On the Granularity of Divide-and-Conquer Parallelism

Hans Wolfgang Loidl
Dept. of Computing Science
University of Glasgow
hwloidl@dcs.gla.ac.uk

Kevin Hammond*
Division of Computer Science
University of St. Andrews
kh@dcs.st-and.ac.uk

Abstract

This paper studies the runtime behaviour of various parallel divide-and-conquer algorithms written in a non-strict functional language, when three common granularity control mechanisms are used: a simple cut-off, a priority thread creation and a priority scheduling mechanism. These mechanisms use granularity information that is currently provided via annotations to improve the performance of the parallel programs.

The programs we examine are several variants of a generic divide-and-conquer program, an unbalanced divide-and-conquer algorithm and a parallel determinant computation. Our results indicate that for balanced computation trees a simple, low-overhead mechanism performs well whereas the more complex mechanisms offer further improvements for unbalanced computation trees.

1 Introduction

The overall goal of our research is to improve the efficiency of parallel functional programs by improving the granularity of the threads which are produced during execution. The granularity of a thread is the time required to perform all its computations, not including the overhead of creating the thread and the other overheads imposed by parallel execution, such as communication costs.

We use a non-strict, purely functional language (Haskell) with an evaluate-and-die mechanism of computation [10]. In this model it is possible to dynamically create new subsidiary threads to evaluate sub-expressions that are found to be needed, or to entirely avoid creating threads by absorbing the work which they would have done into a parent thread.

The optimal granularity for all threads is a compromise between minimal parallel overhead and maximal processor utilisation. This should result in the minimum possible runtime for a parallel program on a given parallel machine.

*Supported by an SOED Research Fellowship from the Royal Society of Edinburgh and the EPSRC Parade grant.

Obtaining the optimal thread granularity for a program is a hard problem since it is not only affected by details of the architecture, such as communications latency, number of processors etc., but also by algorithmic details such as communications patterns, which are generally unpredictable. The order in which threads are scheduled can also have a dramatic impact on granularity.

We have chosen to concentrate on divide-and-conquer algorithms since they exhibit interesting parallel behaviour (simple dynamic partitioning for subdivision, but potentially serious bottlenecks in the combination stage). Furthermore, many widely-used algorithms are divide-and-conquer: matrix operations (determinant computation, multiplication etc.), quicksort, alpha-beta search etc. This study thus has considerable practical relevance.

We have chosen to focus on granularity issues and their impact on time performance. Our previous studies have shown that there is a strong correlation between space usage and time performance at the thread level [4]. Overall, however, space usage is likely to be minimised by maximising granularity, though space leaks mean that this will not always be the case. McBurney and Sleep have studied this issue in a functional context [8].

2 The GranSim Granularity Simulator

Because our objective is to obtain results that apply to as many parallel systems as possible, we have chosen to use a simulator, GranSim, to study granularity effects. This simulator delivers realistic results and has been calibrated against several real architectures. The interested reader is referred to [4] for a description of the construction of the simulator, validation results, and studies of various test programs.

We prefer simulation to a more theoretical approach because it gives a more controllable, and more realistic result. By their nature, it is common for analytical approaches to ignore important costs such as communication, or to fail to deal with complex, but significant, interactions such as the exact scheduling algorithm used, or the precise time at which communications occur.

2.1 Parallelism Control

Our basic parallel construct is the one which *sparks* a closure. Sparks are similar to lazy futures [9] in that they could potentially be turned into parallel threads. If so they compute the result and terminate without having to notify the parent thread. It is important to note that this *evaluate-and-die* mechanism [10] dynamically increases the granularity of the threads: a parent process may subsume the computation of a child thread. However, this does not prevent the system from producing many small threads if the overall workload is low. Therefore, our granularity control mechanisms aim at increasing thread size even further.

If and when a spark is created it is placed at the end of the spark queue on the local processor. Idle processors look for work first in their own spark queue and then in those belonging to other processors. In either case, sparks are chosen from the start of the queue.

The basic difference to the lazy task creation model is that the latter does not have to maintain an explicit spark pool. In order to create parallelism work must be stolen from a certain position on the stack. Lazy futures basically indicate such positions. However, the existence of a spark pool makes it easier to attach granularity information to the sparks. As the creation of a spark is rather cheap (putting a pointer to a closure into a queue) we are willing to pay that overhead in order to improve granularity.

We use the following set of annotations to control parallelism:

- `parGlobal n g x y`: a spark is created for closure `x`, with name `n`. Evaluation continues with expression `y`. The `g` field contains granularity information as explained below.
- `parLocal n g x y`: a non-exportable spark is created.
- `parAt n g e x y`: a spark is created on the processor owning closure `e`.
- `seq x y`: `x` and `y` are evaluated in sequence.

2.2 Granularity Control

Based on the information provided by the `g` field of the above annotations, we have studied three granularity control mechanisms:

- The *cut-off* mechanism compares this value with a fixed cut-off value, which is a parameter to the runtime system, to decide whether a spark should be created at all.
- The *priority sparking* mechanism uses the value as a priority when deciding which spark to turn into a thread.
- The *priority scheduling* mechanism retains the priorities for the threads that are produced and uses them when deciding which thread to run.

These three mechanisms progressively allow more precise control of granularity, but also impose increasing overheads. Comparing a priority with a given threshold at spark creation time is very cheap. However, eliminating all low priority sparks regardless of the processor load may cause starvation. The priority-based mechanisms avoid this problem of starvation. However, it is generally more expensive to maintain priority queues for sparks and threads than to perform the simple threshold comparison that is needed for the cut-off mechanism. One objective of this paper is to assess whether this overhead is worthwhile.

3 Divide-and-Conquer Parallelism

In this section, we discuss the results we obtained from three simple generic divide-and-conquer algorithms. We then consider an algorithm which generates an unbalanced computation tree. Finally, we study a larger program: a parallel determinant computation.

3.1 A Generic Divide-and-Conquer Algorithm

There are three primary components to a divide-and-conquer algorithm:

- how the computation is sub-divided, *split*;
- the function that performs the actual computation, *solve*;
- and how the results are combined, *join*.

A generic divide-and-conquer skeleton *divCon* can be constructed from these three components plus a predicate to determine whether a computation can be sub-divided, *divisible*.

```
divCon divisible split join solve = f
   where f x | divisible x  = join (map f (split x))
             | otherwise    = solve x
```

To create parallel divide-and-conquer programs using the *divCon* template, we simply replace the sequential *map* by a parallel version *parmap*.

```
divCon divisible split join solve g = f
   where f x | divisible x  = join (parmap g f (split x))
             | otherwise    = solve x

parmap g f [] = []
parmap g f (x:xs) = parGlobal gx gx fx (fx:pfxs)
       where fx = f x
             gx = g x
             pfxs = parmap f xs
```

The extra parameter to *parmap* is a function, g, that is used to generate granularity information for each element of the list. To avoid significantly affecting the time performance, this should obviously be a much cheaper function than the worker function f.

In the following sections we study three applications of the generic divide-and-conquer algorithm that differ only in the relative computational costs of the three main steps. All three variants create a *balanced* computation tree in which the total work associated with each node decreases as the tree becomes deeper. This is a common pattern for divide-and-conquer algorithms. The parallel determinant computation described in Section 3.3 is an example of a real program that exhibits this behaviour.

3.1.1 Expensive Split

The first function, `xsplit` has an expensive *split* function (involving a factorial computation) and cheap *join* (maximum) and *solve* (identity) functions.

In this variant, small threads dominate the computation: about 72% of all threads have a runtime of less than 1000 *abstract machine cycles*[1]. Almost all of these threads are created in the last three levels of the divide-and-conquer tree, where hardly any work is left to be done. Approximately the same number of sparks is created in each of these levels: this is a result of the evaluate-and-die model, which causes many tiny sparks to be subsumed by their parent thread.

3.1.2 Expensive Solve

The second function `xsolve` has an expensive *solve* function (sum of factorials), but cheap *split* (enum-from-to) and *join* (maximum) functions.

This program has the coarsest granularity of the three generic algorithms. The average runtime of all threads is 5330 cycles compared to 2387 cycles for `xsplit` and 2304 cycles for `xjoin`. Although there are still many more small threads than medium or large threads (68% of all threads have a runtime less than 1000 cycles), they are less significant than in the other two variants. In total, more relatively large threads are created because more computation is done at the leaves of the tree: 26% of the threads have a runtime greater than 10000 cycles (compared to about 6.5% for `xsplit` and 7% for `xjoin`). This is the main reason why this variant shows the highest average parallelism of the three generic algorithms.

3.1.3 Expensive Join

Finally, `xjoin` has cheap *split* (enum-from-to) and *solve* (identity) functions, but an expensive *join* (expensive sum) function.

The `xjoin` variant has the highest percentage of tiny threads: 84% of all threads have a runtime smaller than 1000 cycles. This is due to the fast creation of the tree structure caused by the cheap split phase. This results in the early creation of many fine-grained leaves which are not subsumed by their parents. This high degree of parallelism creates many runnable or blocked threads (a maximum of 340 for `xjoin` compared to 317 for `xsolve` and 298 for `xsplit`). These threads exist for a rather long time which explains the small number of total threads: 4176 compared to 4868 for `xsplit` and 4854 to `xsolve`.

3.1.4 Granularity Control

In order to reduce the number of small threads in these programs we have used a cut-off mechanism where the depth of the recursion represents the size of the computation. Figure 1 shows how the speedup for the `xsplit` and the `xsolve` variants varies when the cut-off value is changed. Each graph shows results for

[1] This measure is defined precisely elsewhere [4] — we will use it as our basic cost measure throughout this paper.

two different communication latencies. It is obvious that the cut-off is more effective for `xsplit` than for `xsolve`. This is because the former produces more small threads than the latter.

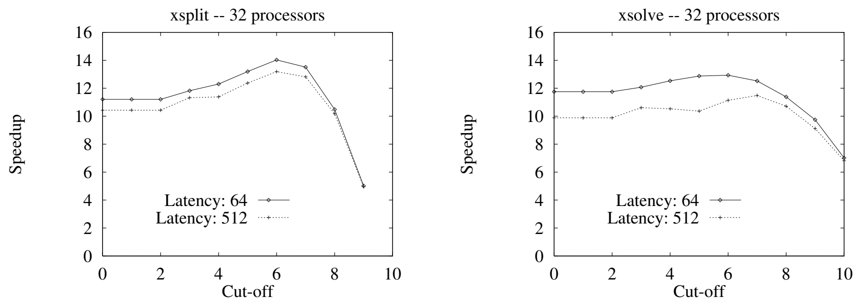

Figure 1: Speedup of `xsplit` and `xsolve` with varying cut-off values

The root cause of the rather small improvement in speedup is the fact that sparks are created in a breadth-first fashion. This means that the coarser-grained threads near the root of a balanced tree are created early on in the computation. Since these are the threads that will be picked up first, the smaller threads at the leaves are rarely executed anyway, and will be pruned automatically by the evaluate-and-die strategy.

3.2 An Unbalanced Divide-and-Conquer Algorithm

In contrast to the programs in the previous section the `unbal` function below produces an unbalanced computation tree as shown in Figure 2.

```
unbal 0 = 1
unbal n
 | one_of_many n = n                              -- leaf case
 | one_of_few  n = maximum list                   -- node case
   where list = parmap costfn unbal [0..n-1]
         costfn i = if one_of_many i then 0 else i
         one_of_few x = x `rem` diverge_every == 0
         one_of_many = not . one_of_few

diverge_every = 5
```

For this function all of the split, join, and solve phases are cheap. Since only every fifth node in the tree performs a recursive call, there are many leaves on all levels of the tree.

Figure 3 shows how the speedup of this program changes as the cut-off values are varied. The improvement is much greater than for the balanced algorithms because only every fifth or so spark is large. The default spark selection strategy

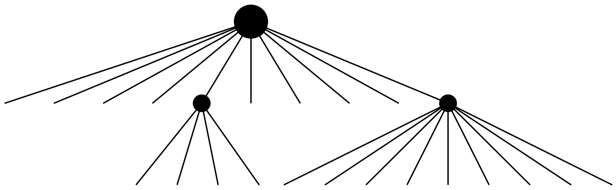

Figure 2: Unbalanced divide-and-conquer tree generated by `unbal`

is therefore likely to choose earlier, but inconsequential sparks for execution as threads.

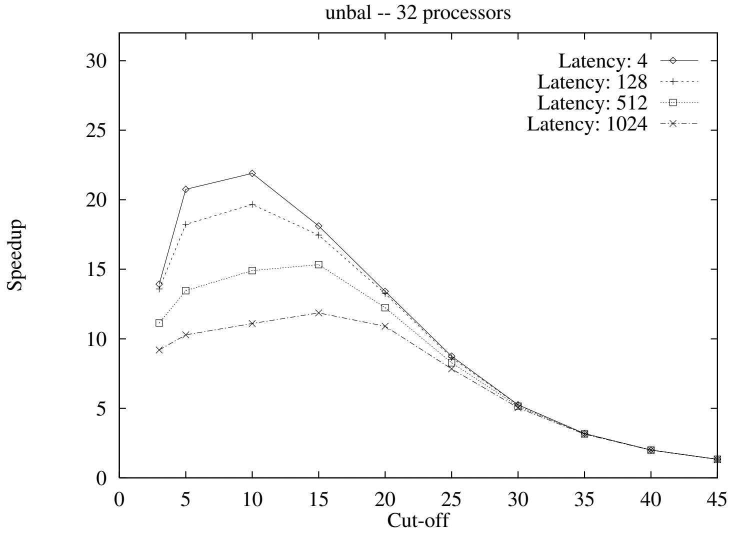

Figure 3: Speedup of `unbal` with varying cut-off values

Figure 4 compares the granularity graph for `unbal` using the optimal cut-off against that when no cut-off is used (the optimal cut-off eliminates only the leaf computations). Threads of similar lengths are grouped together. The height of the dominant bar in the second graph is just about one tenth of that in the first graph (note that we have used a logarithmic scale). This comparison shows that most of the small threads have been eliminated by the cut-off. Note that since the granularity function only approximates the actual granularity not all small threads have been discarded. As long as the cut-off accurately discards most of the small threads yet preserves all of the large threads this will not significantly affect overall performance

A comparison between the two priority mechanisms we have implemented is shown in Figure 5 for `xsolve` and `unbal`. This measures speedup against

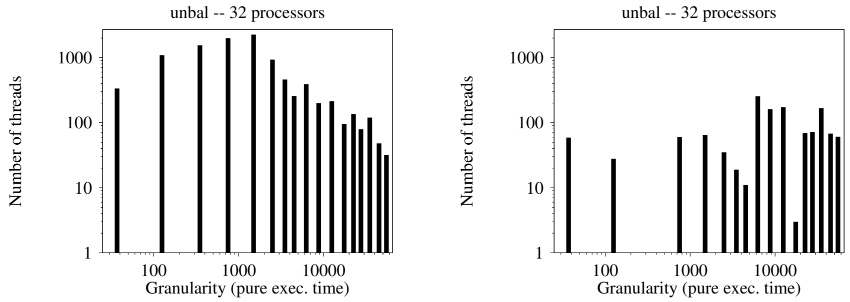

Figure 4: Granularity of `unbal` without and with optimal cut-off

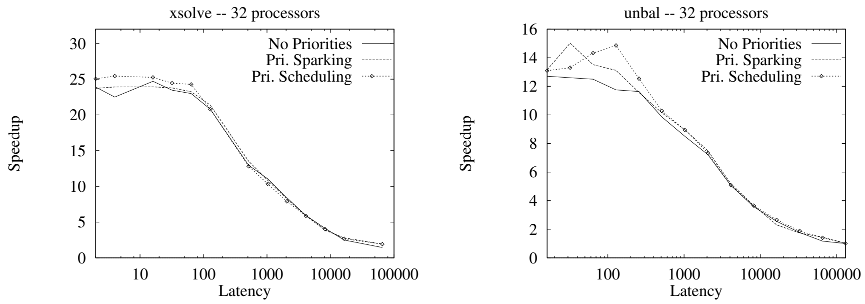

Figure 5: Speedups for `xsolve` and `unbal` with priority sparking/scheduling

communications latency for a 32-processor machine. The main reason for the poor improvement in speedup is that both the spark and thread queues tend to be quite short for these programs. Obviously, a priority scheme will have only minimal effect if there are only a few items to choose from. For `xsolve`, when the latency is less than 64 cycles, the average spark queue length is between 2.9 and 5.4 and the average thread queue length is between 2.3 and 2.7. With higher latencies, both averages quickly approach 1. This behaviour is reflected in the speedup graph of Figure 5 where the priority schemes cease to yield any significant improvement as soon as the latency exceeds approximately 128 cycles.

In contrast, for the `unbal` program, there are on average more than 28 sparks in the spark queue for latencies up to 256 cycles, with a proportional decrease to 4 sparks at a latency of 2048 cycles. The average thread queue length is greater than 2.2 up to a latency of 2048 cycles and then quickly approaches 1.

It is interesting to observe that speedup can be better for low latencies than for zero latency. The explanation for this apparently counter-intuitive result is that at very low latencies sparks are stolen and turned into threads almost instantaneously. If the value of the sparked closure is soon needed by the parent, then the parent thread must now block and search for new work, where it would normally have simply evaluated the closure itself. Thus more overhead can be incurred because more threads are created. As the latency increases, so does the probability that "junk" threads are absorbed by their parents, with a consequent increase in granularity and speedup.

It is interesting to ask whether the priority mechanisms could be more effective if the overhead was reduced. When we completely eliminated the overhead costs in the simulator, we measured execution time improvements of between 5% and 10% over all latencies for the generic divide-and-conquer programs. However, even in this setup the improvements compared to a cut-off mechanism are usually very small. Only for the unbalanced programs the priority mechanisms out-performed the simple cut-off mechanism for all latencies.

3.3 Parallel Determinant Computation

The parallel determinant computation is the central part of a parallel linear system solver which we have described elsewhere [7]. In order to compute the determinant of an $n \times n$ matrix it is first split into n sub-matrices. This is done by choosing one row as the 'pivot' row. For each element of this row a sub-matrix is constructed by cancelling from the original matrix the column and row to which the pivot element belongs. The determinant is the weighted sum of the determinants of these sub-matrices. The weights are the pivot elements.

The following table shows the average runtimes of threads generated by various spark sites for two different input matrices: a dense matrix of size 6×6 (left column) and a sparse matrix (i.e. a matrix with many 0 entries) of size 7×7 (right column).

Spark site	Number of Threads		Average runtime	
	dense	sparse	dense	sparse
zero entries	–	1160	–	188.6
sign	68	1077	107.8	232.0
cancel row/column	224	429	396.9	412.2
construct sub-matrix	316	287	708.4	819.6
sub det	538	91	12703.3	70734.1
Total	1147	3045	6342.4	2407.8

Since each 0 in the pivot row generates a leaf thread in the computation tree, a sparse matrix generates a rather unbalanced computation tree, with many small threads ("‘zero entries" in the table). In contrast, a dense matrix will generate a well-balanced computation tree.

For dense matrices, the most fine-grained threads are those that compute the sign of the pivot element. The next smallest spark sites are those that are

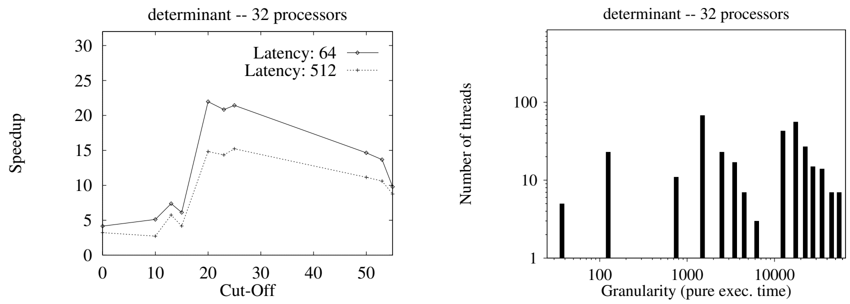

Figure 6: Speedup and granularity of `determinant` with varying cut-off values

involved in splitting the computation: cancelling elements from the pivot row and column, and actually constructing the sub-matrices.

The most interesting spark sites, however, are those that actually compute the determinants of the sub-matrices ("sub-det" in the table). These generate significantly more coarse-grained threads: all threads have an execution time of at least 1200 cycles.

Figure 6 shows the effect of using cut-offs on the parallel determinant computation with a sparse input matrix. The best results are obtained when two groups of small threads are eliminated: those that compute the sign and those that cancel the pivot element. Most of the remaining threads are needed to compute the summands. A few small threads are still needed to avoid starvation at the end of the computation: this is the main reason for low speedups when the cut-off is set too high. The granularity graph in Figure 6 shows that with an optimal cut-off almost all of the small threads have been successfully eliminated.

With a dense matrix, the behaviour of the parallel determinant program is much closer to that of the generic divide-and-conquer algorithms. Even without the cut-off mechanism, good speedup is achieved since there are far fewer leaf nodes and many large threads are created early on in the computation. Speedup hardly varies when the cut-off value is changed, ranging between 19.5 and 19.9.

4 Related Work

Many authors have suggested techniques aimed at controlling the granularity of specifically divide-and-conquer programs, though few of these techniques also apply to other models of parallelism, and fewer still present as detailed results as those we have given here. This section considers some of the results that are most related to those we have presented here.

Roe used a highly idealised simulator to study the effect of the evaluate-and-die model for divide-and-conquer programs [11]. His conclusion then, which we have broadly confirmed, was that evaluate-and-die is not sufficient by itself to achieve high performance, and that additional granularity control mechanisms are needed.

Roe's best results were obtained with a manual cut-off mechanism similar to that we have described here. There have been several other attempts to use cut-offs, usually as manual programming techniques. Typical performance improvements are on the order of 10%–20%. For example, Huelsbergen, Larus and Aiken [6] report improvements of this order for one program on a shared-memory implementation of SML. A more interesting system is π-RED$^+$ [2] which implicitly bases the cut-off on the recursion depth. This achieves 10%–20% improvement on several programs.

Rushall [12] has recently developed a variant of lazy task creation [9] which reduces overhead when a program is running sequentially. Rather than sparking closures, the execution stack is searched from the top for potential threads when and only when a thread is actually needed. This will clearly be a successful strategy for balanced divide-and-conquer programs but, as we have shown, is unlikely to give good results for arbitrary computation structures or even unbalanced divide-and-conquer programs.

Another dynamic technique suggested by Aharoni, Feitelson and Barak [1] involves spawning threads only when the work available for the thread to perform is no less than the cost of spawning that thread. For divide-and-conquer algorithms this will normally prune leaf threads, and low cost sub-trees. In general, there is a danger with this approach of losing parallelism, or perhaps slowing takeup but it seems to behave quite well for rather unbalanced computation trees.

Rather than using either programmer control or analysis to improve granularity, Hofman [5] concentrates on using scheduling strategies in a fork-and-join parallel setting. His techniques aim at optimising joins by preventing thread migration at the end of the computation. This problem is much less severe for the evaluate-and-die mechanism because of the much lower overhead for obtaining results from child threads.

5 Conclusions

In this paper we have studied three different mechanisms for controlling the granularity of divide-and-conquer programs: a cut-off, a priority sparking and a priority scheduling mechanism. For the divide-and-conquer programs studied here, a simple cut-off mechanism often yields better results than the more complex mechanisms that have higher overheads. Closer examination shows that the average thread length depends on how balanced the computation tree is. When the tree is seriously unbalanced, granularity control mechanisms can achieve larger improvements in overall runtime. When the tree is balanced, however, the default ordering of sparks is already very good and so it is only possible to achieve relatively small improvements. As further work we plan to examine a combination of the granularity control mechanisms considered here and to extend the measurements to a broader class of algorithms.

Our results also apply, albeit in a different way, to the lazy task creation approach. Lazy task creation tries to improve granularity by provisionally inlining all potentially parallel threads. This minimises the overhead for parallelism at the expense of an increased overhead for thread creation. Thus, granularity information is still important when creating a thread. Our results show that this information often can be discarded after thread creation. This suggests to 'tag' the inlined, potentially parallel threads with their relative costs. Considering the conflicting goals of minimising inlining overhead and maximising granularity this would add some small overhead in order to reduce the total thread creation costs.

Our results also confirm that it is not reasonable to ignore communication costs when studying parallel behaviour. Especially when using a very sophisticated evaluation mechanism a realistic cost model is essential for understanding the runtime behaviour of the parallel program and the granularity of the generated threads.

The ultimate objective of our research is the implementation of a practical static analysis to determine thread granularity. The analysis must produce information that can be used effectively to control the runtime behaviour of the parallel program. We have demonstrated that a cut-off mechanism based on relative thread sizes gives good results for the examples we have studied. This strengthens our belief that a very simple analysis should be sufficient to provide information that can be effectively exploited by our parallel runtime system.

Acknowledgements

Jim Mattson and Phil Trinder have worked hard on the GUM system, on which our work is partly based. Will Partain does sterling work in keeping the Glasgow Haskell compiler running in spite of our repeated attempts to break it. Simon Peyton Jones has provided several important insights into, and valuable criticisms of the work that is reported here.

References

[1] G. Aharoni, D.G. Feitelson, and A. Barak. A Run-Time Algorithm for Managing the Granularity of Parallel Functional Programs. *Journal of Functional Programming*, **2(4)**:387–405, 1992.

[2] T. Bülck, A. Held, W. Kluge *et al.* Experience with the Implementation of a Concurrent Graph Reduction System on an nCube/2 Platform. *Proc. CONPAR '94*, Linz, Austria, LNCS 854, pp. 497–508, 1994.

[3] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Dept. of Comp. Sci., Yale Univ., Apr. 1988.

[4] K. Hammond, H-W. Loidl, and A.S. Partridge. Visualising Granularity in Parallel Programs: a Graphical Winnowing System for Haskell. *Proc. HPFC '95*, pp. 208–221, Denver, Co, Apr. 1995.

[5] R. Hofman. *Scheduling and Grain Size Control.* PhD thesis, Univ. of Amsterdam, 1994.

[6] L. Huelsbergen, J.R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. *Proc. ACM L&FP '94*, pp. 79–90, Orlando, Fl, June 1994.

[7] H-W. Loidl, K. Hammond, and A.S. Partridge. Solving Systems of Linear Equations Functionally: a Case Study in Parallelisation. Tech. Rep., Dept. of Comp. Sci., Univ. of Glasgow, 1995.

[8] D.L. McBurney and M.R. Sleep. Transputer based experiments with the ZAPP architecture. *Proc. PARLE '87*, LNCS 258, pp. 242–259, Eindhoven, The Netherlands, June 1987.

[9] E. Mohr, D.A. Kranz, and R.H. Halstead Jr. Lazy Task Creation: a Technique for Increasing the Granularity of Parallel Programs. *Proc. ACM L&FP '90*, pp. 185–197, Nice, France, June 1990.

[10] S.L. Peyton Jones, C. Clack, and J. Salkild. High-Performance Parallel Graph Reduction. *Proc. PARLE '89*, LNCS 365, pp. 193–206, 1989.

[11] P. Roe. *Parallel Programming using Functional Languages.* PhD thesis, Dept. of Comp. Sci., Univ. of Glasgow, Feb. 1991.

[12] D. Rushall. *Task Exposure in the Parallel Implementation of Functional Programming Languages.* PhD thesis, Dept. of Comp. Sci., Univ. of Manchester, 1995.