Implementation Designs for Parallel Functional Languages

Hans-Wolfgang Loidl
Institut für Informatik
Ludwig-Maximilians
Universität
D 80538 München
Germany
hwloidl@tcs.informatik.uni-muenchen.de

Norman Scaife
School of Information Science
Japan Advanced Institute
for Science and Technology
Tatsunokuchi, Ishikawa,
923-1211
Japan
norman@jaist.ac.jp

Greg Michaelson, Phil Trinder
School of Mathematical and
Computer Sciences
Heriot-Watt University
Edinburgh EH14 4AS
Scotland
{greg,trinder}@macs.hw.ac.uk

ABSTRACT

Most parallel functional languages have high-level coordination constructs, freeing the programmer from specifying low-level details of how parallel computations interact. To achieve good parallel performance a sophisticated language implementation is required that effectively and efficiently manages the low level coordination aspects. The implementation will include facilities to manage threads and storage on each processor and communication between processors. The management may be determined statically at compile time, dynamically at runtime or in a combination of both.

We investigate alternative implementation design decisions for parallel functional languages, and evaluate their impact on performance in two implementations: GpH and PMLS. The former extends Haskell 98 with a parallel combinator, while the latter is a subset of SML with a small set of internally parallelised functions, or algorithmic skeletons. We measure three GpH and PMLS programs on a Beowulf cluster and a SunSMP and evaluate the implications of design decisions in the thread, storage and communication management subsystems. For irregularly-parallel programs a highly-dynamic approach to work and data distribution, as in GpH, is preferable over PMLS's more static implementation, despite significantly higher overheads. However, for regularly-parallel programs static work placement, as in PMLS, is advantageous.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Functional Languages*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

Keywords

Parallel Functional Languages, Implicit Parallelism

1. INTRODUCTION

Most parallel functional languages match a high-level computation language with high-level coordination control. Analogous to high-level computation notations, high-level coordination requires sophisticated static and dynamic implementation to deliver good performance. Several important coordination aspects must be effectively managed, including threads, storage, and communication.

Although the potential of functional languages to support parallelism with minimal programmer intervention has been long recognised [34], the additional complexity of managing high-level coordination on a range of parallel architectures has hampered the development of mature parallel functional language implementations until recently. Robust, portable implementations like GpH [33], PMLS [24], NESL [3], and Eden [5] now allow us to investigate alternative coordination management designs for parallel functional languages.

A key design decision governs whether a coordination aspect is managed using static techniques, dynamic techniques, or both. Static coordination management requires a sophisticated compiler incorporating analysis or profiling, and gives faster runtime at the price of being less flexible as it is unable to adapt to dynamic properties of the program. Dynamic coordination management requires a sophisticated runtime system and is more flexible at the price of incurring the management overhead at runtime. The main measurements in this paper assess the relative costs and benefits of static versus dynamic coordination management for GpH and PMLS.

The paper is structured as follows. Section 2 shortly summarises the languages, GpH and PMLS. Section 3 presents a classification of implementation design choices. Sections 4 and 5 locate the GpH and PMLS systems, respectively, in this classification. In Section 6 we present measurements of these two systems on a commonly-available, high-latency Beowulf architecture, analysing the impact of thread, memory and communication management mechanisms on parallel performance. Section 7 concludes by summarising the impact of the design decisions on parallel performance.

2. THE LANGUAGES: GPH AND PMLS

Glasgow Parallel Haskell, GpH is a small, conservative extension to Haskell 98, a polymorphic, pure, lazy functional

language. Parallel ML with Skeletons, PMLS is based on the pure functional subset of Standard ML (SML), a polymorphic imperative language. GpH and PMLS embody very different approaches to parallelism from a programmer's perspective. In GpH sites of potential parallelism are nominated explicitly where PMLS tries to exploit potential parallelism implicit in sequential programs.

GpH introduces the `par` construct which indicates that its arguments may be evaluated in parallel. Typically, `par` is used to construct generic parallel *evaluation strategies* [32], which combine a higher order function (HOF) with corresponding uses of `par` to apply the HOF's argument function in parallel across elements of the data structure.

In contrast, PMLS exploits parallelism latent in HOFs by instantiating and invoking equivalent parallel algorithmic skeletons [8]. Where a program lacks HOFs, they may be synthesised automatically using proof planning [9].

We will not discuss parallel programming in GpH and PMLS further here. Rather, we will focus on how their modes of exploiting parallelism are realised in their implementations.

3. IMPLEMENTATION DESIGNS

This section discusses basic aspects of the parallel execution of functional languages and in doing so exhibits design choices in implementing key mechanisms of parallel coordination. For a general discussion of parallel functional languages and their implementations we refer to [13].

3.1 Classification of Parallelism

A common classification of parallel programming languages is provided by Sarkar [30]. It identifies the following three steps in the creation and execution of a parallel program:

- Decomposition, the partitioning of the program into a set of separate tasks;
- Mapping, the placement of these tasks onto a set of available processors;
- Scheduling, the decision, which task to run on a processor, if more than one task is available for computation.

In PMLS *decomposition* is static and implicit in the compiler, which decides when and where to generate parallelism. In GUM decomposition is dynamic and explicit in the runtime system, which upon finding a `par` construct may generate a parallel thread.

In PMLS *mapping* is static and implicit in the compiler, which allocates skeleton components to processors using sequential instrumentation to predict and balance load. In GUM mapping is performed dynamically at run time through idle processors seeking threads from busy processors.

In PMLS *scheduling* is trivial as each processor runs one skeleton component at any given moment, depending on the call hierarchy. In GUM a multi-threaded runtime-system is used to manage the available parallelism.

3.2 Thread Management Model

The *thread management model* determines how parallel threads are generated and how they interact. In particular it decides when to create parallel tasks (*sparking*), how to prevent tasks from evaluating the same expression (*locking*), and how to keep track of and ensure data transfer to tasks that wait for the result of a computation (*waiting list*).

One crucial aspect for performance is the *granularity* of the parallel tasks. Many modern parallel runtime systems put particular emphasis on minimising thread creation overhead to encourage the generation of a large amount of fine-grained parallelism (Cilk [4], CPar [22], Filaments [21]). Related to granularity is the question whether a hierarchy of threads with varying overheads is supported. One prominent example of a hierarchy of tasks is the Split-C systems [10], and its successor TAM [11], which supports totally independent tasks based on functional calls, several threads within each task and specialised inlets (with the guarantee that no blocking occurs within an inlet) for handling the arrival of new data. Obviously, the lower level the notion of a task the cheaper is a context switch (because more of the environment is shared) but the more limited is the execution that can be performed by the task. In that context the threads in GpH are rather high-level, with independent sets of registers and independent stacks. The skeletons in PMLS internally manage a flat set of threads, which are similar in complexity to those in GpH.

3.3 Storage Management Model

The storage management model deals with the distribution of the heap among the processors. Two basic choices in the representation of the heap are to use a *virtual shared heap*, as in GpH, or *disjoint heaps* with explicit, runtime-system-level communication of data, as in PMLS. Sections 4 and 5 discuss these components in more detail.

Since each thread needs a stack, its representation is another important design decision in the implementation. Possible choices are discussed in the following paragraphs.

Early machines like ALICE [14] or the HDG machine [16] used a *linked list of packets* (similar to stack frames). This has low runtime overhead, since the packet size is known at compile time and the allocation can be done at compile time, too. However, there is a serious danger of space leaks, since a frame is alive as long as any of its variables are alive.

A *segmented stack*, as used in GpH and Eden [5], allocates pieces of stack in the heap and links them together to form the overall stack. It is a compromise between the linked list of packets and the contiguous stack model. It is flexible enough to deal with small threads, with only a small overhead for extending the initially rather small stack. The waste of heap space described above is avoided by separating the structure of the stack from that of a normal heap closure, and performing a stack traversal to find the roots of the computation whenever a garbage collection is encountered.

The *contiguous stack* model, as used in PMLS, uses one monolithic block of a stack dedicated to one processor. It is possible to extend the stack at runtime but this operation is expensive and should be avoided as far as possible. The

main disadvantage is the waste of space in the general case.

A *meshed stack*, or spaghetti stack, as used in PASTEL [15], interleaves all logical stacks of local tasks on one physical stack (not in the heap), and thus reduces overhead in the case of sequential execution. A special compacting garbage collector is needed to eliminate "holes" in the stack left behind from threads with stack frames not on the top of the stack.

3.4 Communication Model

The communication model describes how data is exchanged between processors. This in turn depends on three aspects of the runtime-system: the *data placement* mechanism, *latency hiding* and the *packing* mechanism.

Data placement can be governed by a local evaluation or remote mechanism. Local evaluation means that if a process needs data from a remote processor and the data hasn't been evaluated, yet, the unevaluated data is sent and then evaluated locally. GpH uses such a mechansim. In contrast, PMLS will in such a case rely on the remote evaluation of the data, which will be subsequently sent.

Latency hiding is the ability to perform useful computation while waiting for remote data and is typically achieved by overlapping communication with computation via a multi-threaded runtime-system. Provided enough work is available, the time until a requested data item arrives can be spent by executing another thread. However, this imposes potentially high context switching overhead, and may lead to high data fragmentation. GpH uses a fixed FIFO scheduler, and relies on a large number of threads to hide frequent communication. In contrast, in PMLS this behaviour is specified in the implementation of the skeleton.

The *packing mechanism* determines how much of a data structure should be sent upon a data request. In more conventional terms, this decides how much pre-fetching to do.

3.5 Load Distribution

The load distribution mechanism controls how the parallel threads are distributed among the available processors in order to achieve good performance. A basic distinction is between

- *Passive load distribution* (or *work stealing*), where idle processors have to explicitly ask for new work (as in GpH, HDG [16], ZAPP [6] and PAM [20]), and
- *Active load distribution*, where new work is automatically sent to an idle processor (as in [12]).

The advantage of passive load distribution is the reduction of overhead messages because work is only sent when it is needed by a processor. To minimise processor idle time this mechanism is often combined with a mechanism that asks for work if the amount of available work falls below a certain limit, and throttles the generation of new work if it is beyond a certain limit [28]. Overall, passive load distribution best fits a high-latency environment where additional messages and poor data locality, resulting from an aggressive load distribution mechanism, should be avoided as far as possible. In contrast, active load distribution automatically sends new work to an idle processor if one is available, possibly generating more communication.

4. THE IMPLEMENTATION OF GPH

In this section we locate the implementation of GpH, the GUM system, in the design space discussed so far. The key concepts in the design of GUM are an underlying distributed memory model, a virtual shared heap implemented on top of that model and automatic, dynamic resource control for both work and data. For compilation GpH uses a slightly modified version of the Glasgow Haskell Compiler. For a more detailed discussion see [33].

4.1 The Thread Management Subsystem

The thread management subsystem in GUM uses an *evaluate-and-die* mechanism. This model represents potential parallelism as *sparks*, i.e. pointers to structures in the distributed program graph. In the program code a `par` primitive is used to generate sparks, which are maintained by the runtime system in a flat, distributed spark pool. A "sparked" expression may be executed by an independent thread. However, if a thread needs the value of the expression, and no other thread is evaluating it, this thread will perform the computation itself. This behaviour is called *thread subsumption* because the potentially parallel work is inlined by another thread, typically its parent. This idea of dynamically increasing the granularity of the threads by deferring the decision whether to generate a thread is similar to the independently developed lazy task creation model [25].

The synchronisation between threads is implicit via accessing shared *closures*, i.e. nodes in the graph structure. We distinguish between *normal-form closures*, which represent data, and *thunks*, which represent work (unevaluated data). If a thread needs the result of a computation, represented by a graph structure, that is currently active or that is executed on another processing element (PE), the thread is blocked on this graph structure. As soon as the result becomes available the thread is awoken and can continue. To realise this form of blocking, every thread has to lock a closure on entry, and a waiting list of all blocked threads has to be maintained.

The basic load balancing mechanism in GUM is one of *work stealing*, i.e. whenever a PE runs out of work it asks other PEs for work. If work is available it will be transferred as a spark from the busy to the idle PE.

4.2 The Memory Management Subsystem

GUM uses a *flat memory hierarchy*, i.e. the access to any closure in one PE's heap is uniform. Every globally visible closure in the heap is identified via a global address (GA), a globally unique identifier. Global indirection closures (FetchMes) always use the GA to identify the remote object. The mapping of these GAs to local heap addresses and vice versa is done via a hash table, the global indirection table (GIT).

This design enables separate local garbage collection (GC) on each PE, using weighted reference counting on GAs [1].

After each GC the GIT has to be rebuilt to map all live GAs to their new addresses in the heap. This design is based on the observation that only a small fraction of the heap is globally visible and that rebuilding the GIT is cheaper than allocating and maintaining one additional word, the GA, for every closure in the system. The mapping of global to local addresses enables the system to determine whether a newly imported graph structure already exists on that PE and to avoid duplication of data in this case. Duplication of work is avoided by always moving, rather than copying, thunks between PEs.

4.3 The Communication Subsystem

GUM was designed for a generic distributed-memory machine. To reduce the total amount of communication and to permit *latency hiding*, GUM uses asynchronous, bulk communication, packing an entire graph structure. The packing mechanism performs a breadth first graph traversal, limited by the fixed packet size, i.e. when a graph is larger than the packet size only its initial portion is sent (full subgraph packing). Since both work and data are represented as graph structures, both can be transmitted using this mechanism. However, we rely on the actual compiled code to be preloaded on each PE.

5. THE IMPLEMENTATION OF PMLS

5.1 Overview

The PMLS Compiler[1] has the following major components:

- The *front end* parses and type checks a sequential program, and constructs an internal representation based on the abstract syntax tree (AST).
- The *profiler* then runs the sequential program on *test data* sets to gather information about processing and communication costs.
- The *analyser* identifies HOFs and instantiates the corresponding skeleton *performance models* with profile information to determine whether parallelism is exploitable.
- The *transformer* rearranges the prototype, guided by the *analyser* to try and optimise sites of parallelism.
- The *back end* generates the parallel implementation as an imperative program invoking suitably instantiated *skeleton*s.

The central core of the PMLS compiler uses mature ML systems: the ML Kit [2] for the front end, because it provides an evaluator for SML based closely upon the structural operational semantics; Objective Caml [7] for the back end, because of its relatively lightweight implementation and very efficient garbage collection mechanism. While the front end phase in the compiler, which generates parallel OCaml, is architecture independent, the back end makes use of architecture specific properties via C library calls. This design and the choice of MPI [26] as general-purpose communication layer, ease portability of the system, which now runs on several different parallel architectures.

[1]This account draws heavily on that from [23].

5.2 Skeleton Design

The efficient use of skeletons in nested mode is known to be a difficult problem to solve, so we focus on the development of versions of `map` implemented as a process farm and `fold` implemented as a divide-and-conquer tree. The *communication model* of PMLS uses the Objective Caml `Marshal` module for linearising arbitrary types of data, in conjunction with the MPI `MPI_BYTE` datatype. While the Objective Caml foreign function interface is untyped, our compiler generates code from correctly typed source code using semantics-preserving transformations.

The *thread management model* encoded in the implementations of PMLS' `map` and `fold` skeletons provides novel ways of handling nested constructs, while still usgin an SPMD approach, where execution starts as a set of identical programs running on a fixed set of processors. Upon encountering the first skeleton, the familiar *master-worker* model is initiated and this remains so throughout skeleton execution. When a skeleton terminates the master processor can optionally broadcast the final results to the workers which only have partial results at that point.

Processor allocation, or *mapping*, is carried out at startup according to the data extracted from the program by the compiler. Each processor is allocated to a skeleton in a top-down manner. Upon encountering a skeleton, the worker processors resume execution in the instance function for the skeleton to which they were allocated once the initialisation data for that instance function becomes available. This also applies to inner nested skeletons giving a relatively efficient method of nested initialisation. A complication, however, is that free-values have to be propagated through nested levels. The case for inner nested skeletons requires a slightly modified analysis compared with the outer level where we could make use of the initial phase of joint sequential execution to minimise the amount of free-value transmission. It also creates the need to distinguish between worker processors which only perform work under the auspices of a master processor, and worker processors which are themselves master processors for inner skeletons.

The `pmap` skeleton executes as a processor farm with one master processor and the remaining processors as workers. Since our skeletons are list-based the unit of granularity is a fixed-size block of list elements. The block size is set as a global constant with the value set at run-time but the intention is to use the profiling information to set this automatically. The `pfold` skeleton operates as a binary tree of processors. Here, all non-leaf processors operate as nested masters where a distinction is made between the *root* master for the `pfold` skeleton and a *parent* processor for a node in the processor tree.

One limitation of these skeletons is that the master-worker model tends to cause a bottleneck in communications at the master processor. We generally get good speedups for small numbers of processors but scalability is limited by this bottleneck.

5.3 Dynamic profiling

PMLS includes an integrated *dynamic* profiling mechanism. The ML Kit interpreter is used to execute the sequential

prototype. This interpreter is a direct implementation of the dynamic semantics of SML so the compiler is able to annotate the syntax tree with information about the number of rules in the semantics which are fired during execution. This leads to sequential performance prediction.

Given a suitable set of "learning" programs it is possible to assign a weight to each rule in the semantics. The test programs cause as many of the different rules to be fired as possible and the actual execution times for the test programs form the dependent variable for weight determination. The main problem is in selecting a suitable set of test programs whereby each rule has a significant representation and no rules dominate the entire data set. This is difficult since some ubiquitous rules, such as accessing an identifier in an environment, occur far more often than other specialised rules such as raising an exception.

6. PERFORMANCE MEASUREMENTS

6.1 Program Descriptions

For the measurements we use three programs, `sumEuler`, `linSolvand` `raytracer`. `linSolv` and `raytracer` are discussed in detail in [19] and we only provide a short summary here. `linSolv` finds an exact solution of a linear system of equations over integers, using a multiple homomorphic images approach [17], which is common for symbolic computation algorithms. The input is mapped into several homomorphic images, the solutions are computed in each image independently, and finally the results are combined into a solution in the original domain. The computational structure of the algorithm is divide-and-conquer with nested parallelism in the solution phase. `raytracer` calculates a 2D image of a given scene of 3D objects by tracing all rays in a given grid, or window. Its computational structure is data parallel with data clustering over the window, i.e. a tunable number of lines is processed by one thread to increase granularity.

`sumEuler` computes the sum of Euler numbers over a list of integers. Its computational structure is fold-of-map, i.e. several `maps` followed by a `fold`. The sequential Haskell version is shown below (`mkList n` produces the list `n,n-1..1`):

```
euler :: Int -> Int
euler n = length (filter (relprime n) [1..(n-1)])

sumEuler :: Int -> Int
sumEuler = sum . map euler . mkList
```

The corresponding sequential SML algorithm is:

```
fun gcd x 0 = x
  | gcd x y = gcd y (x mod y)

fun relprime x y = gcd x y = 1
fun euler n = if n < 1 then 0
                       else zfilter (relprime n) n
val result = fold (op +) 0 (map euler nList)
```

The `euler` function counts the number of integers that are relatively prime to the given `n`. The SML version uses `zfilter p n` to count the number of elements for which the predicate `p` is true in the integer list `[1..n]`.

The parallel GpH version uses trivial data parallelism and a generic clustering mechanism to combine the execution of several elements in the list into one thread and thereby increase granularity. In this case, `cluster` splits a list into a list of lists of size `z` and `lift` applies a function to all elements in the list of lists. The `parList rnf` function is an evaluation strategy [32] causing the parallel evaluation of all list elements, in this case sub-lists. The degree of evaluation is controlled by `rnf`, for reduction to normal form.

```
sumEuler :: Int -> Int -> Int
sumEuler z n =
  sum ((lift worker) (cluster z (mkList n))
       `using` parList rnf)
  where worker = sum . map euler
```

In PMLS we can directly implement the `map` in parallel, the `fold` being treated as a sequential function:

```
val result = seqfold (op +) 0
               (map sumeuler (split c nList))
```

A nested implementation can be obtained by representing the integers as lists of datatypes (`EI`) and manually splitting this list:

```
fun eulerEI (I n) = E (euler n)
  | eulerEI (E n) = E n
fun sumeulerEI (eil1,eil2) =
  [seqfold eiplus (E 0) (map eulerEI (eil1 @ eil2))]
val [E result] =
  fold sumeulerEI [E 0] (split c (seqmap I nList))
```

Only the outer `fold` and the inner `map` are likely to lead to parallelism. This results in a nested `map` within a `fold`.

6.2 Measurement Setup

All measurements have been performed on a 32-node Beowulf cluster [27] at Heriot-Watt University, consisting of Linux RedHat 6.2 workstations with a 533MHz Celeron processor, 128kB cache, 128MB of DRAM and a 5.7GB local IDE disk. The workstations are connected through a 100Mb/s fast Ethernet switch with a latency of 142μs, measured under PVM 3.4.2. The PMLS results on a SunSMP are obtained from a Sun Wildfire with 900MHz UltrSPARC III processors, 32KB Level 1 (on-chip) instruction cache and 64KB Level 1 data cache, 8MB Level 2 external cache, and 16GB of main memory. A sustained bus bandwidth of 43.2GB/s is reported.

6.3 Performance Results

Figures 1, 2 and 3 summarise runtimes and speedups for `sumEuler`, `linSolv` and `raytracer`, respectively, under GpH and PMLS on a Beowulf. For PMLS we additionally report

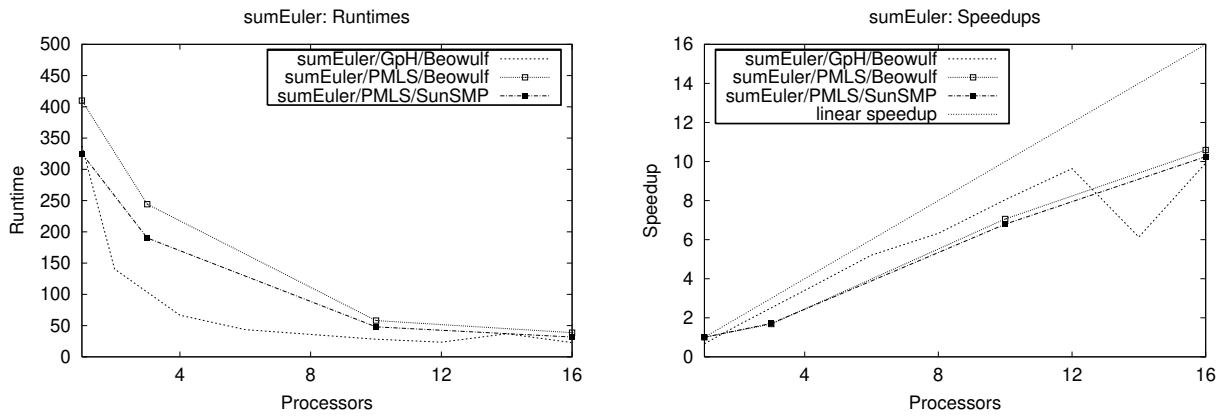

Figure 1: Runtimes and speedup of `sumEuler`

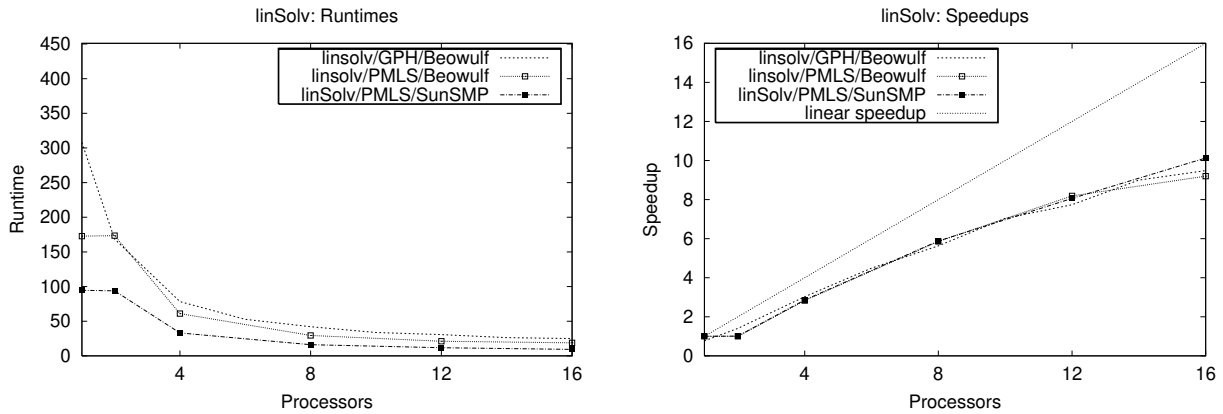

Figure 2: Runtimes and speedup of `linSolv`

the performance on a SunSMP. The GpH port to the SunSMP is currently not reliable enough to report these results.

For `sumEuler`, the GpH version exhibits better single processor performance (337.6s vs. 409.9s). We attribute this to the more aggressive optimisations used in GHC, in particular avoiding the generation of intermediate lists, which is more effective on simple programs such as `sumEuler`. The speedups are similar for both systems (9.9 and 10.6), although key storage and communication management characteristics differ, as discussed below. The dip in speedup for the GpH version for 14 PEs indicates the danger of a highly-dynamic system of work distribution, which in this case leaves PEs poorly utilised for a long period.

For `linSolv`, PMLS achieves significantly better single processor performance (172.82s vs 307.77s), due to one system and one algorithmic difference. On algorithm level the PMLS version uses a slightly more efficient algorithm in the forward substitution phase of computing a solution in one homomorphic image (see [19][Sec 4.3.4]). On system level GpH employs a lazy evaluation mechanism with a less efficient two-space garbage collector. Details are discussed in Section 6.5.

For `raytracer` the sequential performance of PMLS and GpH is fairly similar, but overall speedup is worse for GpH: 6.0 vs. 15.1 for PMLS. This is mainly due to a fairly poor load balance in the GpH version towards the end where collecting the results leads to a sequential tail, i.e. only one processor is used in this stage. The PMLS skeleton is better tuned to avoid such a bottleneck. Although improved load balancing strategies for GpH manage to improve the distribution of potential work, they are ineffective once all potential work has been activated. Initial results with an extension to GpH that supports thread migration show promising results but are too preliminary to include here.

Tables 1 and 2 provide a more detailed view at the main performance characteristics of both systems on a Beowulf architecture and will be discussed in the following subsections. Tables 3 and 4 compare the performance of `linSolv` running under GpH and PMLS on a Beowulf. Additionally, results of `sumEuler` and `linSolv` under PMLS on SunSMP are reported in Tables 6 and 12. Speedups are measured over optimised sequential execution, giving absolute speedups.

The single processor performance of GpH is 67% for `sumEuler` and 77% for `linSolv`. The main reason for the

Table 1: Performance of `sumEuler` (PMLS)

PEs	Time (in sec)	Spdup (as $\frac{\text{seqRT}}{\text{parRT}}$)	Allocation (in MB)	Fragmentation (as std dev)	GC (in sec)	GC (in %)	fold eff.	map eff.
1	409.90	1.00	6	-	0.055	0.01%	-	-
3	244.16	1.68	754	323.3	1.857	0.76%	90.45	88.03
10	58.05	7.06	394	59.7	1.238	2.13%	89.54	87.90
16	38.71	10.59	333	30.4	1.221	3.15%	81.47	88.31

Table 2: Performance of `sumEuler` (GpH)

PEs	Time (in sec)	Spdup (as $\frac{\text{seqRT}}{\text{parRT}}$)	Allocation (in MB)	Fragmentation (as std dev)	Residency (in KB)	GC %
1	337.65	0.7	7259.6	-	407	30.9%
4	66.70	3.4	1814.9	285.3	111	10.5%
10	28.23	8.1	725.9	85.9	58	6.0%
16	22.89	9.9	453.7	133.1	35	3.6%

relatively low efficiency of `sumEuler` is the combination of a high allocation rate of `sumEuler` with the use of a two-space garbage collector in GpH (a current implementation restriction), rather than that of a more efficient generational garbage collector. In comparison, PMLS's single processor performance for `linSolv` is 84% (PMLS uses a generational garbage collector).

6.4 Thread Management

Parallel functional languages typically create a large amount of parallelism to be exploited by the runtime system or a skeleton tailored for a particular architecture. Hence, both PMLS and GpH have been tuned for a low overhead of creating (potential) parallelism. The sparking of parallelism in GpH only involves adding a pointer to an array and increasing a counter. The launching of a skeleton in PMLS is roughly equivalent to the generation of thread for GpH. The skeleton overheads are much lower since, due to the static analysis, the communications required to setup the skeleton are minimised.

Top-level skeleton initialisation times in PMLS are roughly constant per skeleton instance, usually about $50\mu s$. For the non-nested implementations this time is not significant but in the nested instance these times rise on the nested processors to $4ms$ for 3 processors and $0.7ms$ for 30 processors. These are still low compared to a sequential runtime of 60s. In GpH the parallel startup overhead, mainly for synchronising all available PEs and setting up internal tables of PE identifiers, is combined with the general overhead of setting up the IO monad for performing IO. This combined overhead amounts to about $40ms$ and is only paid once when starting the runtime system. The thread creation overhead in GpH with a default stack size of 1KB is ca. $8\mu s$.

The main difference between PMLS and GpH in terms of thread management is GpH's ability to subsume potentially parallel work in a thread thereby increasing the granularity of the parallelism. Previous measurements showed that this is most effective in divide-and-conquer programs. Table 5 summarises key thread management characteristics for a trivial divide-and-conquer program, `parfact`, which sums all integers in an interval by applying binary bisection in each stage. For an input possibly generating 8192 threads in the execution, the GpH version generates only 2669 on 16 PEs. Most notably a higher percentage of potential parallelism is exploited when more PEs become available. The high single processor efficiency of 93% underlines the low overhead of managing potential parallelism in the runtime-system. The heap allocation per PE is roughly constant in this example, because the units of work are independent and do not differ widely in size. The percentage of time spent on garbage collection remains low, as expected for a program without heavy use of compound data structures. Total heap allocation is very low compared to the other programs discussed in the following sections.

Table 5: Thread management for `parfact` (GpH)

PEs	Time (in sec)	Spdup (as $\frac{\text{seqRT}}{\text{parRT}}$)	No. of threads	Allocation (in MB)	GC (in %)
1	150.19	0.9	28	16.3	6.4
4	42.12	3.4	447	16.3	4.2
10	16.67	8.2	1445	16.3	4.6
16	11.04	11.9	2669	16.3	4.3

For the other programs the difference in thread management is much smaller. In particular, the program-level clustering mechanism in `sumEuler` and `raytracer` achieves explicit control over the number of tasks by the programmer. This is preferable in GpH since thread subsumption is not effective for data parallel programs. Hence, the differences in the thread management mechanism between GpH and PMLS have only a small impact on the overall performance.

6.5 Storage Management

Tables 1, 2 and 6 also show a summary of the garbage collector behaviour. Column 4 shows the total amount of allocated heap data, Column 5 shows the heap fragmentation as the standard deviation of the total amount of allocated heap taken over all PEs and Column 6 the heap residency,

Table 3: Performance of `linSolv` (PMLS)

PEs	Time (in sec)	Spdup (as $\frac{1PERT}{parRT}$)	Allocation (in MB)	Fragmentation (as std dev)	GC time (accumulated)	map eff.	
1	172.8	1.0	4658	-	31.9	18.5	-
2	173.5	1.0	5997	2347.4	31.5	18.2	98.4
4	61.0	2.8	6844	536.7	31.0	50.8	94.5
8	29.5	5.9	9267	159.8	31.3	106.1	86.1
12	21.1	8.2	11772	86.2	32.4	153.5	78.2
16	18.8	9.2	14312	63.2	32.9	175.6	73.4

Table 4: Performance of `linSolv` (GpH)

PEs	Time (in sec)	Spdup (as $\frac{seqRT}{parRT}$)	Allocation (in MB)	Fragmentation (as std dev)	Residency (in KB)	GC %
1	307.8	0.8	5286.6	—	9.7	30.9
2	168.5	1.4	2645.4	76.5	4.5	18.3
4	78.3	3.0	1322.1	67.8	1.3	10.5
8	42.1	5.6	660.9	40.1	0.9	6.3
12	30.6	7.7	440.6	22.3	0.6	5.3
16	25.0	9.5	330.4	31.6	0.6	3.6

Table 6: Performance of `sumEuler` (PMLS) on SunSMP

PEs	Time (in sec)	Spdup	Alloc. (in MB)	Frag.	GC (in %)
1	325.29	1.0	6	-	0.0%
3	190.23	1.7	3235	1737.4	0.3%
10	47.94	6.8	1406	369.4	1.2%
16	31.75	10.3	1056	202.2	2.3%

i.e. the maximum amount of live heap taken over all garbage collections. All values are averaged over all PEs. Overall the allocation in the GpH version by far exceeds those in PMLS. This reflects the general trend in the implementation of non-strict languages to allocate more data, with most of this data being very short lived [29]. This is underlined by the low values for the heap residency in Column 6, which show that despite the large amount of allocation at each garbage collection only a small part of the heap is contains live data. The high percentage of execution time spent on garbage collection, shown in Column 7, reflects the usage of a two-space garbage collector in GpH at the moment. This is an implementation restriction in the current system we plan to eliminate in the near future. However, the overall absolute speedup values show that even which such high penalty on the storage management overhead, the overall performance is still comparable to those achieved for PMLS.

The GpH results for `linSolv` in Table 4 show a roughly linear decrease in the heap allocated (per PE) for lower numbers of processors. For higher numbers, some data needs to be duplicated in order to generate more threads on the additional PEs. Overall, GpH is conservative in avoiding any duplication of work, but allows for the duplication of data needed as input for new threads. The amount of heap fragmentation is measured as the standard deviation of total heap allocation over all PEs. This parameter gives a rough idea on the balance, w.r.t. storage management, between the different PEs. Most importantly, this number drops with an increasing number of PEs, indicating that the heap allocations are not dominated by one heavily allocating producer, which would be reflected by a high standard deviation.

The PMLS results for `sumEuler` in Table 1 show consistently low heap allocation and GC overhead, but also exhibit an unusually large amount of memory allocation for 3 processors, with high heap fragmentation. Most likely this is due to an artifact of the skeleton implementation, which, in its startup routine, generates a lot of skeleton instances per node. In contrast to GpH, the total percentage of time spent in garbage collection actually increases with the number of processors, because the fixed data distribution mechanism used in PMLS leads to more duplication of data, which is imported from other PEs and does not show up in the local allocation figures. The degree of memory fragmentation drops even more rapidly than for GpH, however, high fragmentation is less problematic in PMLS since it does not directly affect the amount of communication, in contrast to GpH with its virtual shared heap.

Table 12 shows results for a PMLS version of `linSolv` on a SunSMP with faster processors and much faster communication. Despite these fundamentally different architecture characteristics the overall speedup is similar, indicating that despite using a less flexible design than GpH, PMLS can achieve high performance on many different architectures. This issue is discussed in more detail in [31]. One notable difference are the lower values on heap fragmentation: on low-latency systems it is in general easier to even out imbalances in both processor load and heap consumption since communication is much faster.

Table 7: Communication of `sumEuler` (PMLS)

	Beowulf						SunSMP					
	Root			Nested			Root			Nested		
	Sent+Recvd		Comm	Sent+Recvd		Comm	Sent+Recvd		Comm	Sent+Recvd		Comm
PEs	Count	KB	Msg/s	Count	KB	Msg/s	Count	KB	Msg/s	Count	KB	Msg/s
3	76	98.1	0.3	8524	93.7	34.9	76	98.1	0.4	8660	95.1	45.5
10	72	100.6	1.2	4406	48.3	75.9	72	100.6	1.5	4252	46.5	88.7
16	68	100.5	1.7	2504	27.4	64.7	68	100.5	2.1	2308	25.3	72.7

Table 8: Communication of `sumEuler` (GpH)

	Msgs Sent+Rcvd		Comm degree
PEs	Count	KB	Msg/sec
3	443	129.3	4.3
10	573	95.5	12.8
16	581	89.3	16.2

Table 9: Communication of `linSolv` (PMLS)

	Beowulf			SunSMP		
	Sent+Recvd		Comm	Sent+Recvd		Comm
PEs	Count	KB	Msg/s	Count	KB	Msg/s
2	160	176.5	0.9	160	176.5	1.7
4	160	176.5	2.6	160	176.5	4.8
8	160	176.5	5.4	160	176.5	9.9
12	160	176.5	7.6	160	176.5	13.6
16	160	176.5	8.5	160	176.5	17.1

6.6 Communication

Tables 7 and 8 summarise the communication behaviour of `sumEuler`, and Tables 9 and 10 those of `linSolv`. For each configuration the total number of sent and received messages, the total amount of communication in KB, and the communication degree, measured as the number of messages sent per second, is reported. The data for PMLS in Table 7 splits these figures into those for the root and the node processors of the skeleton. For direct comparison, Table 7 also reports results of running `sumEuler` under PMLS on a SunSMP. As this table shows, the differences in communication characteristics between a Beowulf and SunSMP platform are only minor for PMLS due to the static placement of threads. In particular the number of messages remains unchanged and the communication degree increases only because the SunSMP uses faster processors. Comparing the PMLS and GpH results in Tables 7 and 8, we find that the communication in GpH is usually more fine grained and more frequent: both the number of packets and the communication degree exceeds the corresponding values for PMLS. This reflects that GpH uses demand driven communication, where only the immediately needed portion of a data structure is transfered. Interestingly the amount of data sent per PE in the GpH version drops with an increasing number of PEs.

The more dynamic nature of the parallelism in `linSolv` is reflected in Table 10 by a higher number of messages, increasing by about an order of magnitude from 2 to 16 PEs, and a higher communication degree. Again we find much more communication in comparison to PMLS (Table 9). The high number of messages for `raytracer` in Table 11 is partly due to the splitting of large data structures into several packets that is performed in GpH. Consequently, the average packet size is considerably larger: 315.0 bytes, compared to 244.3 bytes for `linSolv`, and 155.5 bytes for `sumEuler`. In contrast to `linSolv`, the communication happens almost exclusively at the beginning and at the end of the computation.

One artifact of the PMLS implementation, that has an impact on the communication performance of `sumEuler`, is the implementation of the `fold` skeleton. It has to encode lists of datatypes as integers and additionally requires extra information which maintains the order of `fold` function applications (this was found necessary in other applications). This encoding adds data that has to be communicated and hence increases the overall communication overhead. In contrast, GpH uses a packing mechanism directly operating on the internal graph structure representing the data and can therefore achieve higher performance when packing arbitrary data structures. In previous measurements we have determined the packing overhead to be less than 1% for all programs, except for `raytracer` (up to 4.8%), which has to transfer the geometric model of the scene [18].

Table 10: Communication of `linSolv` (GpH)

	Msgs Sent+Recvd		Com deg
PEs	Count	KB	Msg/sec
2	126	25.2	0.5
4	907	194.1	12.5
8	1001	253.6	21.6
12	1091	263.5	34.9
16	1114	272.1	38.5

The GpH results for `raytracer` in Table 11 show a much

Table 11: Communication of `raytracer` (GpH)

	Msgs Sent+Recvd		Com deg
PEs	Count	KB	Msg/sec
2	3991	1764.9	31.9
4	5543	2171.2	77.8
8	12710	4537.5	222.4
12	19135	6656.1	392.6
16	22654	7789.6	482.7

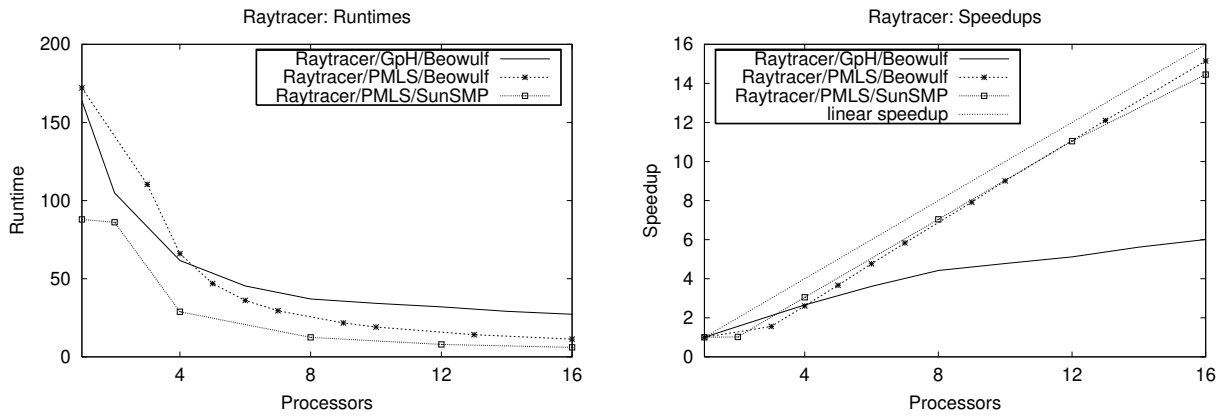

Figure 3: Runtimes and speedup of `raytracer`

larger amount of communication, compared to the `sumEuler` results in Table 8, and a roughly linear increase in the amount of communication over the number of PEs. This is explained by the predominantly data-parallel nature of this application, which does not require intermittent exchange of data. The total amount of communicated data does not increase equally rapidly because for small numbers of PEs larger blocks of data can be exchanged leading to more efficient communication in total. In summary, for a program with rather regular parallelism such as `raytracer`, PMLS's static approach is more appropriate.

7. CONCLUSION

This paper explored implementation design choices for parallel functional languages and evaluated their impacts in two implementations: GpH and PMLS. Two design decisions have a major impact on the dynamic behaviour of the systems: *work distribution*, which is dynamic in GpH but statically determined by PMLS skeletons, and *communication*, which is fixed in a PMLS skeleton but dynamic and demand-driven in GpH. Demand-driven communication determines the amount and granularity of communication, which is crucial on high-latency systems such as the Beowulf clusters used. To evaluate the implementation design choices we have measured three GpH and PMLS programs on a Beowulf cluster and a SunSMP. We have attributed the differences in dynamic behaviour to key components of the implementations, namely the thread management system, the storage management system and the communication system. As a result of the design decisions the two implementations have very different dynamic properties.

Thread management. The number of *threads* generated during the execution is fixed in PMLS using static analysis, whereas GpH can adjust the amount of parallelism dynamically. PMLS's compile-time decision is effective for programs with regular parallelism, in particular `raytracer` and to a lesser extent `sumEuler`, where much of the parallelism is easy to predict statically, whereas GpH is more flexible in dynamically adjusting to changing load on the PEs. Overall, PMLS's low-overhead approach performs better than GpH's high-flexibility approach for the programs used here.

Storage management. GpH performs far more allocation than PMLS: for low numbers of PEs the difference can be up to an order of magnitude. While this is a concern for sequential performance, it is to be expected in a non-strict language. More important for assessing garbage collection overhead is the amount of heap residency, and this value does not show such big differences. Overall, storage management overhead does not adversely affect the scalability of the speedup, since with increasing number of PEs differences in heap allocation and residency become smaller.

Communication. GpH sends more, smaller messages than PMLS in the larger `linSolv` and `raytracer`programs. PMLS profits from good static thread placement, and correspondingly good data locality, in the case of the data-parallel `raytracer` program. GpH exhibits more communication for less regularly-parallel programs such as `linSolv`, where increased communication is the price of more flexible, dynamic data and work distribution. The GpH heap fragmentation results indicate that sufficient data locality is retained to bound the total amount of communication in the system, which is crucial for the scalable parallel performance.

Both systems discussed here can be rated as mature research systems for parallel functional programming. Work on GpH started in 1994 and since then work has been done on system, programming methodology and applications level. The current version is publicly available as version GUM-5.02.3 [35] and from the main GHC CVS repository. Future work will focus on increasing the cross platform performance of the system by improving load balancing and data distribution mechanisms, and to enhance GpH's dynamic adaption mechanisms. Work on PMLS started in 1997, building on previous work on skeleton-based parallelism and parallel computer vision algorithms, and excels in high performance on a range of parallel architectures [31]. The system is available on request from the authors. Future work will focus on providing a more expressive set of skeletons, improving the performance of existing skeletons and automatically synthesising skeleton structures out of arbitrary code [9].

8. REFERENCES

Table 12: Performance of `linSolv` (PMLS) on a SunSMP

PEs	Time (in sec)	Spdup (as $\frac{\text{seqRT}}{\text{parRT}}$)	Allocation (in MB)	Fragmentation (as std dev)	Residency (in KB)	GC time (in sec)	map eff.
1	94.60	1.0	4658	-	10.6	11.2	-
2	93.83	1.0	7739	1115.7	10.2	10.9	99.8
4	33.10	2.9	7556	181.4	10.3	31.1	96.1
8	16.10	5.9	9619	42.6	10.6	65.6	86.8
12	11.74	8.1	12029	21.1	10.7	91.7	81.2
16	9.34	10.1	14520	21.8	11.2	119.8	72.7

Table 13: Performance of `raytracer` (GpH)

PEs	Time (in sec)	Spdup (as $\frac{\text{seqRT}}{\text{parRT}}$)	Allocation (in MB)	Fragmentation (as std dev)	GC (in %)	Residency (in KB)
1	163.83	1.0	342.8	—	6.4	3.2
2	104.89	1.6	172.2	7.5	3.4	2.6
4	61.65	2.7	86.4	8.3	1.7	2.4
8	37.06	4.4	43.2	15.1	0.9	1.9
12	32.01	5.1	28.8	18.5	0.8	1.5
16	27.27	6.0	21.6	14.4	0.7	1.5

[1] D.I. Bevan. Distributed Garbage Collection Using Reference Counting. In *PARLE'87 — Parallel Architectures and Languages Europe*, number 259 in LNCS, pages 176–187, Eindhoven, The Netherlands, June 12–16, 1987. Springer-Verlag.

[2] L. Birkedal, N. Rothwell, M. Tofte, and D. N. Turner. The ML Kit (Version 1). Technical Report 93/14, Department of Computer Science, University of Copenhagen, 1993.

[3] G.E. Blelloch. Programming Parallel Algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[4] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP'95 — Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 19–21, 1995.

[5] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí. The Eden Coordination Model for Distributed Memory Systems. In *HIPS'97 — Workshop on High-level Parallel Programming Models*, pages 120–124, Geneva, Switzerland, April 1997. IEEE Computer Science Press.

[6] F.W. Burton and M.R. Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *FPCA'81 — Conf. on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[7] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, April 2000.

[8] Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts, 1989.

[9] A. Ireland Cook and G. Michaelson. Higher order function synthesis through proof planning. In *Proceedings of 16th Annual International Conference on Automated Software Engineering (ASE 2001), San Diego, USA*, pages pp307–310. IEEE Computer Society, November 2001.

[10] D.E. Culler, A.C. Arpaci-Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K.A. Yelick. Parallel Programming in Split-C. In *Supercomputing*, pages 262–273, 1993.

[11] D.E. Culler, S.C. Goldstein, K.E. Schauser, and T. von Eicken. TAM — A Compiler Controlled Threaded Abstract Machine. *Journal of Parallel and Distributed Computing*, 18:347–370, June 1993.

[12] B. Goldberg. Multiprocessor Execution of Functional Programs. *Intl. J. of Parallel Programming*, 17(5):425–473, October 1988.

[13] K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999.

[14] P.G. Harrison and M.J. Reeve. The Parallel Graph Reduction Machine, ALICE. In *Workshop on Graph Reduction*, volume 279 of *LNCS*, pages 181–202, Santa Fe, NM, USA, September/October 1986. Springer-Verlag.

[15] G. Hogen and R. Loogen. Efficient Organization of Control Structures in Distributed Implementations. In *CC'94 — International Conference on Compiler Construction*, volume 786 of *LNCS*, pages 98–112. Springer Verlag, 1994.

[16] H. Kingdon, D.R. Lester, and G. Burn. The HDG-machine: a Highly Distributed Graph-Reducer for a Transputer Network. *Computer Journal*, 34(4):290–301, 1991.

[17] M. Lauer. Computing by Homomorphic Images. In *Computer Algebra — Symbolic and Algebraic Computation*, pages 139–168. Springer-Verlag, 1982.

[18] H-W. Loidl. The Virtual Shared Memory Performance of a Parallel Graph Reducer. In *CCGrid 2002 — Intl. Symp. on Cluster Computing and the Grid*, pages 311–318, Berlin, Germany, May 2002. IEEE Press.

[19] H-W. Loidl, F. Rubio Diez, N.R. Scaife, K. Hammond, U. Klusik, R. Loogen, G.J. Michaelson, S. Horiguchi, R. Pena Mari, S.M. Priebe, A.J. Rebon Portillo, and P.W. Trinder. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 16(3), 2003.

[20] R. Loogen, H. Kuchen, and K. Indermark. Distributed Implementation of Programmed Graph Reduction. In *PARLE 89 — Conf. on Parallel Architectures and Languages Europe*, volume 365 of *LNCS*, pages 136–157. Springer-Verlag, 1989.

[21] D.K. Lowenthal, V.W. Freeh, and G.R. Andrews. Using Fine-Grain Threads and Run-Time Decision-Making in Parallel Computing. *Journal of Parallel and Distributed Computing*, 37(1):41–54, 1996.

[22] E. Methot, M. Feeley, and B. Gendron. CPar: a parallel language for divide and conquer parallelism. In *CIC'00 — Intl. Conf. on Communicating in Computing*, pages 255–258, Las Vegas, USA, June 2000. CSREA Press.

[23] G. Michaelson and N. Scaife. *Skeletons and Patterns for Parallel and Distributed Computing*, chapter Skeleton Realisations from Functional Prototypes, pages 129–153. Wiley, 2002.

[24] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested Algorithmic Skeletons from Higher Order Functions. *Parallel Algorithms and Applications*, 16:181–206, 2001. Special Issue on High Level Models and Languages for Parallel Processing.

[25] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

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

[27] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs. In *IEEE Aerospace*, 1997.

[28] C.A. Ruggiero and J. Sargeant. Control of Parallelism in the Manchester Dataflow Machine. In *FPCA'87 — Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 1–15, Portland, Oregon, September 14–16, 1987. Springer-Verlag.

[29] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for haskell. In *FPCA'93 — Functional Programming Languages and Computer Architecture*, pages 106–116, Copenhagen, Denmark, 1993.

[30] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[31] N. Scaife, G. Michaelson, and S. Horiguchi. Comparative Cross-Platform Performance Results from a Parallelizing SML Compiler. In *IFL'01 — Intl. Workshop on the Implementation of Functional Languages*, volume 2312 of *LNCS*, pages 138–154, Stockholm, Sweden, Sep 24–26, 2001.

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

[33] P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *PLDI'96 — Programming Language Design and Implementation*, pages 78–88, Philadephia, USA, May 1996. ACM Press.

[34] P. Wegner. *Programming Languages, Information Structures and Machine Organisation.* McGraw-Hill, New York, 1971.

[35] Glasgow Parallel Haskell. WWW page, January 2001. <URL:`http://www.cee.hw.ac.uk/~dsg/gph/`>.