Parallel Cost Centre Profiling

Kevin Hammond
Division of Computer Science, University of St. Andrews
email: `kh@dcs.st-and.ac.uk`

Hans-Wolfgang Loidl, Phil Trinder
Dept. of Computer Science, University of Glasgow
email: `{cvh,hwloidl,trinder}@dcs.gla.ac.uk`

September 10, 1997

Abstract

Good profiling is a major issue in extracting performance from parallel programs. We report on a novel synthesis of sequential cost-centre profiling and state-of-the-art parallel simulation for the pure functional language Haskell that promises to provide detailed and accurate information about parallel Haskell programs. Exploiting simulation also improves the quality of sequential cost centre profiling, though at a significant performance overhead.

1 Introduction

It has become something of a commonplace for us to complain bitterly of the difficulty of understanding the behaviour of parallel functional programs in the presence of lazy evaluation. In order to help overcome this, we have designed a suite of visual profiling and simulation tools, originally aimed specifically at research into granularity issues, but now also aimed at general parallel program development.

Unfortunately, it is still a frequent occurrence for some member of our team to be asked to explain the behaviour of a parallel program by a confused programmer. When this happens, we are forced to rely on our experience of parallel program development and the arcana of the parallel runtime system to provide an intuitive explanation of the program behaviour. Sometimes the intuition is correct. Sometimes, it is not. Usually, it is impossible to tell without experimentation, and the explanation inevitably surprises the programmer!

This paper describes a combination of techniques that we believe will help reduce the intuitive understanding needed to successfully produce parallel functional programs and so make it easier for inexperienced programmers to write their own parallel programs. Both techniques are well-known, but the combination is especially powerful.

The structure of this paper is as follows. Section 2 describes the implementation of the parallel cost centre mechanism. Section 3 gives some simulation results obtained using the simulator and visualisation tools. Finally, Section 4 concludes.

2 Implementation

We have chosen to use the Glasgow Haskell compiler as the basis for our system. This is an especially suitable testbed since it already contains a flexible and fairly accurate parallel simulator, GranSim, as well as an implementation of cost-centre profiling. All that is needed is to combine the two in a sensible way. Importantly for longer-term research, the parallel compilation system also provides a real parallel implementation, GUM, which is potentially capable of supporting cost-centre profiling. This will allow programs to be prototyped using simulated cost-centre profiles and verified using actual parallel runs.

2.1 Cost Centre Profiling

The Glasgow implementation of the cost centre profiling mechanism is a fairly simple one. A register is used to record the current cost centre throughout the program execution. This is updated to reflect the current activity, perhaps set by explicit `scc` constructs in the program source. At regular intervals a counter is incremented corresponding to the current cost centre. Counters may also be incremented to record the number of times a closure is entered and similar low-level profiling information. At the end of the program run, a profile is produced indicating the proportion of the total execution time that was spent executing code attached to each cost centre.

2.2 GranSim

Our simulator, GranSim, is built around the threaded runtime system for the Glasgow Haskell compiler [Pe93], which Mattson has developed for work on concurrency in a sequential setting [FiPe94].

In Mattson's threaded system, there is a single thread pool which is scheduled in a round-robin fashion. There is also a single pool of sparks, which represent potential parallel work, and which are turned into threads as required. If sparked expressions are in weak head normal form (WHNF), then the sparks are discarded without ever becoming threads. Threads context-switch after a pre-defined time interval, when

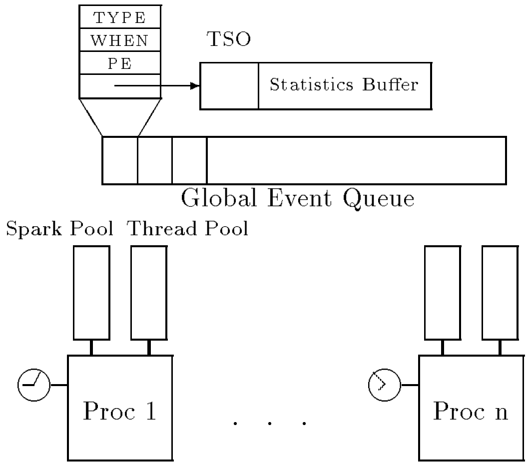

Figure 1: Overall structure of the simulator

they reach a suitably safe point in the computation. This is normally at the next heap allocation, where the thread must be prepared to transfer control to the garbage collector, and thus where registers and heap pointers are in a known state. By setting the timeout to zero, a context-switch can be forced as often as possible.

The overall structure of our parallel simulator is shown in Figure 1. The simulator has multiple thread and spark pools, one of each per processor that is being simulated. Context-switches are forced after each basic block, so that each simulated processor is scheduled as frequently as possible. Unlike Mattson's system, our normal scheduling algorithm is *unfair*: each processor executes the same thread for as long as possible (typically until it communicates). In the future we plan to investigate different scheduling strategies.

In the simulator, each thread has a *statistics buffer* attached to its thread state object (TSO). This records:

- which spark created the thread;
- when it started;
- whether it was produced by an exportable spark;
- whether it has been migrated;
- how many exportable and non-exportable sparks it has generated;
- how many heap allocations it has performed;
- how many basic blocks it has executed;
- how long overall it has been blocked;

- if it is currently blocked, when it became blocked;
- how long it has spent reading remote data.

When a thread terminates, the contents of this buffer and the time it terminated are dumped to a trace file. If a more detailed trace is required then important events, such as communication between two processors, may also be dumped to the trace file as they occur.

2.2.1 Global events

We introduce a global event queue, with entries of the form: <Type, When, Proc, TSO>. When gives the time at which the event should occur. Proc identifies the processor on which the event should occur. TSO is an optional pointer to a *thread state object* which identifies a thread on Proc when this is relevant to the event.

The possible event types in the Type field are:

Continue:	Continue running a thread
Start:	Turn a spark into a new thread
Resume:	Resume a blocked thread
FindWork:	Search for work
MoveSpark:	Move a spark to a new processor
MoveThread:	Move a thread to a new processor
FetchNode	Fetch a node
FetchReply	A node has been fetched

The events in the event queue are ordered by time. There is a global notion of time, which we use to represent instruction costs.

2.3 Runtime System

The changes to the runtime system that are needed to support cost-centre profiling in GranSim are straightforward in principle, essentially involving using all simulator and cost-centre profiling code simultaneously. It is also necessary to save and restore the current cost centre via the thread state object (which contains the thread's context including all registers used by the thread) whenever a new thread is scheduled, whether this is on the currently simulated processor or some other processor.

2.4 Simulator Transcript

The format of the simulator transcripts has been extended to incorporate cost centre information. New events are recorded in the transcript when the cost centre is changed using either an explicit `scc` construct or via an internal change of cost centre. In addition, the current cost centre is recorded when a new task is created, and should also be recorded for sparks.

2.5 Visualisation Tools

The visualisation tools are based on existing GranSim profiling tools, modified to account for changing cost centres. The tools allow individual profiles to be created for each task queue against time: running, runnable, blocked, fetching, or migrating. These profiles are broken down by cost-centre, so giving an accurate picture of system activity. This is unfortunately an expensive process in the prototype system, though we hope to make this more efficient in future.

3 Exploiting Cost-Centre Profiling

This section gives examples of the kind of profile that can be produced using the profiling tools. We first present some simple examples, then consider alpha-beta search. Eventually there will be more substantial examples in this section.

3.1 Simple Examples

This section will contain some simple examples. For now Figure ?? shows the profile for the simple producer-consumer program given below, with an input of 10.

3.2 Alpha Beta Search

The Alpha-Beta search algorithm typifies artificial intelligence applications based on solving search problems. Normally it is used by game-playing programs to find the best next move by generating all possible moves up to a certain depth, applying a static evaluation function to each of the leaves in this search tree, and combining the result by picking the best move for the player assuming that the opponent picks the worst move for the player.

We discuss two versions of the Alpha-Beta search algorithm: a *simple* version, and a *pruning* version.

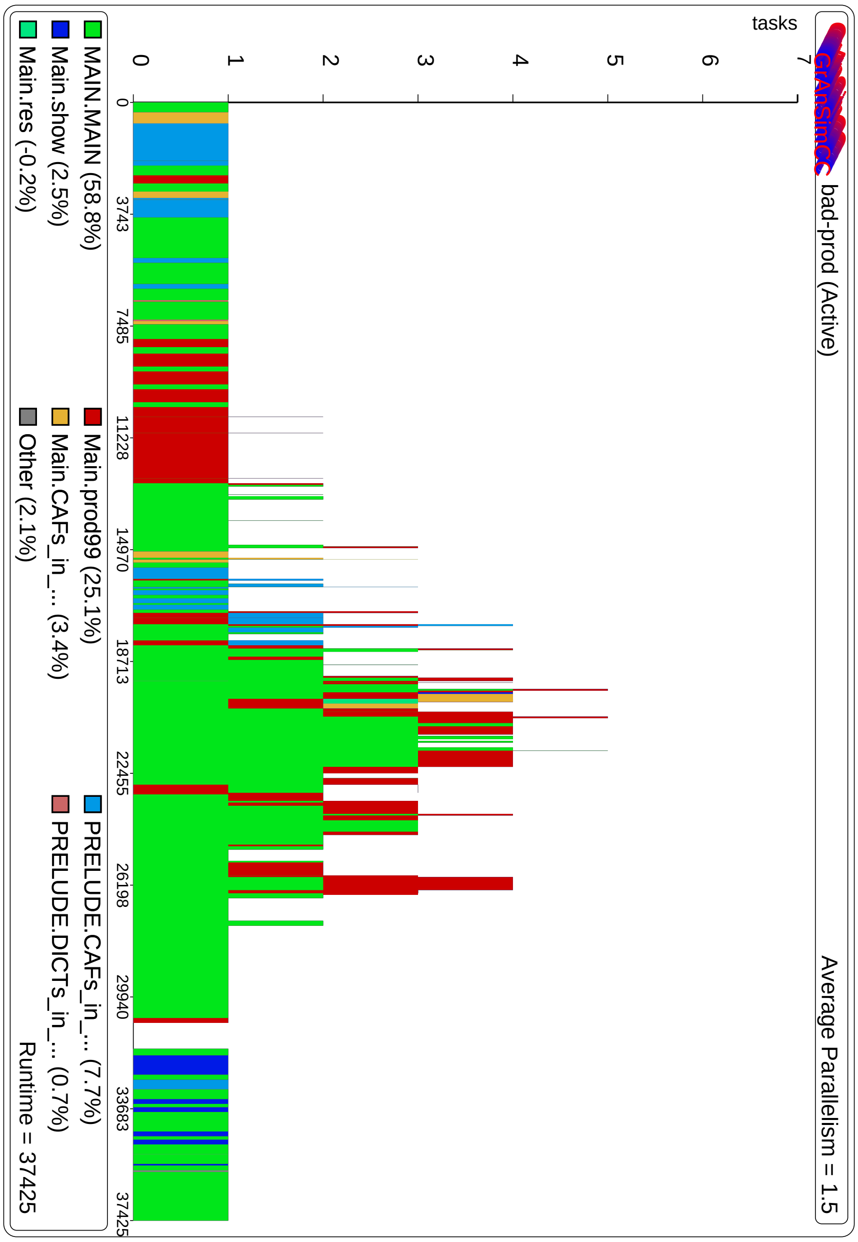

Figure 2: Strategy for a parallel pruning version with a static force length

Both versions are basically Haskell versions of the Miranda[1] code presented by John Hughes [Hug89] in order to demonstrate the strengths of lazy functional languages. We have implemented two simple games (tic-tac-toe and escape) in combination with a generic Alpha-Beta search algorithm. An interesting aspect of this algorithm is the fact that the pruning version relies on laziness to prune the search tree based on intermediate results of the computation. This is crucial for the efficiency of the sequential algorithm. In

[1] Miranda is a trademark of Research Software Ltd.

this section we parallelise both versions and study the parallel runtime behaviour. The algorithm is more fully described elsewhere [LoTr97].

3.2.1 Simple Algorithm

In the simple algorithm each possible next move is evaluated independently yielding a compositional structure of the algorithm. The result is either the maximum (player's move) or the minimum (opponent's move) of the evaluations of all next positions. This algorithm can be very naturally described as a sequence of function compositions performing the following tasks:

1. Construct a tree with positions as nodes and all possible next moves as subtrees.
2. Prune the tree, which might be infinite at this stage, to a fixed depth to bound the search.
3. Map a static evaluation function over all nodes of the tree.
4. Crop off subtrees from winning or losing positions. If such a position is found it is not necessary to search deeper in a subtree.
5. Finally, pick the maximum or minimum of the resulting evaluations in order to determine the value of the current position via `mise f g`. The functions `f` and `g` represent the combination functions for the two players and alternate when traversing the tree.

Dynamic Behaviour. The compositional nature of this algorithm makes parallelisation rather easy. For both versions of the algorithm we study the following three sources of parallelism:

Parallel Static Evaluation Function. The idea of a parallel static evaluation function is to reduce the costs of the function, which will be mapped to the leaves of the pruned search tree. This only makes sense for a rather time consuming static evaluation function, otherwise it creates a lot of fine-grained parallelism. However, an underlying assumption of the Alpha-Beta search algorithm is that the static evaluation function can be very simple when using a tree search structure to determine the best value. In our example implementations, the static evaluation function computes the distance of the current position to a set of known winning positions. The parallel version computes all distances in parallel.

Parallel Higher-order Functions over Trees. Parallelising the definitions of some higher-order functions is a bottom-up approach, which can be used for the parallelisation of many functional programs. In this case we use a parallel version of a map function over search trees. However, our measurements below show, that without any knowledge about the context in which these higher-order functions are used a lot of redundant work may be generated resulting in extremely poor parallelism.

Data Parallelism over All Possible Next Moves. In a data parallel approach the goal is to evaluate all possible next moves in parallel. This turns out to be the best source of parallelism in particular for such a compositional algorithm with no dependencies between the evaluations of the subtrees. A simple `parMap rnf` strategy can be used to capture the dynamic behaviour of this function. The only necessary change in the algorithm affects the `mise` function in Stage 5 of the algorithm:

```
{code-mise.ps}
%-- This does simple minimaxing without pruning subtrees based on intermediate
%-- evaluations (i.e. purely compositional)
%mise :: Player -> Player -> (Tree Evaluation) -> Evaluation
%mise f g (Branch a []) = a
%mise f g (Branch _ l) = foldr f (g OWin XWin)
%                                (parMap rnf (mise g f) l)
```

Either the binary `max` or `min` function is folded over the list of results from the subtrees. Note that the functions `f` and `g` change position in the recursive call to record the switch in turns.

3.2.2 Pruning Algorithm

The simple algorithm described in the previous section lacks one crucial optimisation of the Alpha-Beta search: the pruning of subtrees based on intermediate results. The pruning algorithm returns an increasing list (player's move) of approximations with the exact value as last list element rather than a single value. The key function in combining the lists of the subtrees to a result list is the `minleq` function, which tests whether the best result of the list l is better than the intermediate result x. Or more formally: `minleq` $l\ x \Leftrightarrow x \leq \min\ l$. Since `minleq` works on decreasing lists it can stop examining the list as soon as it finds a value less than x. Thus, laziness is used to ignore parts of the list of approximations, which amounts to pruning subtrees in the search tree. A complete description of this lazy functional pruning algorithm can be found in [Hug89].

Dynamic Behaviour. Unfortunately, the pruning version seriously complicates the parallelisation of the algorithm. We have already seen in the simple algorithm that the most promising source of parallelism is the parallel evaluation of all next positions. However, using a simple `parList rnf` strategy over all next positions is no longer advisable, since this might result in a lot of redundant work, if many subtrees can be pruned. The measurements of the data parallel strategy on the pruning algorithm in Table ?? show a rather high degree of redundant work. In fact, in the data parallel strategy on Position III parallel simple version is even faster than the highly speculative pruning version of the algorithm!

A better approach for parallelisation is to force only an initial segment in the list of possible next positions. We call the length of this segment the "force length". We have experimented with static force lengths

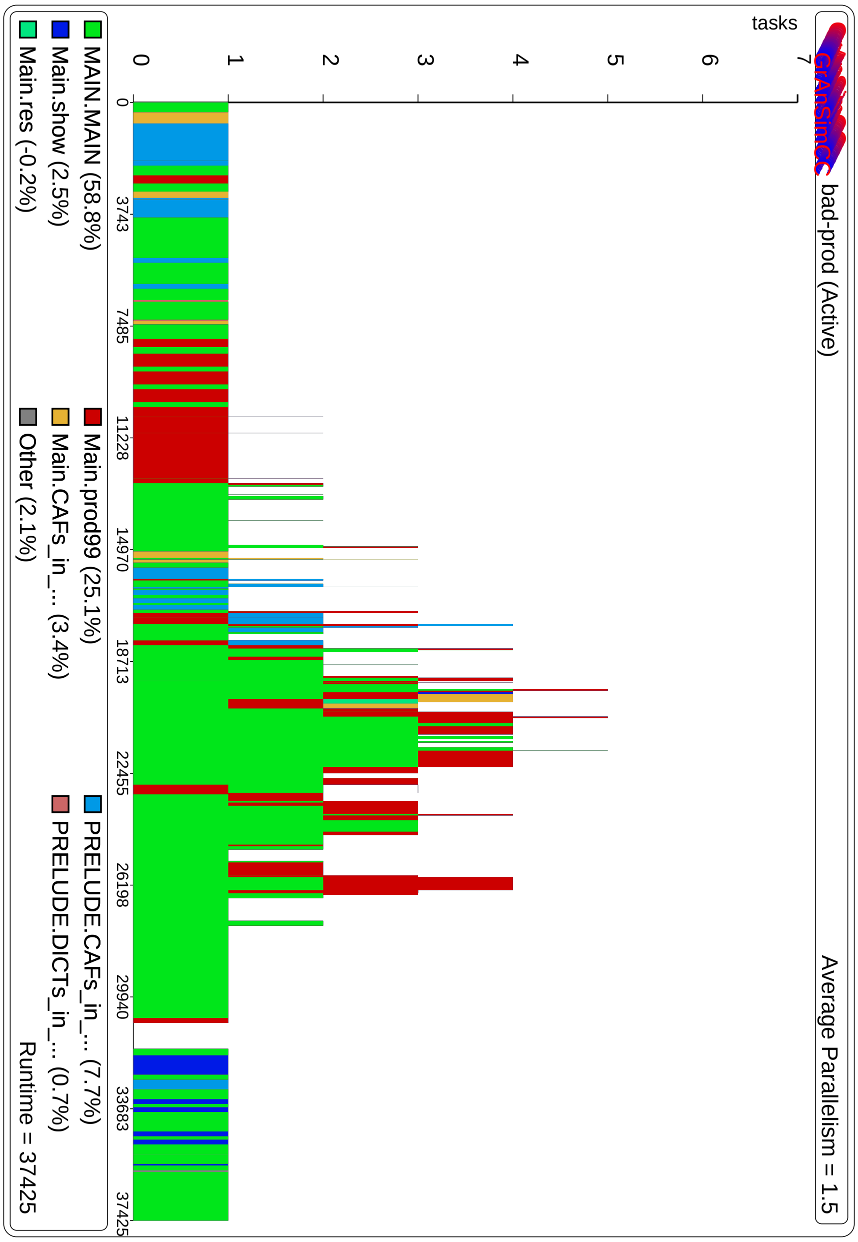

Figure 3: Minimax, Depth 3, Overall Activity Profile

as well as dynamic force lengths that depend on the level in the search tree. To date the best results have been obtained from using a static force length as shown in the code in Figure 1. Note that the force length represents a trade-off between increasing the degree of parallelism and reducing the total amount of work being done.

Figures 3.2.1–3.2.1 show the overall activity and cost centre profiles for the final version of minimax at a

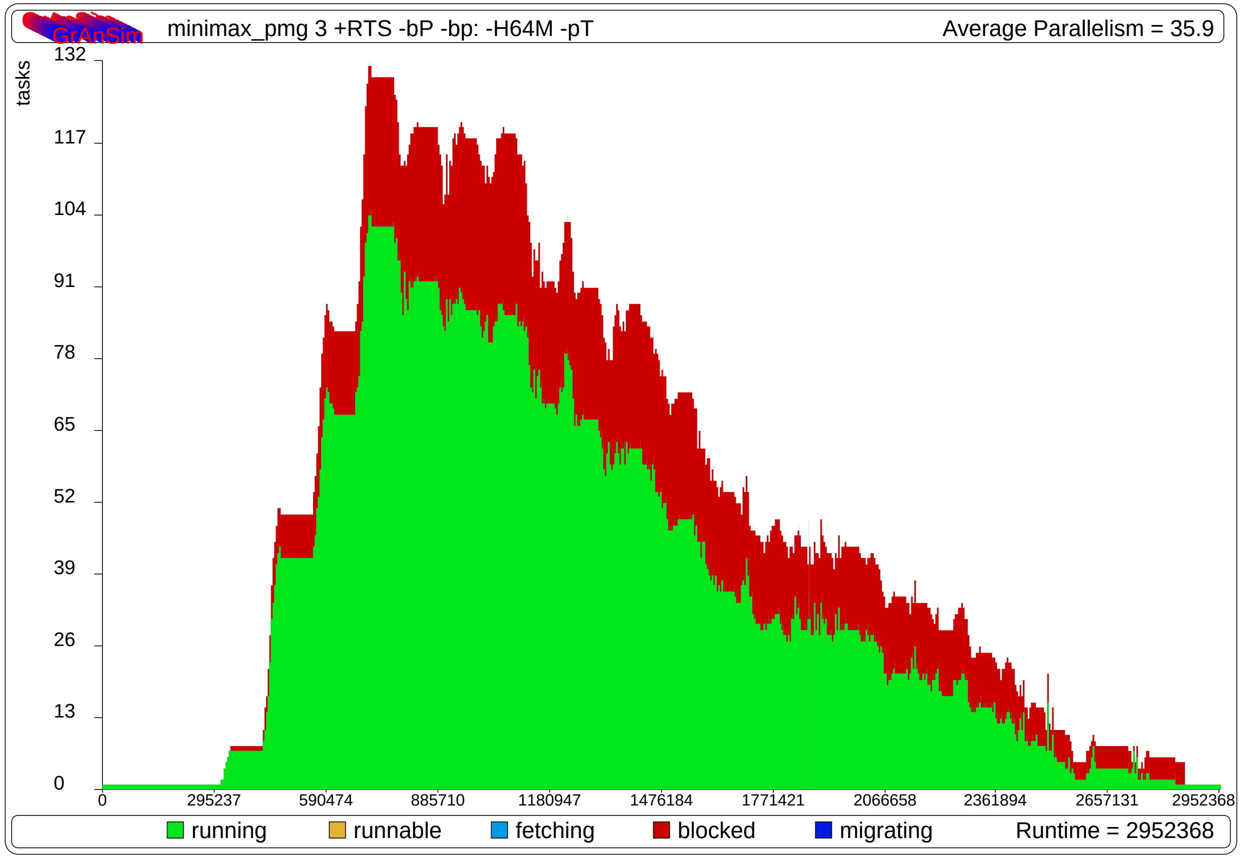

Figure 4: Minimax, Depth 3, Cost Centre Profiling

search depth of 3.

4 Conclusions and Further Work

Although we have not yet developed the system fully, the combination of technologies appears to be very promising. We are in the process of using this tool to help parallelise some real applications. One complication we have encountered is in excessive attribution of cost to the MAIN.Main cost centre. We are investigating this at present. The current system works only with our parallel simulator – we have not yet compared simulation results with those from conventional cost centre profiling on GUM [THM+96].

In addition to its main use for parallel profiling, we expect the system will also have some use for sequential profiling. Compared with conventional cost-centre profiling, the simulation delivers a more accurate picture of system activity, but at a considerable time penalty. The accuracy derives from using event-driven logging rather than interval-based sampling. Some inaccuracy is, however, introduced by the CPU parameters to the simulation. Sample timings show that this is within 10

One of our main interests lies in using this kind of profiling to determine what strategic code is used at what point in the user program. In this case, cost-centres must be attached to strategies, and possibly tracked dynamically. It is not clear whether additional support is required for this purpose. Dynamic support may also be required to track cost centres for individual tasks.

Our ultimate goal is to develop a suite of tools and methodologies whose purpose is to simplify the task of parallel programming. We feel the tool described here represents a valuable step towards achieving this goal.

References

[FiPe94] SO Finne and SL Peyton Jones. Supporting Concurrent State Threads. In *Glasgow Workshop on Functional Programming*, Ayr, Scotland, September 12–14, 1994. Springer.

[HLP95] K Hammond, H-W Loidl and AS Partridge, "Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell". In *HPFC'95—Conf. on High Performance Functional Computing*, pp. 208-221, Denver, CO, April 10-12, 1995.

[HaPe90] K. Hammond and SL. Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *Second Int. Workshop on the Parallel Implementation of Functional Languages*, pp. 51–72, Univ. of Nijmegen, June 1990.

[HPW92] P Hudak, SL Peyton Jones, and PL Wadler (eds.), "Report on the Programming Language Haskell Version 1.2". *ACM SIGPLAN Notices* **27**(5), May 1992.

[Hug89] RJM Hughes, "Why Functional Programming Matters", *Computer Journal*, 1989.

[Juna97] S Junaidu *A Parallel Functional Language Compiler for Message Passing Multicomputers*. Forthcoming PhD thesis, School of Mathematical and Computational Sciences, St Andrews University, Scotland, 1997.

[LoTr97] H-W Loidl and PW Trinder "Engineering Large Parallel Programs". To appear in *Proc. IFL '97*, St Andrews, September, 1997.

[Peyt92] SL Peyton Jones, "Implementing Functional Languages on Stock Hardware: the Spineless Tagless G-machine". *Journal of Functional Programming*, **2**(2), pp. 127-202, 1992.

[Pe93] SL Peyton Jones, C Hall, K Hammond, WD Partain, and PL Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology Technical Conference*, Keele, 1993.

[Roe91] P Roe. *Parallel Programming using Functional Languages.* PhD thesis, Dept. of Computing Science, Univ. of Glasgow, February 1991.

[RuWa93] C Runciman and D Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines). In JT O'Donnell and K Hammond (eds.), *Glasgow Workshop on Functional Programming*, pp. 236–251, Ayr, Scotland, July 5–7, 1993. Springer.

[SaPe95] PM Sansom and SL Peyton Jones, "Time and Space Profiling for Non-strict Higher Order Functional Languages". In *Proc. 22nd ACM Symposium on Principles of Programming Languages*, San Francisco, Carlifornia, January 1995.

[THL+96] PW Trinder, K Hammond, H-W Loidl, SL Peyton Jones, and J Wu, "A Case Study of Data-intensive Programs in Parallel Haskell". In *Proc. 1996 Glasgow Workshop on Functional Programming 1996*, Ullapool, Scotland, July 8-10.

[THM+96] PW Trinder, K Hammond, JS Mattson Jr., AS Partridge and SL Peyton Jones, "GUM: A Portable Parallel Implementation of Haskell". In *PLDI '96 — Programming Languages Design and Implementation*, pp. 78-88, Philadelphia, PA, May 1996.

[THLP98] PW Trinder, K Hammond, HW Loidl and SL Peyton Jones "Algorithm + Strategy = Parallelism". To appear in *Journal of Functional Programming*, 1998.