Routine and Parameters	Description
Spark Management	
`void createSpark(StgClosure *la)`	Generate a spark and put it into the local spark pool
`StgClosure* findSpark()`	Select and return a spark from the the local spark pool
`void activateSpark(StgClosure *spark)`	Turn a spark into a thread, allocating a thread, and put it into the thread pool
Thread Management	
`void startNewProcess(StgClosure *la)`	Create a new thread in its own process, to evaluate given (`IO`) closure
`StgTSO* createIOThread(StgClosure *la)`	Create a new thread (sharing a process with other threads) to evaluate given closure

3.1 PAEAN thread management

PAEAN coordinates multiple *instances* of the standard GHC RTS. Each PAEAN instance is typically mapped onto a separate processor in its own operating system process. When run on a multicore, an instance may comprise multiple Haskell Execution Contexts (HECs) which share the same heap, i.e. the standard shared-memory parallel Haskell implementation of GpH, GHC-SMP (Marlow *et al*., 2009). These instances are connected to form a collaborating parallel system with a distributed heap, as shown in Figure 10. Each PAEAN instance manages its own *thread pool* and, if required, a *spark pool*, providing its own independent thread scheduler. In addition, PAEAN introduces the concept of a *process*, a thread group that shares a region of the Haskell heap and communication channels (explained in Section 3.4.3). As shown in Table 3, PAEAN routines are provided to: spark closures and add them to the local spark pool (`createSpark`); select a spark from the local spark pool (`findSpark`); turn a spark into a thread and move it from the spark pool to the thread pool (`activateSpark`); add an IO thread directly to the local thread pool (`createIOThread`) and start a new process (`startNewProcess`).

3.2 Virtual shared heap management

Following our shared-nothing design, each PAEAN instance maintains its own independent heap, with its own address space. Each instance undertakes its own local garbage collection (GC) on this heap, using, for example, the usual GHC GC mechanism. By default, this is a stop-and-copy *generational garbage collector* (Appel, 1989). On a shared-memory system, the GHC-SMP shared-memory GC implementation may be used, if desired, though this may degrade performance. It is possible

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

to combine the heaps from multiple PAEAN instances to form a single virtually shared heap that can be accessed by any PAEAN instance. Shared heap nodes are accessed through a shared *global* address (GA) space.

Table 4. *PAEAN runtime system routines for global heap management*

PAEAN routine and parameters	Description
`RtsGA* makeGlobal()`	Create a new unique global address (GA) on the current node
`StgClosure* lookupGA(RtsGA *ga)`	Look up global address `ga` in the GIT, return a local address (LA) if found
`RtsGA* lookupLocal(StgClosure *la)`	Look up local address `la`, return GA if found
`void commonUp(RtsGA *ga, StgClosure *la)`	Check whether a `ga` already exists on this node; if so, keep only the further evaluated of the two (a data duplicate is eliminated)
`void splitWeight(RtsGA *ga1, RtsGA *ga2)`	Split weight attached to `ga1`, into 2 components adding up to the original, one attached to `ga1` and one attached to `ga2` (a reference is created)
`RtsGA* addWeight(RtsGA *ga)`	Add the weight in `ga` to the weight of the GA in the local GIT table (a reference is deleted)
`void markGIT()`	Traverse in-pointers section of GIT during GC (local closures with GAs are roots)
`void rebuildGIT`	Traverse GIT to fix references to local addresses (out-pointers), return weight for dead entries

3.2.1 Global addresses

The virtual shared heap forms a subset of the individual PAEAN instance heaps. Each (potentially) shared heap object is identified by a unique GA, created by `makeGlobal`. Unshared objects exist only in the local heap for the PAEAN instance and so do not have a GA. Only selected unevaluated objects need to be shared, other objects referenced from them will be handled transparently, and evaluated data can simply be copied. This minimises the number of globally addressed objects and brings benefits in terms of performance (quicker lookup), space leaks (fewer external pointers) and memory usage (smaller pointers). This design is based on the assumption that only a small fraction of any given instance heap is globally visible, as verified on the GRIP architecture (Peyton Jones *et al.*, 1987; Hammond

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

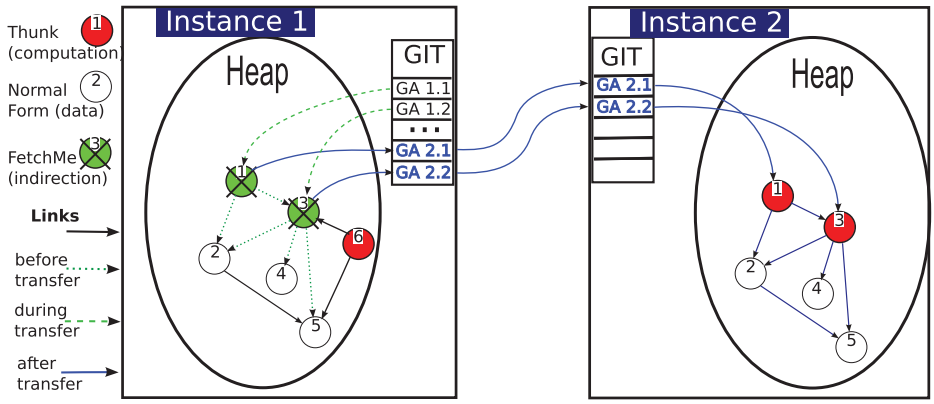


Fig. 11. Transferring graph structures between PAEAN instances. Here, the sub-graph rooted at address 1 (with `GA2.1`) has been transferred from instance 1 to instance 2. Thunks are moved, replacing them by *FetchMes* on instance 1, while normal forms are copied. `GA1.1` and `GA1.2` are only used during transfer. At completion of the transfer, they are replaced by `GA2.1` and `GA2.2`.

& Peyton Jones, 1992). Each PAEAN instance maintains tables of *in-pointers* from other PAEAN instances (a *Global In-Pointer Table*, or GIT), which provide external references to shared objects that are defined within the instance. By maintaining a separate table, we avoid the need to record a GA in every allocated closure, and also make the inter-heap dependencies explicit. This allows an instance to freely choose any appropriate local GC mechanism — it is only necessary to treat the global in-pointers as additional GC roots. As discussed in Section 3.2.3, *global GC* is performed using weighted reference counting. The PAEAN virtually shared heap implementation that is described here mostly follows that of Trinder *et al.* (1995), but reduces the amount of heap management information that is needed. The most notable new extensions are user-tunable policies for globalising heap structures and for the size of a sub-graph to be transferred.

3.2.2 *Transferring program graph between PAEAN instances*

Every object which is moved from one PAEAN instance to another is given a new, unique GA in the target instance's GIT. The original local heap node is replaced with a special *FetchMe* closure that refers to this GA. When the value of a *FetchMe* node is needed, the communication subsystem is invoked to obtain the value of the node from the remote PAEAN instance. In this way, local heap nodes are linked to remote heap nodes. The `commonUp` routine is used to avoid replicating data where the same GA is communicated to an instance more than once. An example is shown in Figure 11. Here, two *thunks* (nodes 1 and 3) are moved between heaps, replacing the original nodes with *FetchMe* nodes. The packing routine checks whether the thunks are already global. If they are not, GAs are allocated locally for each thunk (`GA1.1`–`GA1.2`). These GAs are used to refer to the thunks while they are in transit. When the thunks are unpacked, new GAs (`GA2.1`–`GA2.2`) are generated to indicate

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

the transfer of ownership to **Instance 2**. An acknowledgement message informs **Instance 1** that `GA1.1` and `GA1.2` have been superseded by `GA2.1` and `GA2.2`. The latter become the permanent remote references, whereas the former become garbage. Figure 11 shows the final state, with *FetchMes* on **Instance 1** referring to GAs on **Instance 2** (`GA2.1`–`GA2.2`).

3.2.3 Global garbage collection

A *weighted reference counting* scheme (Bevan, 1987) is used to garbage collect the independent PAEAN heaps. As we have seen, the GIT indicates the possibly live *in-pointers* for each instance. These are marked as roots during GC using the `markGIT` routine. Weights are associated with each GA. Whenever a GA is shared between instances, e.g. because a *FetchMe* has been replicated, then its weight is split between the two relevant GIT entries using `splitWeight`. Conversely, when an instance frees a remote closure during its local GC, the weight is returned to the original GIT using `addWeight`. The same happens when old GAs are replaced by new ones. When all the weight is concentrated in the original GIT, then the closure is no longer referenced globally and the GIT entry can be removed. The local closure can then be freed if it is not referenced locally. One general weakness of this scheme is that it will not collect cycles across independent heaps. Various schemes can be used to deal with this issue, if needed, such as collating cross-heap cycles onto a single instance, where they can be collected (Bevan, 1987). Such a mechanism, however, is not currently implemented in PAEAN.

3.3 Haskell program graph serialisation

Haskell program graphs are communicated between RTS instances as *serialised sub-graphs* (see Figure 12). These are packed into one or more packets for efficient communication. PAEAN provides methods to traverse a graph in a local heap, packing it for efficient transmission (`packGraph`), and to unpack a serialised graph into the remote heap (`unpackGraph`), restoring its original structure and building any inter-instance links using new FetchMe closures. The serialisation concept used in PAEAN is a breadth-first traversal that is designed to limit the amount of communicated graph to a single efficiently-transmitted message. This was first used in Trinder *et al.* (1995) as a modification of the original Graph for PVM mechanism (Loidl & Hammond, 1994; Hammond, 1993) that packed entire sub-graphs. Loidl & Hammond (1996) studies the best strategies for efficiently packing shared program graph. We have followed this work here.

3.3.1 Closure layout and serialisation

Most GHC closures are laid out in the heap as a header section containing the key meta-data, followed by any pointers to other closures, and then by all the non-pointers. The first header field is an *info pointer*, which points to a record containing information such as the number of pointers and non-pointers for a particular heap

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 5. *PAEAN runtime system routines for serialisation*

PAEAN routine and parameters	Description
`rtsPackBuffer* packGraph(StgClosure *la)`	serialise a sub-graph rooted at `la`
`StgClosure* unpackGraph(rtsPackBuffer *buf)`	de-serialise a buffer, returning a local address

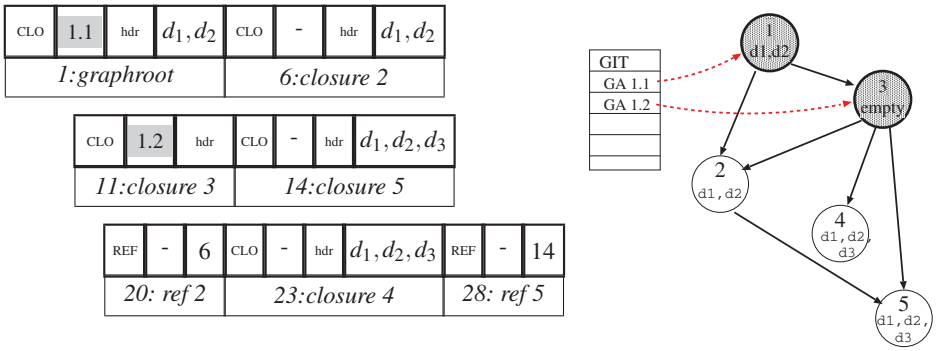


Fig. 12. Example serialisation of the five-closure graph from Figure 11 with two shared closures. The packet on the left encodes the graph on the right, using REF tags to describe sharing.

closure, plus a method that can be used to evaluate the closure. During packing, the Haskell program graph is traversed and serialised so that it can be written into the communication packet. Every unique closure in the packet is identified using a *CLO* tag, followed by the GA of the packed closure. The header and all non-pointer data is packed directly into the packet immediately after the GA. Pointers into the local instance heap will, however, need to be re-established once the program graph has been successfully transmitted to the remote heap. They are therefore omitted from the packet. The order in which the graph is reconstructed allows them to be correctly re-established. Where a packed graph contains shared closures, explicit *REF* tags are used to preserve sharing. They also allow cyclic structures to be transmitted. Finally, top-level program constants (*constant applicative forms*) do not need to be transmitted between PAEAN instances, since each instance maintains its own identical collection of such constants, embedded in the binary program that is executed by all instances. These are identified using a *CAF* tag. Clearly, the graph referenced from the root might be larger than could fit into a single fixed-size packet. In this case, PAEAN will either use *FetchMe* closures to reference the additional heap, or will raise an error.

A few special types of GHC closures cannot be packed using the mechanism described here. These include primitive GHC data structures for synchronisation: Haskell mutable variables, *MVars*, and transactional variables that used to implement software transactional memory, *STM*, as well as some system-level data structures.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 6. *PAEAN runtime system communication routines*

PAEAN Routine	Parameters	Description
`rtsBool MP_start`	Program arguments	Start/synchronise all nodes
`rtsBool MP_sync`	Program arguments	Start/synchronise all nodes
`rtsBool MP_quit`	Error code	Disconnect from system
`rtsBool MP_send`	Receiver, tag, data	(non-blocking) data sending
`int MP_recv`	Destination buffer	(blocking) data reception
`rtsBool MP_probe`	Void	(non-blocking) probe for available messages

Packing these nodes would not be sensible since they encode state that is local to a specific PAEAN instance, such as operating system file handles.

3.3.2 Example: Packing/unpacking graph

Figure 12 shows how the graph from Figure 11 can be serialised into a single packet and transferred between PAEAN instances. Here, closures 2 and 5 are shared using *REF* tags. The `packGraph` function traverses the graph in a breadth-first manner. This means that if the complete sub-graph cannot be packed, then a closure will generally be packed with all its children. In the example, closures 1 and 3 are thunks, and are therefore replaced by *FetchMe* closures in the sender's heap. The `packGraph` method assigns temporary GA, `GA 1.1` and `GA 1.2`, to the two thunks and includes them in the serialised structure. During de-serialisation, these temporary addresses will be replaced by the final addresses that are assigned by the receiver, `GA 2.1` and `GA 2.2`. The same serialisation process can also be used without taking advantage of the PAEAN virtual shared heap mechanism. In this case, the GA fields are omitted and no GIT table is maintained by the receiving instance. In place of GA, global references are established by *communication channels* connected to heap cells. These can be created from within Haskell source programs using the `createC` primitive operation (Section 3.4.3).

3.4 The PAEAN communication subsystem

PAEAN provides a number of communication primitives that allow instances to communicate using asynchronous messages. The RTS implementation provides event-handling routines that allow messages to be received at certain safe execution points, and that link to one of several low-level transport libraries.

3.4.1 Low-level communication primitives

The PAEAN design is based on simple asynchronous point-to-point message passing communication, plus a mechanism for structured startup and shutdown of PAEAN instances, where one PAEAN instance acts as the *main instance*. As shown in Table 6,

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 7. *PAEAN message tags and protocols*

Message tag		Meaning/effect
System Sync:	`READY`	Sync. message from non-main to main instance
	`PETIDS`	Sync. message from main to non-main instances
	`FINISH`	Shutdown signal (from one non-main instance to main instance, or broadcast from main to all non-main instances)
	`FAIL`	Failure indicator (middleware to main instance)
Data transfer:	`CONNECT`	Receiver registers a sender for an inport
	`DATA`	Receiving instance replaces heap placeholder by arriving data
	`STREAM`	Receiving instance modifies heap placeholder, adding arriving data into a list
	`FETCH`	Receiving instance sends back requested data (identified by a global address)
Global Addresses:	`ACK`	Sender acknowledges reception of thunks by communicating new GAs, receiver creates FetchMe closures with GAs
	`NACK`	Sender indicates failure to receive thunks, receiver will revert thunk
Process control:	`RFORK`	Receiver instance creates a new Haskell process
	`TERMINATE`	Receiving instance terminates a Haskell thread (identified by thread ID)
Work distribution:	`FISH`	Sender requests work, receiver will forward fish or answer with `SCHEDULE`
	`SCHEDULE`	Receiver will create new thread to evaluate data (answered by `ACK` or `NACK`)

message passing is implemented by `MP_send` (non-blocking send) and `MP_recv` (blocking receive). These are complemented by the (non-blocking) `MP_probe`, which tests whether messages are available to receive. `MP_start` and `MP_sync` start and synchronise PAEAN instances. `MP_quit` performs a controlled shutdown of the system. Different middleware can be used to implement this communication sub-system. Existing implementations can use both PVM (Geist, 2011) and MPI (MPI Forum, 2012). The shared-nothing message-passing mechanism can also be implemented on top of physically shared memory. The Eden implementation supports POSIX shared memory, Windows shared memory, and Windows Mailslots.

3.4.2 High-level messages

A high-level set of messages is built on this low-level implementation. Table 7 summarises all the PAEAN message types and their semantics. Many kinds of messages are sent internally by the RTS itself: *system* messages are exchanged during startup and shutdown to synchronise and coordinate instances; *data transfer*

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Table 8. *PAEAN primitive operations (callable from Haskell)*

data `Mode` = `Stream` | `Data`	Data modes: Stream or Single data
| `Connect` | `Instantiate` **Int**	Special modes: Connection, Instantiation
data `ChanName'` = `Chan` **Int#** **Int#** **Int#**	A single channel
`createC` :: *IO (ChanName' a, a)*	Create a channel and placeholder
`connectToPort` :: *ChanName' a → IO ( )*	Receiver registration
`sendData` :: *Mode → a → IO ( )*	Send data to registered receiver
`fork` :: *IO ( ) → IO ( )*	New thread in same process (Conc.Haskell)
`noPe` :: *IO Int*	Number of instances
`selfPe` :: *IO Int*	ID of own instance (1..N)

messages (Section 3.2.2) request globalised data (`FETCH`); *GA* messages `ACK`nowledge a new GA or indicate failures (`NACK`); and *work distribution* messages (Section 3.5) search for work (`FISH`) and pass on new work (`SCHEDULE`).

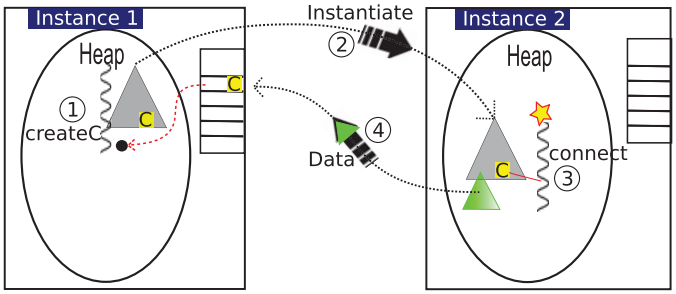
3.4.3 Explicit communication via channels

PAEAN also provides facilities to enable explicit communication and process control directly from within Haskell, using a number of new *primitive operations* (Table 8). These primitives may be preferred where a full shared heap is not required, and allow direct communication between two instances, forming a dual to the *FetchMe* closures described above. EdI uses the primitives directly as a simple explicit language for parallel coordination. Figure 13 shows a simple example of using the explicit communication primitives, which provide functionality for *creating* and *using* typed channels, as well as *instantiating* computations on nodes.

3.5 Work distribution through offloading and work stealing

In a shared-nothing system, work will need to be moved from instances that have excessive amounts of work to those that are idle. When and how this is done is an important policy choice that can significantly affect performance. There are two basic approaches: work can be offloaded *eagerly* at the time of creation (active load distribution); or it can be offloaded *lazily*, e.g. in response to changes in system load (passive load distribution). Eager work offloading aims to quickly saturate the parallel system. This is beneficial in cases of regularly structured parallelism where a single main task generates all the available parallelism at the start of the execution. Such a policy is easy to implement using PAEAN's remote execution mechanisms, as used by Eden. Typically, under such an approach, all the data that a thread requires will also be sent with the thread when it is offloaded. This may, of course, add significant delay before a thread can start. When used with virtual shared memory, it also runs the risk of significant heap fragmentation and increased communication costs. In contrast, lazy work offloading is typically implemented by

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press




```
-- parent process
do -- create reply channel (1)
   (ch,bkh) ← createC
   -- spawn remote process (2)
   sendData (Instantiate 2)
                (thread2 ch)
   -- next function call will
   -- block until response arrives
   print bkh
```

```
-- child process

-- thread2 ch =
   do connectToPort ch -- (3)
      let dat = fib 42
      rnf dat `pseq`    -- (4)
          sendData Data dat
-- terminated
```

Fig. 13. Communicating computations and values explicitly. A thread on Instance 1 creates a channel (1) and passes the created `ChanName`' to a new process on Instance 2 (2). A newly created thread `connects` to the channel (3) and sends evaluated `Data` (4).

```
-- insert a closure into a local HEC's spark pool:
par# :: a → b → b
-- as above, but with locality control, specifying min and max distances:
parDist# :: Int → Int → a → b → b
```

Fig. 14. Spark insertion primitives in GHC-SMP and PAEAN.

some kind of *work stealing* mechanism, where idle instances attempt to obtain work from busy instances. PAEAN provides flexible support for distributing work lazily across a shared-nothing system by building on, and extending, the basic GHC-SMP spark/thread mechanisms (Figure 14). The serialisation mechanism described in Section 3.3 is used to offload a *spark* plus some or all of the data that is associated with it. The PAEAN work stealing mechanism supports stealing sparks, which are fairly cheap to transfer, rather than threads, which may carry significant local state.

3.5.1 Example of work stealing

Figure 15 illustrates this work-stealing load-balancing mechanism. When **Instance 2** becomes idle, it sends a work request message (or FISH) to another processor. In this case, **Instance 1** is targeted. Since **Instance 1** has some sparks in its spark pool, it responds by sending one of these sparks to **Instance 2** in a SCHEDULE message, serialising the graph that it refers to, as described in Section 3.3, and replacing the original sparked closure by a *FetchMe*. The program graph will be de-serialised when

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

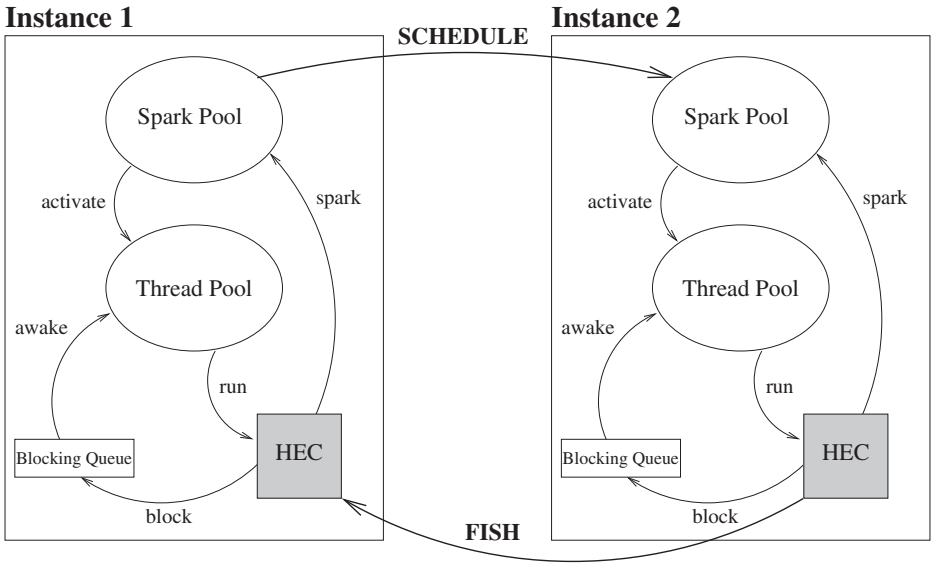


Fig. 15. Work stealing using PAEAN's spark-based mechanisms. An idle instance (Instance 2) sends a FISH message to Instance 1, aiming to steal some available work. If Instance 1 has any sparks, then the spark and any associated graph will be serialised and sent to Instance 2 in a SCHEDULE message.

it is received on **Instance 2**. The spark is added to its local spark pool, whence it can be turned into a thread using the usual thread scheduling mechanisms, or even passed on to another PAEAN instance if **Instance 2** has gained sufficient work[6]. If a targeted instance has no sparks, it forwards the FISH message a fixed number of times. If no instance has any sparks, the FISH message is sent back to the originator. A back-off delay is introduced, and a new FISH message will be sent (Trinder *et al.*, 1995).

Typically, execution of the thread that created the spark on **Instance 1** depends on the result of this spark that has been moved to **Instance 2**, and other threads might also depend on it. When execution requires the exported spark, the thread will be added to a local queue of blocked threads. The *FetchMe* will then be followed, and a FETCH message will be sent to **Instance 2**. If the spark has been evaluated by **Instance 2**, its (weak head) normal form will be serialised and returned to **Instance 1** in a DATA message. Otherwise, **Instance 2** will record the attempt to fetch the closure in its blocking queue, so that its (weak head) normal form can be returned to **Instance 1** once it becomes available.

3.5.2 Work distribution mechanisms for hierarchical systems

We have made several enhancements to our work distribution mechanism in order to improve performance for large-scale systems, which typically involve *clusters* of

[6] This helps avoid starvation. It would be possible, but highly unlikely, for a spark to be transferred between the same instances multiple times, but only where the instances gain sufficient work elsewhere. Although some effort would then be wasted, progress would have been made, so termination is ensured.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

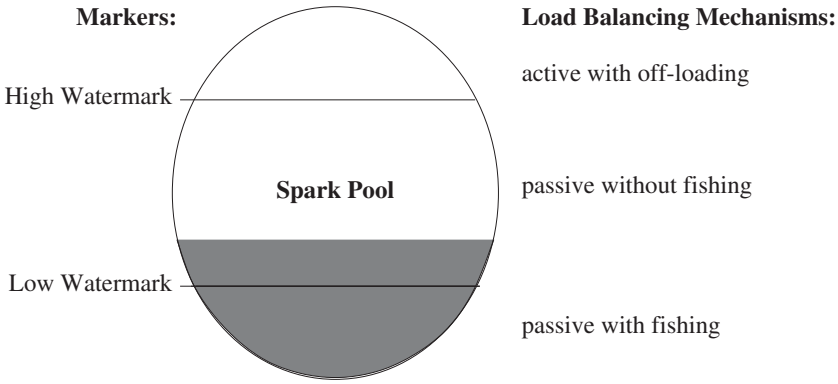


Fig. 16. Low- and high-watermark mechanisms for spark distribution. (showing a single spark pool). By default work-stealing (passive) load distribution is used. If the size exceeds the higher-watermark, work is actively offloaded. The system aims to always fill the pool up to the low-watermark.

multicore processor nodes connected using high-latency networks. Such systems are usually hierarchical with widely varying inter-node latencies, i.e. they are genuinely NUMA. The largest system we have studied is Edinburgh Parallel Computer Centre's HECToR, which has over 90,000 cores and a complex hardware interconnect. Section 4 shows scalability results from a smaller, but still realistic, hierarchical cluster.

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 16. 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 instance will try to obtain additional sparks from other instances. 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 use SCHEDULE messages to actively offload sparks to other instances without being asked for work. 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. If all instances have large numbers of sparks, a back-off mechanism is used to delay SCHEDULE messages, as described above for FISH messages. Our experiments identify this mechanism as the most important one for tuning cluster performance (Aljabri *et al*., 2013). Measurements on a 32-node cluster with up to 100 cores show a reduction in runtime on a set of micro-benchmarks of up to 57%, i.e. in the best case, we can gain a speedup of more than two over the standard FISHing policy that we described above. For larger benchmarks, improvements range from 16% to 28%. Further improvements are possible, by dynamically adapting the low-watermark over the runtime of the program. In more recent work (Aljabri, 2015) (Section 5.4), we have extended the scalability measurements to a combination of two remote

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

clusters with 300 cores in total. Using our `sumEuler` example, we have obtained speedups of up to 140 on this more heterogeneous, distributed architecture.

Locality control: Controlling locality is important for ensuring good performance on hierarchical shared-nothing systems, especially those with high latency connections (Loidl, 1998; Loidl, 2001). PAEAN provides several enhanced primitives to support locality control. The most powerful of these primitives is `parDist` (Aswad, 2012) (Figure 14), which takes two additional integer arguments, specifying a lower and an upper limit on the distance of work distribution from the current instance. If these are both 0, then this forces local evaluation. Setting a minimum limit forces work to be distributed through the system. This is useful in low-load situations, such as system startup.

4 Evaluation of scalability and performance

We have carried out a series of experiments to assess the scalability and performance of our GpH, Eden and EdI implementations using the versions of `sumEuler` from Section 2. Our experiments used a commodity cluster of 32 nodes, each with an 8-core x86_64 CPU[7]. The cluster therefore provides a total of 256 processor cores. Since the full machine was not available for our exclusive use, the measurements reported here are limited to a maximum of 64 cores on 16 nodes (i.e. using at most four cores per node). We delegate the *placement* of PAEAN instances to the communication middleware. Both MPI and PVM prefer using different nodes to using different cores on the same node. This balances the load at the cost of increased network communication. For our experiments, we have mainly used PVM, but, where indicated, some of the EdI results have also used MPI. The implementations used in our measurements are the Eden/EdI system version 7.8.3 (mid 2014), and a research version of GpH on clusters, GUM-SMP, based on GHC 6.12.3.

4.1 Measured speedup

Figure 17 shows measured speedups for each `sumEuler` variant, with $n = 90,000$. We can clearly see that the sumEuler program gives near-linear speedup for all three implementations, even when large numbers of cores are used. Overall, the EdI implementation shows the best speedups. The MPI implementation is slightly better than PVM. The Eden version comes close to EdI. However, since the GpH version requires more management (i.e. overhead), it delivers worse speedup. It does, however, scale robustly (we have confirmed this up to 180 cores). The difference between the EdI and the Eden versions can be explained by differences in communication behaviour: while the Eden version communicates the full list of arguments to each child process as a stream, the EdI version requires the child

[7] The beowulf cluster at Heriot-Watt University Edinburgh. Each node is built from two quad-core Xeon E5506 2.13GHz, with 256kB L2 and 4 MB shared L3 cache, connected by gigabit Ethernet, and running CentOS 6.4.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

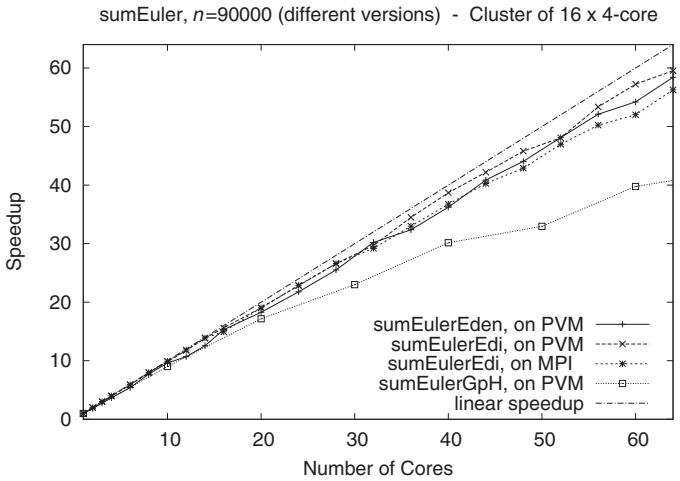


Fig. 17. Speedup of `sumEuler` variants: $n = 90,000$.

processes to generate their arguments from parameters (size, offset, stride), and communicates only the results. This reduces communication costs. The GpH version suffers in this example due to the higher overheads associated with managing the virtual shared heap. Since the work is already well distributed through the structure of the problem, the automatic load balancing mechanism does not improve performance in this case. It would, however, deal better with situations where some nodes were executing external workloads, and were not dedicated to the Haskell program. Overall, the `sumEuler` example clearly demonstrates the ease of parallelisation and simplicity that is claimed for parallel functional programming. It was straightforward to achieve the near-perfect speedup reported here, since the problem structure itself is intuitive and easy to parallelise. Note that the speedup we obtain is dependent on the problem size. For smaller problem sizes, where $n < 90,000$, it would scale less well with increasing number of cores, since the constant overhead of instantiating the program and using the middleware would have an increasing effect.

4.2 Scalability

While the *sumEuler* example is easy to decompose into independent parallel tasks, many parallel algorithms have more complex structures with inter-task dependencies. One example is the *N-Body* simulation, which simulates the movement and gravitational acceleration of a number of "particles" (physical bodies) in a 3-dimensional space. The simulation is typically approximated iteratively in a number of discrete time-steps. A simple straightforward parallelisation of this problem will exploit parallelism within each iteration, assigning subsets of particles to different threads, where each thread computes the new velocity and position for its own particles. In order to update the velocities for its own particles, *each thread* needs the position and mass of *all* the other particles. This is approximated by exchanging new information after each iteration, in a potentially costly global exchange of data.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

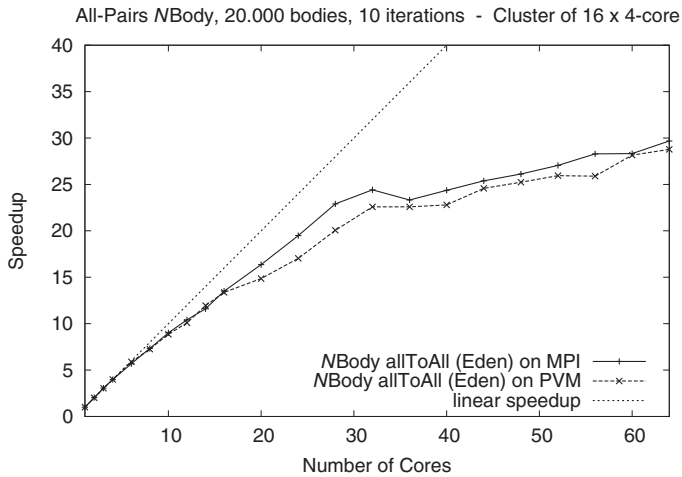


Fig. 18. NBody (Eden): 20,000 particles, 10 iterations.

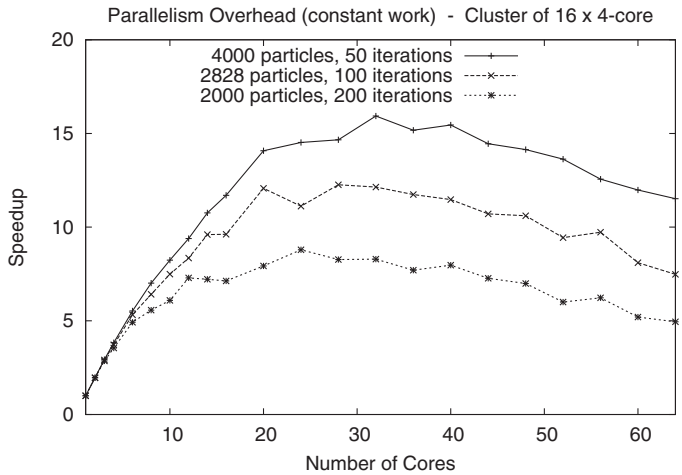


Fig. 19. NBody (Eden) on MPI: Overhead increasing with iteration count.

We have measured the performance of an Eden implementation of N-Body based on *iteration skeletons* (Dieterle *et al.*, 2013). These use the `allGather` skeleton (Dieterle *et al.*, 2010) to implement the global data exchange. Figure 18 shows the speedups that we obtain with this implementation for a problem size of 20,000 particles. For this problem size, the program scales reasonably well, achieving a maximum speedup of around 30 on 64 cores. Performance degrades when more than two cores per node are used. Once again, the MPI back-end performs similarly to, but slightly better than, its PVM counterpart.

The iterative structure of this algorithm means that each iteration is performed sequentially. Figure 19 shows the *overhead* when the algorithm performs more iterations while the amount of computation is kept constant ($N^2 \cdot k$ computations for N particles and k iterations). We can see that speedup degrades earlier and is

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

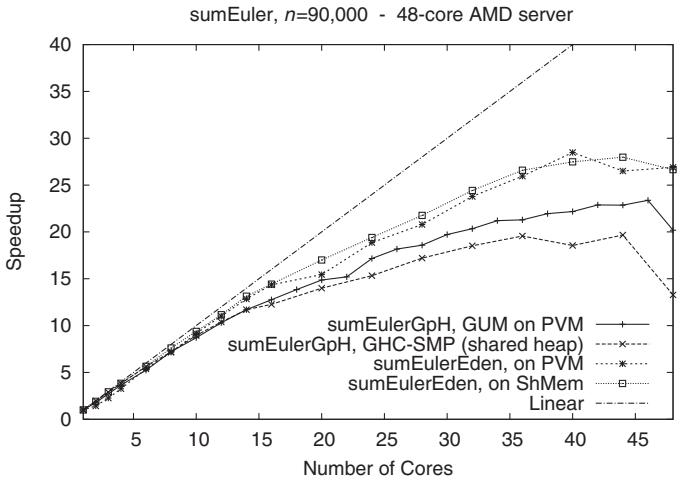


Fig. 20. Speedup of `sumEuler` variants on an 48-core machine: $n = 90,000$.

generally lower when more iterations are performed, since each iteration requires a global synchronisation.

4.3 Shared-nothing performance on shared-memory systems

As a baseline for our shared-memory results, we again use the *sumEuler* program on a 48-core machine[8] to compare the performance of the GpH and Eden implementations. Figure 20 shows the results that we obtain. While the speedups are fairly modest for this simple parallel program, we observe a steady increase in speedup for all versions, up to 28 on 46 cores in the best case for Eden with a shared-memory (ShMem) back-end. All implementation show a drop in speedup when using all 48 cores. This is consistent with previous experimental results on similar systems, where we have found that allocating all available processors to the parallel execution can interfere with system process execution. As with the cluster results, the GpH speedups are slightly worse than those for Eden, but notably better than those for the GHC-SMP shared-memory implementation. This matches with our previous study of the impact of NUMA architectures on parallel Haskell performance (Aljabri *et al.*, 2014). We summarise our findings below. These findings are also consistent with parallel benchmarks, across languages, on this NUMA architecture. Due to the different memory latencies to different memory banks in NUMA architectures, even simple parallel programs struggle to efficiently exploit this kind of architectures.

For a more detailed evaluation of the performance of the PAEAN shared-nothing approach on a physically shared-memory system, we have chosen to use

[8] Four AMD Opteron-based processors, each with two NUMA regions, where each region has six 2.8GHz cores; total RAM is 512 GB, with 64 GB per region; there is a 2 MB L2 cache, shared between every two cores in each region, and a globally shared 6 MB L3 cache.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

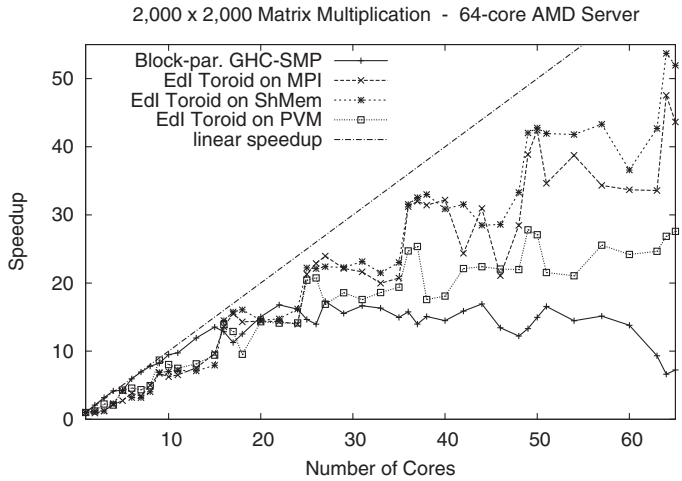


Fig. 21. Matrix multiplication (2000 × 2000) comparing to GHC-SMP.

the `MatMult` benchmark program from the `nofib` suite[9]. Matrix multiplication can be easily decomposed, with independent calculations, so should give good scalability, in principle, provided that the thread granularity is managed carefully. We compared the scalability of the original GHC-SMP implementation against that of an equivalent Eden version. The Eden version uses a variant of Cannon's algorithm, communicating matrix blocks in a toroidal topology, using a recursive toroid skeleton (Berthold & Loogen, 2005). The GHC-SMP version uses a similar blockwise task distribution, but tasks communicate via the shared heap without using a specific communication topology. Hence, the GHC-SMP version can freely choose the block size to improve load balancing, but might encounter bus contention when doing so. Since we are evaluating scalability on shared-memory architectures, we have used a 64-core physically shared-memory system[10]. Our speedup results are shown in Figure 21. They clearly show that the Eden version can exploit the available cores, delivering a peak speedup of approximately 54 on 64 cores. The GHC-SMP shared-memory version delivers the best speedups on small numbers of cores, but flattens out at about $16 - 17\times$. The Eden versions scale better when using more than 25 cores; all PAEAN shared-nothing implementations have speedup that is equivalent to or better than GHC-SMP from that size onwards. This is probably an effect of shared-memory GC. Note that the speedup for the Eden versions does not scale linearly with the number of cores. The toroidal algorithm delivers a "stepped" speedup profile, that is ideally matched to a square number of cores. The Eden program attempts to utilise all the cores by placing more than one process on some of them when approaching square numbers. This however, hardly yields any better speedup.

[9] http://git.haskell.org/nofib.git

[10] Compute server *lovelace*, University of St.Andrews: 4 × 16 core Opteron 6376 1.4GHz with 2 MB L2-cache (per core), 16 MB shared L3 cache, 640 GB shared RAM.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

We note, in passing, two interesting phenomena: First, the GHC-SMP version is faster than the equivalent sequential version without any spark creation when executed on only one core. However, it uses a considerably larger heap. Creating sparks reserves heap space early, and therefore accelerates execution. This effect is intensified when using column-wise instead of block-shaped sparks: column division exposes the worst access locality. On more cores, this effect quickly disappears, and the spark strategies only have a minor influence on the runtimes. Second, the Eden version has been measured with all three possible back-ends for data transfer: the shared-memory back-end, MPI and PVM. As expected, PVM performs worse than MPI, and the shared-memory back-end yields the best performance. This exposes the overhead that is imposed by PAEAN communication subsystem middleware: the bespoke shared-memory back-end was tailor-made for its specific message-passing needs, without any additional features which would worsen performance.

Finally, we have recently performed a systematic evaluation of shared-nothing versus shared-memory parallel Haskell systems on a number of modern NUMA architectures (Aljabri *et al.*, 2014). We have explored a range of configurations, including purely distributed memory, classic GUM; a mixture of shared-memory and distributed-memory, GUM-SMP (Aljabri *et al.*, 2013; Aljabri, 2015) and purely shared-memory, GHC-SMP. Our measurements show that the shared-memory implementation consistently performs worst on these architectures, repeatedly hitting memory access and locking bottlenecks. A pure distributed-memory implementation improves runtimes by a factor of about three compared with the shared memory implementation. Further improvements can be achieved by configuring the system to use one shared heap for each NUMA region on the machine. These results support the findings reported above, underlining that even on the current generation of moderately parallel systems, a shared-nothing design delivers consistent and tangible performance benefits.

5 Related work

5.1 Built-in parallelism support in GHC: GHC-SMP

The standard GHC RTS provides built-in support for parallelism (Marlow *et al.*, 2009), but only on physically shared-memory systems. Drawing on experience with e.g. the GUM design for GpH (Trinder *et al.*, 1995), GHC-SMP supports both the GpH and Par-Monad models of parallelism. It uses a physically shared heap with a common address space. Synchronisation is achieved using a "black-holing" mechanism. Memory management uses a shared-memory implementation of GHC's standard generational garbage collector, with independent nursery areas, but shared old generations. GC is currently *stop-and-copy*: all threads must be suspended during GC. While there has been some work on independent GC (Marlow & Peyton Jones, 2011), the complexity of supporting this efficiently and effectively on a physically shared-memory architecture, without adopting a shared-nothing implementation, means that this has not yet been integrated into the production version of GHC.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

5.2 Implementations of GpH and Eden

The first implementation of GpH was designed to work on the novel GRIP multiprocessor (Peyton Jones *et al.*, 1987; Hammond & Peyton Jones, 1990; Hammond & Peyton Jones, 1992). This was first adapted to yield a message-passing implementation of virtual-shared memory using PVM, GRAPH for PVM (Loidl & Hammond, 1994) and subsequently to both physically shared-memory and shared-nothing parallel machines in the form of GUM (Trinder *et al.*, 1995). GUM uses a two-level message-passing communication system, where high-level messages for synchronisation, workload distribution etc. are built on a basic communication system. Low-level communication can be realised either via shared-memory or via an explicit message-passing communication library such as PVM or MPI. This gives significant flexibility. GUM has been extended using a variety of workload management systems (Loidl & Hammond, 1996; Loidl, 2001; Du Bois *et al.*, 2002; Al Zain *et al.*, 2008; Aljabri *et al.*, 2012). Using these mechanisms, it has been deployed on a variety of systems ranging from small-scale multicores to hierarchical clusters of up to 90,000 cores (Maier *et al.*, 2014a).

The DREAM RTS (Breitinger *et al.*, 1998; Klusik *et al.*, 1999) was the first working implementation of Eden. It was originally implemented by adapting the GUM implementation to comply with the the DREAM abstract machine model (Breitinger *et al.*, 1997). It has subsequently seen several complete rewrites and major revisions. Some parts of the implementation which were originally implemented directly in the RTS have now been lifted to the Haskell level, relying on simpler and more modular RTS support (Berthold *et al.*, 2003). This refinement ultimately led to factoring out the EdI language: all Eden language constructs are now implemented in Haskell, using the simpler primitive operations that implement EdI directly (Berthold & Loogen, 2007; Berthold, 2008).

5.3 Distributed systems: Cloud Haskell and HdpH

A contrasting approach to that described here is taken by Cloud Haskell (Epstein *et al.*, 2011) and HdpH (Maier *et al.*, 2014b). Rather than providing flexible and general RTS support, which can be used to build appropriate high-level parallel programming constructs for different language designs, Cloud Haskell and HdpH both use a fully explicit approach to parallelism on distributed memory platforms, relying on the programmer to build any necessary higher-level abstractions. Cloud Haskell provides a minimal abstraction level, while HdpH also includes ideas of work stealing and virtual sharing that were previously developed for GpH. As with Eden or GpH, higher-level parallel abstractions can be built on these primitives, in the form of structured parallel approaches such as *algorithmic skeletons* (Cole, 1989) or *evaluation strategies* (Trinder *et al.*, 1998). In our opinion, this is likely to be the best way for functional applications programmers to use libraries such as Cloud Haskell or HdpH, since many problematic details can then be encapsulated into higher level abstractions. This is also consistent with the functional programming philosophy of good abstraction and encapsulation.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

5.4 Work distribution for large-scale hierarchical PAEAN systems

The basic PAEAN work distribution model (Section 3.5) has been extended to provide better support for large-scale, hierarchical architectures. Grid-GUM2 (Al Zain *et al*., 2008) systematically attaches load information to *FISH* messages in order to refine the search for available work. Unlike the standard PAEAN mechanism, it delivers a variable number of work items in order to amortise the cost of transferring work and data between sub-clusters in a Grid infrastructure. Grid-GUM2 also supports migration of live threads (Du Bois *et al*., 2002): threads are serialised in their current execution state, and can be migrated across processors in order to improve load balance. Migration should, however, be used carefully since it is expensive and increases heap fragmentation. Aljabri *et al*. (2013), Aljabri (2015) describe GUM-SMP, which combines our shared-nothing work-stealing mechanisms with those in the standard GHC-SMP. This gives additional freedom in configuring PAEAN systems for large-scale clusters of multicore machines.

5.5 Scheduling support in Haskell

Both the Par-monad (Marlow *et al*., 2011) and Meta-par (Foltzer *et al*., 2012) have adopted the idea of defining parallel scheduling policies in Haskell. However, these approaches do not consider more general RTS support and system architecture, as we have done in this paper. Moreover, unlike our approach, the Par-monad is restricted to shared memory parallelism. Two separate lightweight implementations of concurrent Haskell have also been produced that lift scheduling and other concurrency features to the Haskell level (Li *et al*., 2007; Sivaramakrishnan *et al*., 2013). This complements our work, that instead focuses on the introduction and control of parallelism. It would, however, be both straightforward and useful to incorporate similar concurrency features into PAEAN, and we intend to investigate this in future work. Similar support for scheduling operating system level threads and processes has been provided in the prototype House operating system (Hallgren *et al*., 2005) , which was written entirely in Haskell.

5.6 The ArTCoP micro-kernel

ArTCoP [11] is a highly modular micro-kernel design for parallel Haskell (Berthold *et al*., 2008) that is closely related to PAEAN. The *kernel level* implements simple and generic routines for asynchronous communication, basic execution management and system information. *System Modules* written in Haskell restrict and combine these kernel routines to provide higher level coordination constructs, narrowing the generic runtime support to a particular programming model. Coordination constructs can be used at both a *Library* and an *Application level* either to implement algorithms directly or to implement parallel algorithmic patterns and skeletons (Cole, 1989). System modularity is achieved similarly to PAEAN components: driven by

[11] ARchitecture-Transparent Control Of Parallelism

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

```
data Serialized a = Serialized { packetData :: ByteArray# }

serialize   :: a → IO ( Serialized  a)
deserialize :: Serialized  a → IO a

-- | Packing and unpacking  exceptions :
data PackException
    = P_BLACKHOLE | P_NOBUFFER | P_CANNOTPACK  | ... | P_GARBLED  -- from RTS
    | P_ParseError | P_BinaryMismatch | P_TypeMismatch          -- from Haskell

-- Serialized  instances :
instance Typeable a ⇒ Binary (Serialized a) ...
instance Typeable a ⇒ Show (Serialized a) ...
instance Typeable a ⇒ Read (Serialized a) ... -- read ∘ show  == id
```

Fig. 22. Proposed Haskell types for serialised data.

a *workload distribution logic*, tasks are distributed to processors and controlled by a *scheduling* component; scheduling relies on a *communication* subsystem; and a *monitoring* component informs dynamic and adaptive scheduling decisions. In Berthold (2008), Berthold *et al.* (2008), we presented a Haskell scheduling framework in the spirit of the ArTCoP design. Complex scheduling policies and heuristics can be defined in Haskell, based on a system of type classes for parallel jobs and scheduler states. The scheduling framework can express complex mechanisms such as the FISH mechanism discussed in Section 3 and Grid-GUM2.

5.7 Haskell-level serialisation for PAEAN

The PAEAN serialisation and de-serialisation routines allow data to be exchanged between different heaps in a running system. In doing so, however, they also provide type- and evaluation-agnostic data persistence for Haskell. This has several important uses (Berthold, 2011), including supporting persistence for memoised functions and checkpointing. Figure 22 shows how Haskell-level serialisation can be realised, supported by PAEAN primitive operations. Serialisation is *orthogonal to evaluation*: any Haskell program graph can be serialised. It is also *orthogonal to Haskell types*: as shown, type-safety can be re-established by phantom types. When `Serialized` data is externalised (written to files or sent over the network), phantom type information is lost and must be recovered dynamically, using `Typeable`. This is achieved using the representations provided by the `Binary`, `Show` and `Read` instances. To make our work usable in other settings, e.g. networked systems, we have factored out this serialisation functionality, so that it can be installed as a Haskell library[12] using *cabal*. This library is a good example of the targeted bespoke RTS support for parallelism that we advocate for PAEAN: well-tailored operations enable a simple and straightforward API and avoid the pervasive complications of library-

[12] Available at `http://github.com/jberthold/packman/` and on hackage.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

only solutions to Haskell data communication, as typified by e.g. Cloud Haskell or HdpH.

5.8 Kali scheme

As with ARTCOP, Kali Scheme (Cejtin *et al.*, 1995) aimed to provide user-level scheduling operations, including load balancing and thread migration. Like PAEAN, Kali Scheme is a shared-nothing implementation, but for Scheme rather than Haskell. Kali Scheme operates at a similar level to EdI, HdpH or Cloud Haskell, using explicit message-passing communication. Unlike the PAEAN approach, but like Cloud Haskell and HdpH, it provides explicit primitives to create and manage independent heap spaces, including serialisation and deserialisation for transmitting closures. Kali Scheme assigns *unique identifiers* to *all* heap objects, and uses a mechanism similar to our *FetchMe* to fetch all data referenced by unique identifier in a received message before taking any action on it. The PAEAN system, in contrast, assigns GA only to thunks that were *actually* communicated before being evaluated, and lazily fetches other references only when the data is required. Kali Scheme also provides a proxy mechanism, which allows remote values to be accessed using Kali's strong code mobility capabilities. As with GUM and PAEAN, Kali Scheme supports a two-level GC scheme, where local collections can proceed independently. Global collection uses a combination of lightweight distributed reference counting with a separate mechanism for global cycle detection. A similar mechanism to our ACK/NACK messages is used to ensure that messages are not lost in transit, and that in-flight data is not garbage collected prematurely. At the time of writing, the Kali Scheme system seems to be no longer maintained. However, there are calls to revive the project.

5.9 Manticore

Finally, while it does not aim to provide a reusable library as with PAEAN, the Manticore (Fluet *et al.*, 2007; Fluet *et al.*, 2010) implementation for parallel ML provides similar implicitly threaded parallelism and follows the same spirit of semi-explicit adaptivity as our earlier implementations of GpH. Futures and data parallel constructs are used to manage local parallelism. Explicit synchronisation and coordination is used at a larger scale (Reppy *et al.*, 2009). At the runtime level, Manticore provides a notion of *virtual processor* similar to a HEC, which can be similarly mapped to physical cores by the operating system. A process, comprising multiple lightweight threads, is mapped to the required number of virtual processors using a `provision` operation. There is a simple built-in mechanism for basic load-balancing, extended using a low-latency work-stealing mechanism (Acar *et al.*, 2013). Manticore also provides a low-level internal scheduling interface (Acar *et al.*, 2012), which provides preemption and interrupt via a signal-based approach. Processes are scheduled in a round-robin fashion, and schedulers may be nested, so that, for example, a data parallel scheduler may be used within a process-level scheduler.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

6 Conclusions

This paper has introduced PAEAN, an RTS framework that supports efficient and scalable execution of parallel Haskell programs on a variety of (possibly NUMA) parallel architectures, and shown how it can be used to implement three distinct parallel Haskell dialects: GpH, Eden and EdI. PAEAN is based on the widely used and heavily tuned GHC RTS. It offers a flexible and general API for workload distribution, virtual shared memory, communication and serialisation that can be used to implement a variety of further Haskell dialects, on a "pick-and-mix" basis.

The central design decision in PAEAN is its shared-nothing heap, which greatly enhances the scalability of the system. PAEAN abstracts over low-level communication libraries, linking to standards such as PVM and MPI. We believe that with hardware scalability limitations imposed by upcoming many-core architectures, such a design is the only way forward for modern, massively parallel systems. These concerns are already visible in modern NUMA hardware architectures, and threaten to disrupt current programmer models of memory behaviour. It is no longer tenable to assume that access to an array is uniform cost, for example.

This provides an opportunity for more structured and flexible approaches to data as promoted by Haskell and other functional languages, that will better fit limited sharing or shared-nothing settings. In order to achieve this potential, it is important to provide good abstractions over this rapidly increasing complexity at the hardware memory level. We have built a programmer level virtual shared heap abstraction on top of a set of low-level message passing primitives. PAEAN also supports physically shared memory, but as a special case of our shared-nothing design. Our performance results show that we can obtain good, scalable and easy-to-use parallelism for multiple Haskell dialects beyond the usual shared-memory limitations of standard parallel Haskell implementations.

6.1 Limitations and further work

While we have considered reasonably sized clusters in this paper (of up to 64 cores), in order to fully study scalability, we will need to consider the use of PAEAN implementations on larger machines such as EPCC's 90,000 core HECToR. Our results with a challenging symbolic computation application on up to 1,024 cores of HECToR show good scalability for a program with complex data dependencies (Maier *et al.*, 2014a). However, we anticipate that larger, hierarchical systems will highlight the need for improved control of locality, hierarchical workload management, and other mechanisms of work and data distribution that we have discussed above.

One other obvious limitation is that PAEAN does not currently provide support for fault tolerance. This is not a major issue in a typical parallel computing setting (whether shared-something or shared-nothing), since such systems can generally be assumed either to be reliable or to provide systems-level recovery mechanisms. Likewise, cloud and other managed distributed servers typically provide good

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

system-level mechanisms for checkpointing and recovery, which should deal with system-level failures. A language-level mechanism is more useful for dealing with programmer level failures, where the cause of any error can be predicted and corrected by the programmer. In such a situation, the direction taken by Cloud Haskell and HdpH, using Erlang-style supervisor processes to monitor the health of large distributed systems (Stewart *et al.*, 2012), would seem to offer a relative simple and effective mechanism that can easily be added to PAEAN. We intend to investigate this in due course.

Parallel programming is increasingly mainstream, and functional languages like Haskell are, *in principle*, an excellent fit to modern massively parallel architectures. However, the chief advantages of functional programming, namely *composability* and *referential transparency*, can easily be lost, for example, by using explicit communication primitives to link shared-nothing machines. Advanced runtime support such as we provide in PAEAN aims to bridge and structure necessary performance tweaks in a carefully designed and modular RTS. In our opinion, such an approach is crucially important to achieving large-scale parallel programming without losing key semantic advantages of the functional approach. Ultimately, we intend to develop a revised PAEAN system that captures the spirit of our ArTCoP design: with the necessary RTS internals largely programmed in Haskell, while still achieving good scalable performance through flexible RTS support. We anticipate that this will allow us to address both near-term and more distant parallel hardware developments, while providing support for *high-level* parallelism abstractions in Haskell.

Acknowledgments

This work has been partially supported by the European Union grants IST-248828 ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering, IST-288570 ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems, and IST-644235 RePhrase: Refactoring Parallel Heterogeneous Systems, a Software Engineering Approach; by EU COST Action IC1202: Timing Analysis On Code-Level (TACLe); and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP: High Performance Computational Algebra. Jost Berthold was partially supported by the Danish Council for Strategic Research DSF under contract number 10-092299 (Hiperfit). We would like to thank the Scottish SICSA initiative for funding a visit of Jost Berthold to Heriot-Watt and St Andrews Universities, and for supporting numerous events promoting research on (parallel) programming languages and implementations. We would also like to thank Malak Aljabri for her contributions in terms of performance measurements for the GUM-SMP system, as well as all the PhD students and others who have contributed to extending the GUM and Eden systems in many interesting directions.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Systems availability

Source code for the latest version of the GpH runtime system, including GUM and GUM-SMP, is available at `http://www.macs.hw.ac.uk/~hwloidl/hackspace/GUMSMP`. This version covers both GpH and Eden. The full Eden source code, which implements a subset of the PAEAN design, plus pre-compiled binary releases for selected GHC versions, are available for download at `http://www.mathematik.uni-marburg.de/~eden/`. Binaries are provided for Windows, Linux, and Mac OS versions. All Haskell modules are available on *hackage*. RTS source code is available at `http://github.com/jberthold/ghc`.

References

Acar, U. A., Charguéraud, A. & Rainey, M. (2012 January) Efficient primitives for creating and scheduling parallel computations. In *Workshop contribution for DAMP'12*. Available at `http://chargueraud.org/research/2012/damp/damp2012_primitives.pdf`. [Retrieved 14/12/2015]

Acar, U. A., Charguéraud, A. & Rainey, M. (2013) Scheduling parallel programs by work stealing with private deques. *ACM SIGPLAN Notices*, vol. 48. (18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP). ACM, New York, pp. 219–228.

Aditya, S., Arvind, Augustsson, L., Maessen, J.-W. & Nikhil, R. S. (1995) Semantics of pH: A parallel dialect of Haskell. In *Proceedings of the Haskell Workshop*, Hudak, P. (ed), La Jolla, USA, pp. 35–49.

Al Zain, A. D., Trinder, P. W., Michaelson, G. J. & Loidl, H.-W. (2008) Evaluating a high-level parallel language (GpH) for computational GRIDs. *IEEE Trans. Parallel and Distrib. Syst.* **19**(2), 219–233.

Aljabri, M., Trinder, P. & Loidl, H.-W. (2012 August) The design of a GUMSMP: A multilevel parallel Haskell implementation. In Proceedings of IFL'12: 24th Symposium on Implementation and Application of Functional Languages. Oxford, UK (draft proceedings), pp. 75–90.

Aljabri, M., Loidl, H.-W. & Trinder, P. W. (2013) The design and implementation of GUMSMP: A multilevel parallel Haskell implementation. In Proceedings of ACM SIGPLAN Symposium on Implementation and Application of Functional Languages (IFL'13). ACM, New York, pp. 37–48.

Aljabri, M., Loidl, H.-W. & Trinder, P. (2014) Balancing shared and distributed heaps on NUMA architectures. In Proceedings of *TFP'14: Symposium on Trends in Functional Programming*. LNCS 8843. Springer, Berlin/Heidelberg, pp. 1–17.

Aljabri, M. S. (2015 October) *GUMSMP: A Scalable Parallel Haskell Implementation*. Ph.D. thesis, School of Computing Science, University of Glasgow.

Appel, A. W. (1989) Simple generational garbage collection and fast allocation. *Software: Pract. Exp.* **19**(2), 171–183.

Aswad, M. K. (2012 April) *Architecture Aware Parallel Programming in Glasgow Parallel Haskell*. Ph.D. thesis, School of Mathematical and Computer Sciences, Heriot-Watt University.

Berthold, J. (2008 June) *Explicit and Implicit Parallel Functional Programming: Concepts and Implementation*. PhD Thesis, Philipps-Universität Marburg, Germany.

Berthold, J. (2011) Orthogonal serialisation for Haskell. In *IFL'10: Implementation and Application of Functional Languages*, Hage, J. & Morazan, M. (eds), LNCS, vol. 6647. Springer, Berlin/Heidelberg, pp. 38–53.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Berthold, J. & Loogen, R. (2005) Skeletons for recursively unfolding process topologies. In *Proceedings of ParCo 2005*, Joubert, G. R., Nagel, W. E., Peters, F. J., Plata, O. G., Tirado, P. & Zapata, E. L. (eds), John von Neumann Institute of Computing Series, vol. 33. Central Institute for Applied Mathematics, Jülich, Germany, pp. 835–843.

Berthold, J. & Loogen, R. (2007) Parallel coordination made explicit in a functional setting. In *IFL'06: Implementation and Application of Functional Languages*, Horváth, Z. & Zsók, V. (eds), LNCS, vol. 4449. Springer, Berlin/Heidelberg, pp. 73–90.

Berthold, J., Klusik, U., Loogen, R., Priebe, S. & Weskamp, N. (2003) High-level Process Control in Eden. In *EuroPar 2003 – Parallel Processing*, Kosch, H., Böszörményi, L. & Hellwagner, H. (eds), LNCS, vol. 2790. Springer, Berlin/Heidelberg, pp. 732–741.

Berthold, J., Loidl, H.-W. & Al Zain, A. D. (2008) Scheduling light-weight parallelism in ArTCoP. In *PADL'08 — Practical Aspects of Declarative Languages*, Hudak, P. & Warren, D. (eds), LNCS, vol. 4902. Springer, Berlin/Heidelberg, pp. 214–229.

Bevan, D. I. (1987) Distributed garbage collection using reference counting. In *PARLE'87 — Parallel Architectures and Languages Europe*, LNCS, vol. 259. Springer, Berlin/Heidelberg, pp. 176–187.

Breitinger, S., Klusik, U., Loogen, R., Ortega Mallén, Y. & Peña Marí, R. (1997) DREAM — the DistRibuted Eden abstract machine. In *IFL'97: 9th International Workshop on the Implementation of Functional Languages*, LNCS, vol. 1467. Springer, Berlin/Heidelberg, pp. 250–269.

Breitinger, S., Klusik, U. & Loogen, R. (1998) From (Sequential) Haskell to (Parallel) Eden: An implementation point of view. In Proceedings of the 10th International Symposium on Principles of Declarative Programming, LNCS, vol. 1490. Springer, Berlin/Heidelberg, pp. 318–334.

Cejtin, H., Jagannathan, S. & Kelsey, R. (1995) Higher-order distributed objects. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **17**(5), 704–739.

Chakravarty, M. M. T., Leshchinskiy, R., Peyton Jones, S., Keller, G. & Marlow, S. (2007) Data parallel Haskell: A status report. In *DAMP'07, Workshop on Declarative Aspects of Multicore Programming*. ACM, New York, pp. 10–18.

Cole, M. I. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation*, Research Monographs in Parallel and Distributed Computing. Cambridge(MA), USA: MIT Press.

Dieterle, M., Horstmeyer, T. & Loogen, R. (2010) Skeleton composition using remote data. In *PADL 2010: Practical Aspects of Declarative Languages*, LNCS, vol. 5937. Springer, Berlin/Heidelberg, pp. 73–87.

Dieterle, M., Horstmeyer, T., Berthold, J. & Loogen, R. (2013) Iterating Skeletons – structured parallelism by composition. In *IFL'12: 24th Symposium on Implementation and Application of Functional Languages*, Hinze, R. & Gill, A. (eds), LNCS, vol. 8241. Springer, Berlin/Heidelberg, pp. 18–36.

Du Bois, A. R., Loidl, H.-W. & Trinder, P. W. (2002) Thread migration in a parallel graph reducer. In *IFL'02: International Workshop on the Implementation of Functional Languages*, LNCS, vol. 2670. Springer, Berlin/Heidelberg, pp. 199–214.

Epstein, J., Black, A. P. & Peyton-Jones, S. (2011) Towards Haskell in the cloud. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell'11)*. ACM, New York, pp. 118–129.

Fluet, M., Rainey, M., Reppy, J., Shaw, A. & Xiao, Y. (2007) Manticore: A heterogeneous parallel language. In *DAMP 2007: Workshop on Declarative Aspects of Multicore Programming*. ACM, New York, pp. 37–44.

Fluet, M., Rainey, M., Reppy, J. & Shaw, A. (2010) Implicitly threaded parallelism in Manticore. *J. Funct. Programm.* **20**(5–6), 537–576.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Foltzer, A., Kulkarni, A., Swords, R., Sasidharan, S., Jiang, E. & Newton, R. (2012) A Meta-scheduler for the Par-monad: Composable scheduling for the heterogeneous cloud. In *ICFP'12: 17th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, pp. 235–246.

Geist, Al. (2011) Parallel virtual machine. In *Encyclopedia of Parallel Computing*, Padua, D. (ed), Heidelberg/Berlin: Springer, pp. 1647–1651.

Gray, J. (1985) *Why Do Computers Stop and What Can Be Done About It?* Tandem Computers, Technical Report 85.7.

Hallgren, T., Jones, M. P., Leslie, R. & Tolmach, A. (2005) A principled approach to operating system construction in Haskell. In *ICFP'05: 10th ACM SIGPLAN International Conference on Functional Programming*, Danvy, O. & Pierce, B. C. (eds), ACM, New York, pp. 116–128.

Hammond, K. (1993 September) Getting a GRIP. *IFL'93: International Workshop on the Parallel Implementation of Functional Languages.* Nijmegen, the Netherlands (draft proceedings).

Hammond, K. (2011) Glasgow parallel Haskell (GpH). In *Encyclopedia of Parallel Computing*, Padua, D. (ed), Heidelberg/Berlin: Springer, pp. 768–779.

Hammond, K. & Peyton Jones, S. L. (1990) Some early experiments on the GRIP parallel reducer. In *IFL'90: International Workshop on the Parallel Implementation of Functional Languages.* TR 90-16, Department of Informatics, University of Nijmegen, pp. 51–72.

Hammond, K. & Peyton Jones, S. L. (1992 September) Profiling scheduling strategies on the GRIP multiprocessor. In *IFL'92: International. Workshop on the Parallel Implementation of Functional Languages.* vol. 92–19, Aachener Informatik-Berichte, pp. 73–98.

Klusik, U., Ortega-Mallén, Y. & Peña Marí, R. (1999) Implementing Eden – or: Dreams become reality. In *IFL'98: 10th International Workshop on the Implementation of Functional Languages*, LNCS, vol. 1595. Springer, Bertln/Heidelberg, pp. 103–119.

Lameter, C. (2013) NUMA (non-uniform memory access): An overview. *Acm Queue* **11**(7), 40:40–40:51.

Li, P., Marlow, S., Peyton Jones, S. & Tolmach, A. (2007) Lightweight concurrency primitives for GHC. In *ACM SIGPLAN Workshop on Haskell (Haskell'07)*. ACM, New York, pp. 107–118.

Loidl, H.-W. (1998 (March) *Granularity in Large-Scale Parallel Functional Programming*. PhD Thesis, Dept. of Computing Science, Univ. of Glasgow.

Loidl, H.-W. (2001) Load balancing in a parallel graph reducer. In *SFP'01 — Scottish Functional Programming Workshop*, Hammond, K. & Curtis, S. (eds), Trends in Functional Programming, vol. 3. Intellect, Bristol, pp. 63–74.

Loidl, H.-W. & Hammond, K. (1994 September) GRAPH for PVM: Graph reduction for distributed hardware. In *IFL'94: International Workshop on the Implementation of Functional Languages.* Norwich, England (draft proceedings).

Loidl, H.-W. & Hammond, K. (1996) Making a packet: Cost-effective communication for a parallel graph reducer. In *IFL'96: International Workshop on the Implementation of Functional Languages*, LNCS, vol. 1268. Springer, Bertln/Heidelberg, pp. 184–199.

Loogen, R., Ortega-Mallén, Y. & Peña-Marí, R. (2005) Parallel functional programming in Eden. *J. Funct. Programm.* **15**(3), 431–475.

Maier, P. & Trinder, P. (2012) Implementing a high-level distributed-memory parallel Haskell in Haskell. In *IFL'12: 24th Symposium on Implementation and Application of Functional Languages*, Gill, A. & Hage, J. (eds) LNCS 7257. Springer, Bertln/Heidelberg, pp. 35–50.

Maier, P., Livesey, D., Loidl, H.-W. & Trinder, P. (2014a) High-performance computer algebra — a parallel Hecke algebra case study. In *EuroPar'14: Parallel Processing*, LNCS, vol. 8632. Springer, Bertln/Heidelberg, pp. 415–426.

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press

Maier, P., Stewart, R. & Trinder, P. (2014b) The HdpH DSLs for scalable reliable computation. In Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell (Haskell'14). ACM, New York, pp. 65–76.

Marlow, S. & Peyton Jones, S. (2011) Multicore garbage collection with local heaps. In *ISMM '11: Proceedings of the 10th International Symposium on Memory Management*. ACM, New York, pp. 21–32.

Marlow, S., Peyton Jones, S. & Singh, S. (2009) Runtime support for multicore Haskell. In *ICFP'09: 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, pp. 65–78.

Marlow, S., Maier, P., Loidl, H.-W., Aswad, M. K. & Trinder, P. (2010) Seq no More: Better Strategies for Parallel Haskell. In Proceedings of the Third ACM Haskell Symposium (Haskell'10). ACM, New York, pp. 91–102.

Marlow, S., Newton, R. & Peyton Jones, S. (2011) A monad for deterministic parallelism. In Proceedings of the 4th ACM Haskell Symposium (Haskell'11). ACM, New York, pp. 71–82.

Mohr, E., Kranz, D. A. & Halstead Jr., R. H. (1991) Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.* **2**(3), 264–280.

MPI Forum (ed). (2012) *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS). Available at: `http://www.mpi-forum.org/docs/`. [Retrieved 14/12/2015]

Peyton Jones, S., Clack, C., Salkild, J. & Hardie, M. (1987) GRIP — a high-performance architecture for parallel graph reduction. In *Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA'87)*, LNCS, vol. 274. Springer, Bertln/Heidelberg, pp. 98–112.

Reppy, J., Russo, C. & Xiao, Y. (2009) Parallel concurrent ML. In *ICFP'09: 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, New York, pp. 257–268.

Sivaramakrishnan, K. C., Harris, T., Marlow, S. & Peyton Jones, S. (2013) *Composable Scheduler Activations for Haskell*. Technical Report, Microsoft Research, Cambridge.

Stewart, R., Trinder, P. & Maier, P. (2012) Supervised workpools for reliable massively parallel computing. In *TFP12: International Symposium on Trends in Functional Programming*, LNCS, vol. 7829. Springer, Bertln/Heidelberg, pp. 247–262.

Totoo, P. & Loidl, H.-W. (2014) Parallel Haskell implementations of the N-body problem. *Concurrency Comput.: Pract. Exp.* **26**(4), 987–1019.

Trinder, P. W., Hammond, K., Loidl, H.-W. & Peyton Jones, S. (1998) Algorithm + Strategy = Parallelism. *J. Funct. Programm.* **8**(1), 23–60.

Trinder, P. W., Hammond, K., Mattson Jr., J. S., Partridge, A. S. & Peyton Jones, S. L. (1995) GUM: A portable parallel implementation of Haskell. In *IFL'95: 7th International Workshop on the Implementation of Functional Languages*. Båstad, Sweden (draft proceedings).

https://doi.org/10.1017/S0956796816000010 Published online by Cambridge University Press