Implementation Designs for Parallel Functional Languages

Hans-Wolfgang Loidl[1], Norman Scaife[2], Susumu Horiguchi[2], Greg Michaelson[3] and Phil Trinder[3]

[1] Institut für Informatik, Ludwig-Maximilians Universität, D 80538 München, Germany; hwloidl@tcs.informatik.uni-muenchen.de
[2] School of Information Science, Japan Advanced Institute for Science and Technology, Tatsunokuchi, Ishikawa, 923-1211, Japan; {norman,hori}@jaist.ac.jp
[3] School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh EH14 4AS, Scotland; {greg,trinder}@macs.hw.ac.uk

Abstract. Parallel functional languages (PFLs) provide high-level coordination using a sophisticated implementation that automatically manages low-level coordination details. The primary coordination aspects to be managed are threads, storage and communication. We compare and contrast two approaches, GpH which uses dynamic management and PMLS which is mostly static. The approaches are compared using careful measurements of three simple programs on a Beowulf cluster and a SunSMP machine. The comparison gives some useful insights into the costs and benefits of alternative PFL implementation designs.

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, until recently the additional complexity of managing high-level coordination on a range of parallel architectures has hampered the development of mature parallel functional language implementations. However, robust, portable implementations like GpH [16], PMLS [11], and Eden [3] 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.

This paper extends our previous work [9] by precisely quantifying the relative costs and benefits of static versus dynamic coordination management for GpH and PMLS on a distributed memory Beowulf. While it may not be surprising that the highly-dynamic nature of managing parallelism in GpH is a better match for irregularly-parallel applications, overall, despite the fundamentally different implementation approaches, the comparative results between GpH and PMLS are similar. This indicates that the coordination structure of the algorithm is far more important than the machine characteristics.

The paper is structured as follows. Section 2 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 8 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 [15], 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 [5].

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 [7].

3.1 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 (for example Cilk [2]). 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 TAM [6] system, which supports totally independent tasks based on function calls, several threads within each task and specialised inlets for handling the arrival of new data, with the guarantee that no blocking occurs within an inlet. Clearly, 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 this context, the threads in GpH are relatively 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.

3.2 Storage Management Model

The storage management model deals with the distribution of the heap amongst 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.

Early machines like the HDG machine [8] used a *linked list of packets* which are 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 [3], 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, 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.

3.3 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 local or remote evaluation. 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 mechanism while PMLS relies 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 perform.

4 The Implementation of GpH

4.1 Overview

In this section we locate the GUM system, which implements GpH, 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 [16].

4.2 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 [12].

In GUM computations are represented by graph structures. 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 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.

4.3 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. 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.

4.4 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 pre-loaded on each PE.

5 The Implementation of PMLS

5.1 Overview

The PMLS philosophy is to perform many analyses at compile time to enable the static allocation of high-granularity threads to processors. The PMLS Compiler [10] 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 model* 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 [1] for the front end, because it provides an evaluator for SML based closely upon the structural operational semantics; Objective Caml [4] 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 as general-purpose communication layer, ease portability of the system, which now runs on several different parallel architectures.

5.2 Thread Management Subsystem

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 *thread management model* encoded in the implementations of PMLS' `map` and `fold` skeletons provides novel ways of handling nested constructs, while still using 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 an efficient method of nested initialisation. A complication 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. PMLS generally achieves good speedups for small numbers of processors but scalability is limited by this bottleneck.

5.3 The Memory Management Subsystem

In contrast to GpH, PMLS uses disjoint heaps and all data that is needed for the computation in a skeleton has to be explicitly sent at the beginning of the computation. Thereafter, each thread is a sequential computation running on a single processor with a single heap and stack.

5.4 The Communication Subsystem

Again in contrast to GpH, PMLS distributes copies of data for local manipulation without sharing. 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.

6 Performance Measurements

6.1 Program Descriptions

For the measurements we use three programs, `sumEuler`, `linSolv` and `raytracer`. `sumEuler` computes the sum of Euler numbers over a list of integers. Its computational structure is fold-of-map such that the Euler numbers are computed by the `map` and are then summed by the `fold`.

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:

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

A nested PMLS implementation can be obtained by representing the integers as lists of datatypes (`EI`) and manually splitting this list. This results in a nested `map` within a `fold`:

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

`linSolv` and `raytracer` are discussed in detail in [9] 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. 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, a number of rows or rectangular areas is processed by one thread/skeleton to increase granularity.

6.2 Results

Figure 1 summarises runtimes and speedups for `sumEuler`, `linSolv` and `raytracer`, under GpH and PMLS on a Beowulf. For PMLS we additionally report the performance on a SunSMP. The GpH port to the SunSMP is currently not reliable enough to report these results.

All measurements have been performed on a 32-node Beowulf cluster 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 Starfire with 900MHz UltraSPARC 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.

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

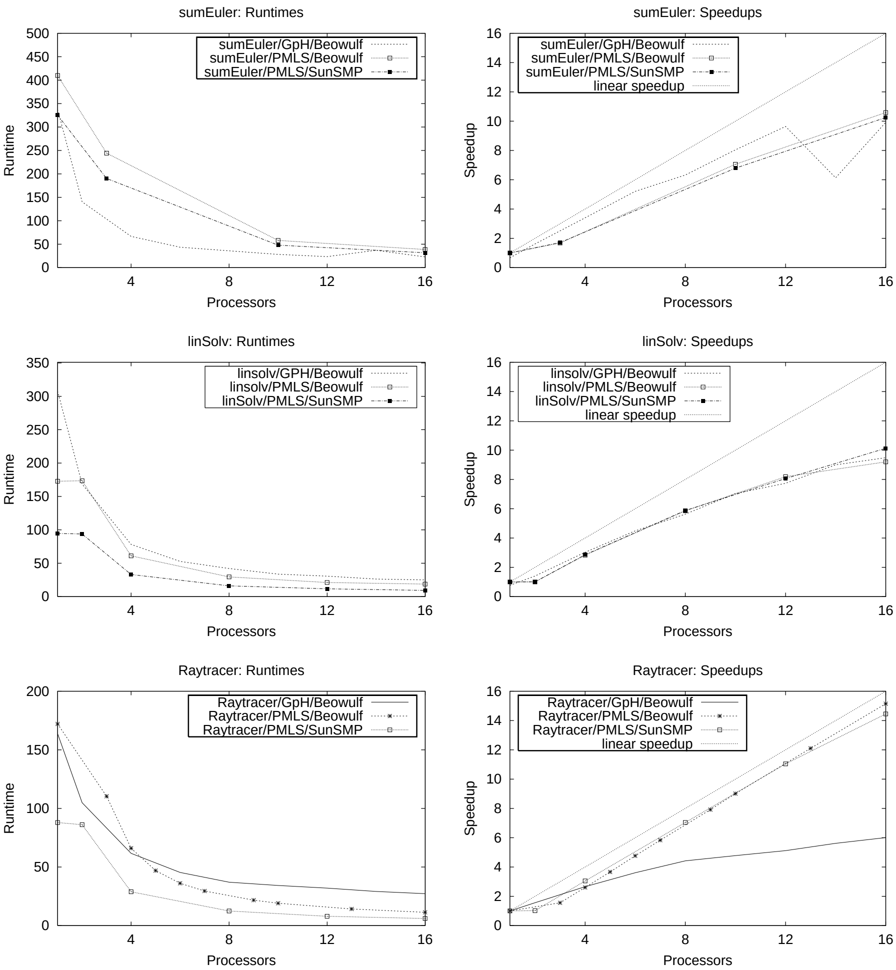

Fig. 1. Runtimes and speedup for `sumEuler`, `linSolv` and `raytracer`

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, although even PMLS with its static work distribution has shown this kind of behaviour with irregular problems.

For `linSolv`, PMLS achieves significantly better single processor performance (172.82s vs 307.77s), due to an algorithmic difference. The PMLS version uses a slightly more efficient algorithm in the forward substitution phase of com-

Table 1. Performance for `sumEuler` (PMLS and GpH)

	PEs	Time (secs)	Speedup $(\frac{\text{seqRT}}{\text{parRT}})$	Allocation (MB)	Fragmentation (std dev)	Residency (MB)	GC %
PMLS	1	409.90	1.00	6	—	—	0.01%
	3	244.16	1.68	251	323.3	—	0.25%
	10	58.05	7.06	39	59.7	—	0.21%
	16	38.71	10.59	21	30.4	—	0.19%
GpH	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%

Table 2. Performance for `linSolv` (PMLS and GpH)

	PEs	Time (secs)	Speedup $(\frac{\text{1PERT}}{\text{parRT}})$	Allocation (MB)	Fragmentation (std dev)	Residency (MB)	GC %
PMLS	1	172.8	1.0	4658	—	—	18.5
	2	173.5	1.0	2998	2347.4	—	9.1
	4	61.0	2.8	1711	536.7	—	12.7
	8	29.5	5.9	1158	159.8	—	13.2
	12	21.1	8.2	981	86.2	—	12.8
	16	18.8	9.2	894	63.2	—	10.9
GpH	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

puting a solution in one homomorphic image (see [9][Sec 4.3.4]). The speedups are virtually identical, however, showing that GpH's use of parallelism is slightly more efficient.

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, so that only one processor is used at this stage. The PMLS skeleton is better tuned to avoid such a bottleneck but requires careful selection of granularity. 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.

Table 3. Performance for `raytracer` (PMLS and GpH)

	PEs	Time (secs)	Speedup ($\frac{1PERT}{parRT}$)	Allocation (MB)	Fragmentation (std dev)	Residency (MB)	GC %
PMLS	1	242.06	1.00	17275	—	—	8.4
	2	249.07	0.97	9879	5231.8	—	3.0
	4	80.06	3.02	4388	686.5	—	5.4
	8	34.97	6.92	2170	106.4	—	6.2
	12	21.77	11.12	1444	37.3	—	6.7
	16	16.99	14.24	1082	17.9	—	6.7
GpH	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

Tables 1, 2 and 3 provide a more detailed view of the main performance characteristics for both systems on the Beowulf architecture. These are discussed in the following sections.

6.3 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 conceptually equivalent to the generation of a thread for GpH. The skeleton overheads are slightly lower due to the static analysis which mitigates processor allocation costs and reduces communications.

Top-level skeleton initialisation times in PMLS are roughly constant per skeleton instance, usually about $50\mu s$. For non-nested implementations this time is not significant but for nested cases 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 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 incurred once when starting the runtime system. The thread creation overhead in GpH with a default stack size of 1KB is about $8\mu s$.

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. Most notably a higher percentage of potential parallelism is exploited when more PEs become available. Under these circumstances a high single processor efficiency and low garbage collection times are observed. Total heap allocation is also very low compared to the other programs discussed in the following sections. For PMLS granularity has to be set statically at the start of execution.

For the programs used here, 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.

6.4 Storage Management

Tables 1, 2, 3 and 5 also show a summary of the garbage collector behaviour. Column 5 shows the total amount of allocated heap data, Column 6 shows the heap fragmentation as the standard deviation of the total amount of allocated heap taken over all PEs and Column 7 the heap residency, i.e. the maximum amount of live heap taken over all garbage collections. Heap residency figures are not available for PMLS. All values are averaged over all PEs. In most cases the allocation in the GpH version 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 [13]. This is underlined by the low values for the heap residency in Column 7, which show that despite the large amount of allocation at each garbage collection only a small part of the heap contains live data. The high percentage of execution time spent on garbage collection, shown in Column 8, reflects the usage of a two-space garbage collector in GpH. This is an implementation restriction in the current system we plan to address in the near future. However, the overall absolute speedup values show that even with such a high penalty on the storage management overhead for GpH, the overall performance is still comparable to those achieved for PMLS.

The GpH results for `linSolv` in Table 1 show a roughly linear decrease in the heap allocated (per PE) as the number of PEs increases. For GpH with higher numbers of PEs some data needs to be duplicated in order to generate more threads. Overall, GpH is conservative in avoiding any duplication of work, but allows for the duplication of data needed as input for new threads. For PMLS there is a large increase from 1 to 3 processors but then the total heap allocation reduces for larger numbers of PEs. The crucial factor here is the amount of memory allocated during the execution of an instance function relative to the default allocation size. For fine granularity the entire computation can fit within the initial heap. This is constant across PE numbers resulting in a per processor decrease. These observations are reinforced by the fragmentation figures which measure, roughly, how evenly the heap allocation is spread across PEs. The degree of memory fragmentation drops even more rapidly for PMLS than for GpH, however, high fragmentation is less problematic in PMLS since it does

Table 4. Communication for (PMLS and GPH) on Beowulf

		PMLS Root			PMLS Nested			GPH		
		Sent+Recvd		Comm	Sent+Recvd		Comm	Sent+Recvd		Comm
	PEs	Count	KB	Msg/s	Count	KB	Msg/s	Count	KB	Msg/s
`sumEuler`	3	76	98.1	0.3	8524	93.7	34.9	443	129.3	4.3
	10	72	100.6	1.2	4406	48.3	75.9	573	95.5	12.8
	16	68	100.5	1.7	2504	27.4	64.7	581	89.3	16.2
`linSolv`	2	160	176.5	0.9	—	—	—	126	25.2	0.5
	4	160	176.5	2.6	—	—	—	907	194.1	12.5
	8	160	176.5	5.4	—	—	—	1001	253.6	21.6
	12	160	176.5	7.6	—	—	—	1091	263.5	34.9
	16	160	176.5	8.5	—	—	—	1114	272.1	38.5
`raytracer`	2	348	1417	1.40	—	—	—	3991	1764.9	31.9
	4	344	1406	4.30	—	—	—	5543	2171.2	77.8
	8	336	1384	9.61	—	—	—	12710	4537.5	222.4
	12	328	1354	15.07	—	—	—	19135	6656.1	392.6
	16	320	1321	18.83	—	—	—	22654	7789.6	482.7

not directly affect the amount of communication, in contrast to GPH with its virtual shared heap.

6.5 Communication

Table 4 summarises the communication behaviour of `sumEuler`, `linSolv` and `raytracer`. 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 PMLS data in Table 4 splits the `sumEuler` figures into those for the root and node skeletons.

Comparing the PMLS and GPH results for `sumEuler` in Table 4 we find that the communication in GPH is usually more fine grained and more frequent: both the number of packets and the communication degree exceed the corresponding values for PMLS. This reflects the fact that GPH uses demand driven communication, where only the immediately needed portion of a data structure is transfered. In contrast, PMLS transmits all required free values during skeleton initialisation. 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 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 for which the communications degree increases proportionally with the number of PEs.

The high number of messages for `raytracer` is partly due to the splitting of large data structures into several packets that is performed in GPH. Conse-

Table 5. Performance and Communication for PMLS on SunSMP

	PEs	Time (secs)	Speedup ($\frac{1PERT}{parRT}$)	Allocation (MB)	Fragmentation (std dev)	GC %	Sent+Recvd Count	KB	Comm Msg/s
`sumEuler`	1	325.29	1.0	6	—	0.0%	—	—	—
	3	190.23	1.7	1078	1737.4	0.1%	76	95.8	0.40
	10	47.94	6.8	140	369.4	0.1%	72	98.2	1.50
	16	31.75	10.3	66	202.2	0.1%	68	98.2	2.14
`linSolv`	1	94.60	1.0	4658	—	11.2	—	—	—
	2	93.83	1.0	3869	1115.7	5.4	160	172.4	1.71
	4	33.10	2.9	1889	181.4	7.7	160	172.4	4.83
	8	16.10	5.9	1202	42.6	8.2	160	172.4	9.94
	12	11.74	8.1	1002	21.1	7.6	160	172.4	13.62
	16	9.34	10.1	907	21.8	7.4	160	172.4	17.14
`raytracer`	1	87.95	1.00	17295	-	7.20	—	—	—
	2	86.13	1.02	9741	10687.6	2.96	350	1414.5	4.06
	4	28.85	3.05	4545	2442.7	4.59	350	1414.5	12.13
	8	12.49	7.04	2213	733.8	5.52	350	1414.5	28.02
	12	7.96	11.04	1463	381.9	6.14	350	1414.5	43.95
	16	6.08	14.45	1094	242.7	6.80	350	1414.5	57.52

quently, 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. Although PMLS can also split messages into smaller portions a more efficient solution is to simply increase the maximum buffer size for MPI, about 1MB was required for the `raytracer` program.

7 Sun SMP

Table 5 shows performance results for PMLS versions of the three test programs 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 consistent performance on different architectures. This issue is discussed in more detail in [14]. One notable difference is 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.

The communication data in the last three columns show, that 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 from 64.7 to 72.7 (for 16 PEs in nested nodes) only because the SunSMP uses faster processors.

8 Conclusion

This paper explores 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. We have also measured the PMLS versions on a SunSMP.

The main contribution of this paper is to quantify the impact of the different design decisions, and classify them by the key components of the implementations, namely the thread management system, the storage management system and the communication system. More specifically the different dynamic properties of the two systems are as follows.

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.

References

1. L. Birkedal, N. Rothwell, M. Tofte, and D.N. Turner. The ML Kit (Version 1). Technical Report 93/14, Dept. of Computer Science, Univ. of Copenhagen, 1993.
2. 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.
3. S. Breitinger, R. Loogen, Y. Ortega Mallén, and R. Peña 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.
4. E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, Paris, April 2000.
5. Murray I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. The MIT Press, Cambridge, Massachusetts, 1989.
6. 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.
7. K. Hammond and G. Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999.
8. 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.
9. 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.
10. G. Michaelson and N. Scaife. *Skeletons and Patterns for Parallel and Distributed Computing*, chapter Skeleton Realisations from Functional Prototypes, pages 129–153. Wiley, 2002.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.