Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell

Kevin Hammond*, Hans Wolfgang Loidl†

Department of Computing Science
University of Glasgow
Glasgow, G12 8QQ, U.K.
{kh,hwloidl}@dcs.gla.ac.uk

Andrew Partridge

Department of Computer Science
University of Tasmania
Australia
A.S.Partridge@cs.utas.edu.au

Abstract

To take advantage of distributed-memory parallel machines it is essential to have good control of task granularity. This paper describes a fairly accurate parallel simulator for Haskell, based on the Glasgow compiler, and complementary tools for visualising task granularities. Together these tools allow us to study the effects of various annotations on task granularity on a variety of simulated parallel architectures. They also provide a more precise tool for the study of parallel execution than has previously been available for Haskell programs.

These tools have already confirmed that thread migration is essential in parallel systems, demonstrated a close correlation between thread execution times and total heap allocations, and shown that fetching data synchronously normally gives better overall performance than asynchronous fetching, if data is fetched on demand.

1 Introduction

Our aim is to produce fast, cost-effective implementations of lazy functional languages. One way to improve speed is to introduce parallelism, and pure functional languages offer long-term hope for good speedups on many processors without much programmer effort.

We have already successfully shown that good absolute speedups can be achieved on a machine which has a limited number of processors and a relatively small interprocessor communications latency, the novel GRIP multiprocessor [8]. This is comparatively straightforward, because such machines can take advantage of quite fine grains of parallelism. However, as the number of processors in the machine is increased, architectural constraints tend to force an increase in interprocessor communication latency. This means that to achieve acceptable speedups, it is necessary to break the program into appropriately-sized grains for parallel execution.

In his thesis, Goldberg explored the idea of "optimal" grains of parallelism through the use of serial combinators [5]. Unfortunately, this level of granularity proved far from optimal for the Alfalfa implementation [6] on the distributed-memory Intel iPSC, despite showing promising performance for the Buckwheat implementation on the shared-memory Encore Multimax. Although other authors have returned to this issue (e.g. [14]) there has been very little systematic research on the granularity displayed by real programs and the mechanisms which are necessary to control that granularity in a general framework. Partly this is due to the lack of realistic monitoring tools which can be used to obtain information about a program's granularity.

In this paper we describe a tool for investigating task granularity, and propose annotations which will enable us to exploit granularity information in a parallel environment. As a first step, we are simply investigating the mechanisms which should be used to control granularity. Since we do not know what information a granularity analysis might need to provide, we have currently annotated the programs by hand. However, it is our intention that the annotations will eventually be inserted by a *static granularity analysis*, so as to provide automated control of parallel programs.

*Supported by a SOED Research Fellowship from the Royal Society of Edinburgh and the UK EPSRC Parade project.

†Supported by a Kurt-Gödel scholarship (GZ 558.012/38-IV/5a/93) from the Austrian Ministry of Science and Research and by a KIP scholarship from the government of Upper Austria.

The structure of this paper is as follows. Section 2 describes the design of a fairly realistic parallel machine simulator, based on the Glasgow Haskell compiler. Section 3 presents a set of annotations we intend to use to improve parallel performance through optimising grain size. Section 4 describes the tools we have built for visualising thread granularities and related information. Section 5 gives some simulation results obtained using the simulator and visualisation tools. Finally, Section 6 concludes.

2 Granularity simulation

Many parallel simulators have been constructed. For the most part, these either deal with highly idealised parallel machines, or are accurate simulations of real or proposed parallel machines. The simulator described here lies somewhere between these extremes, in aiming to model a range of machines and architectures to a reasonable degree of accuracy.

In a recent paper Runciman and Wakeling describe an idealised simulator for parallel machines [15]. This simulator is similar to previous work (e.g. [2], [14]) in adding pseudo-parallel traces to an otherwise sequential run, but has the advantage of producing a graphical profile of the simulated processor activity in terms of blocked, runnable and running tasks. While such a tool is potentially highly useful in identifying programs which are insufficiently parallel (the FLARE project produced several such examples [16]), it has a number of deficiencies which make it an unreliable predictor of real parallel performance:

- each supercombinator reduction is considered to take an identical amount of time;
- communication costs are not accounted for;
- task creation costs are not considered; and
- the scheduling algorithm is highly unrealistic.

Typically these inaccuracies will lead to an overstatement of the degree of parallelism in a program, but we cannot honestly state that the figure derived is a strict upper bound on parallelism.

A more detailed simulator than the one built by Runciman and Wakeling, tunable to suit different classes of architecture, but based on the same ideas, would seem to be ideal for monitoring granularity effects.

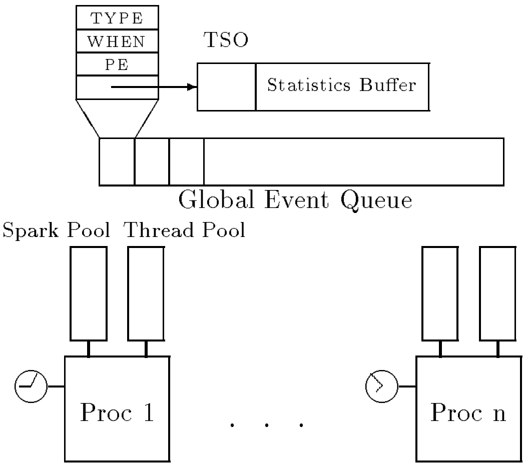


Figure 1: Overall structure of the simulator

2.1 Threads

Our simulator, GranSim, is built around the threaded runtime system for the Glasgow Haskell compiler [12], which Mattson has developed for work on concurrency in a sequential setting [4]. The compiler is a state-of-the-art optimising Haskell compiler.

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 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 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 Running threads

At each context switch, a CONTINUE event for the current thread is added to the event queue. We then create new threads from the spark pool of each idle processor as START events at CREATE-TIME steps in the future. The next event is then scheduled, and the global timer is advanced if necessary.

When a thread terminates, the processor chooses a runnable thread, if it has one, from its local thread pool, and posts a CONTINUE event. If there are no runnable threads but its spark pool is not empty, it starts a new thread, posting a START event. Otherwise it is idle, so it posts a FINDWORK event.

A FINDWORK event simulates the action of an idle processor looking for work. If any processor has excess work, then this is stolen by the idle processor by generating a MOVESPARK or MOVETHREAD event at STEAL-TIME in the future. Sparks are stolen in preference to threads, and have a lower steal time reflecting their smaller size. Although many implementations ignore thread migration on the grounds that it is costly and difficult to implement, experiments on GRIP have shown that this is essential for good parallel performance of coarse-grained applications [9].

2.4 Fetching Nodes

During execution, a processor may need the value of a node which it does not have available locally. In this case, it issues a FETCHNODE event at FETCH-TIME steps in the future to the processor which owns the node. On receipt of a FETCHNODE event, the owner issues a FETCHREPLY event at FETCHREPLY-TIME steps in the future to the original processor. Evaluated nodes (*normal forms*) are copied to the processor that requested the value; unevaluated nodes (*thunks*) are moved to the requesting processor, which then becomes the owner of that node; nodes under evaluation (*black holes*) become blocking queues as described below. Thus, it is possible for a FETCHNODE event to arrive at a processor which no longer owns the node because it was a thunk which was moved. In this case the FETCHNODE event is forwarded to the new owner. On receipt of a FETCHREPLY event, the thread which needed the value of the node may continue execution.

Packing graph

We currently model the GRAPH for GRIP style of incremental data-fetching [8], where nodes are copied or moved to a processor only when they are explicitly demanded. This is potentially disastrous in a distributed-memory setting since it significantly increases the number of packets which must be sent when copying a large data structure between processors. In future, we plan to model the more sophisticated data-fetching models proposed for GRAPH for UMM [17], which are similar to those used by the Concurrent Clean implementation [10]. These models improve performance in a distributed-memory setting by packing related graph into a single packet for transmission. In time we hope to answer the question of exactly how much data of what type should be packed into a single packet in order to ensure the best possible performance at a given latency.

Fetching data asynchronously

The most basic strategy to adopt when fetching data from a remote processor is to simply wait until the data arrives, a *synchronous* fetch. This is potentially inefficient, since the processor is idle until the data becomes available. An alternative is to temporarily suspend the current thread in favour of some other work, thus performing an *asynchronous* fetch. Such a context-switch is likely to be expensive on current hardware, so asynchronous fetching will normally only be beneficial if significant parallelism can be gained as a result, for example when communications latency is high or because work can be scheduled much earlier.

Our simulator implements both synchronous and asynchronous fetching. When searching for work following an asynchronous fetch, several possible sources are tried, in order:

1. a locally runnable thread;
2. a spark in the local queue;
3. a spark from another processor;
4. a runnable thread from another processor.

In some circumstances it may be desirable to try only the first few of these possibilities. This issue is considered in more detail in Section 5.4.

2.5 Blocking and resumption

If a thread demands the value of a node which another thread is evaluating, and which therefore has been turned into a "black hole", then it *blocks* and a blocking queue is created (Figure 2). When a node that has a blocking queue is updated, a RESUME event is scheduled at RESUME-TIME in the future for each <PROC,TSO> which is in the blocking queue. When a RESUME event occurs, the corresponding TSO is added to the queue of executable threads on PROC.

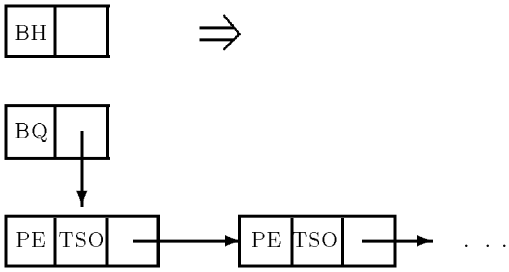


Figure 2: Blocking queue

2.6 Measuring costs

Execution Time

Each basic block is analysed to produce a count of the number of operations in each of five categories: arithmetic/register operations, loads, stores, branches, and floating-point operations. These counts are passed to the simulator each time the basic block is executed. It is much easier to produce reasonable values for each category of instruction than to determine the exact runtime of a basic block (in fact, discovering the exact runtime of a program without actually running it is something of a black art for many modern architectures). The use of categories also allows the operations to be weighted appropriately for different architectures such as the MC68020 used in GRIP or the SPARC used in the IBM SP/2, ICL Goldrush etc.

Heap allocations

Each time a heap allocation occurs, the number of words allocated is added to the appropriate part of the statistics buffer. This provides an accurate count of the number of words allocated by each thread, which can be directly compared with results previously obtained on GRIP [7].

Shared data

Each object has a fixed-size bitmask attached, which shows which processors have a copy of that object. Newly created objects reside on the current processor. When a node is entered to retrieve its value, a check is made to see if there is a local copy. If not, the current thread becomes temporarily suspended and the object is copied to the current processor by setting a bit in the bitmask. Figure 3 shows the layout of closures in the global heap using this scheme.

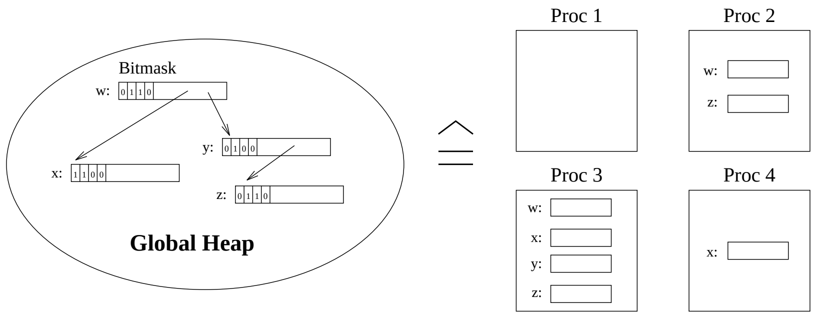


Figure 3: Space-efficient representation of globally shared closures

This technology has the advantage of avoiding duplicate copies in the simulator and eliminates the need to simulate the FETCHME closures etc. which are used in GRAPH for UMM [17]. It is, however, limited to simulating at most a fixed number of processors because of the use of a fixed-size bitmask. A 32- or 64-bit bitmask is easily provided, and should be capable of representing an interesting number of processors. If it becomes necessary to model large numbers of processors, the bitmask could easily be replaced by a linked-list of processor numbers.

Communications

Several parameters are available to tune the costs of the communications system.

- **Packet pack time**: the time needed to create a packet before it is sent.
- **Packet unpack time**: the time needed to read a packet on the receiving processor.
- **Packet tidy time**: the time needed to free buffers etc. on the sending processor after a packet is sent.
- **Communications latency**: the time between sending a packet on one processor and receiving it elsewhere.
- **Additional latency**: the latency for the second and subsequent packets of a message.

The latency parameters represent hardware costs, the packet construction parameters represent software costs. It is quite possible for hardware costs to be swamped by software costs and thus for the theoretical communications limits never to be reached in practice. For example, on the Intel iPSC used by Goldberg for the Alfalfa system, the communications latency, though high, was much less than the packet tidy time [5]. This had a serious effect on overall performance.

Often the latency for sending multiple packets as part of a message will be less than the latency of sending the first packet. This is because the hardware/software cost of packet routing is often borne only by the first packet of a series.

The latency parameters are fixed values which do not depend on the processors involved in the communication. This simplifies the simulation at the expense of losing information about locality and topology. At the moment, we do not regard this as a significant defect: there is some evidence that irregular parallel programs are generally insensitive to the topology of the communications system; and in many systems communications latency does not vary significantly across the network. This is something that might be interesting to study at a later date, however.

No attempt has been made to model the effect of limited bandwidth on communication costs, other than through increased latency. Our experience with bandwidth limitations is that, as with virtual memory, performance degrades rapidly to the point of unusability once physical limits are reached. Rather than attempting to locate bandwidth limits through detailed simulation, it seems more sensible to attempt to reduce bandwidth requirements through reduced communication.

2.7 Performance of the simulator

Because the simulator is compiled as part of the compiler's runtime system, and does not interpret functions etc. it is relatively fast. Obviously, simulating many processors is more costly than simulating one processor. The table below summarises the costs for the three programs which were studied here. Times are for a DEC Alpha 3000/600 running at 166MHz.

Program	*Normal Runtime*	*Processors*	*Simulator Runtime*	*Events per second*
ray	2.77s	1	38.48s	26000
		16	40.96s	24600
		64	44.76s	23500
word	0.05s	1	1.66s	26300
		16	1.90s	25700
		64	2.27s	23100
factorial	0.01s	1	1.79s	33500
		16	1.94s	31400
		64	2.13s	29100

The simulation is between 14 times as slow (*ray*, 1 processor) and 213 times as slow (*factorial*, 64 processors) as the compiled sequential version. Interestingly, the simulation costs are greatest for the simplest program, and decrease with the complexity of the program. This is because more simulated work is done between expensive context-switches in the more complex programs, as can be seen from the fact that the number of events per second which are processed is relatively constant. The cost of managing more thread queues and handling more events as more processors are simulated means that runtimes increase. Because the increase in the number of events handled is not usually large, the rate of events handled per second of simulation time decreases as more processors are simulated.

Clearly, the simulation is much slower than the compiled version, but overall performance is still quite acceptable. For comparison, on a 16-processor GRIP, *word* ran in 0.69s, and *ray* ran in 11.475s (best cases). With the march of time, progress on sequential architectures obsoletes parallel machine designs based on earlier processor designs (in this case 16MHz Motorola MC68020s).

Comparisons with other simulators rather unsurprisingly reveals that GranSim is midway in cost between an idealised simulator such as `hbcpp`, and a detailed instruction tracer such as Irlam's SPA. We do not have SPA or `hbcpp` available on a DEC Alpha, therefore these times are for a Sun SPARC 1. Both `hbcpp` and GranSim were simulating 32 processors.

Program	*Sequential*	*hbcpp*	*SPA*	*GranSim*
factorial	0.05s	2.4s	11.6s	10.3s
word	0.5s	4.8s	58.5s	19.5s
ray	14.1s	79.8s	1741.2s	262.1s

Computing cost functions does increase compilation time slightly, but the increase is so small as to be effectively unmeasurable.

3 Granularity annotations

The annotations (really combinators) used on GRIP are `par` and `seq`, which have the following behaviour:

- `par s e` sparks a thread to evaluate `s` to WHNF. The sparking thread continues to evaluate and returns `e`. In order to ensure correct functional semantics, `s` must terminate whenever `e` does.
- `seq x y` evaluates `x` to WHNF, then evaluates and returns `y` as part of the same thread.

For the most part these two annotations will be all that is necessary to use the results of granularity analysis. For fine control, a number of additional annotations will be necessary, such as those suggested for Concurrent Clean [3]. However, for the purposes of this paper we introduce only two new annotations:

- `parGlobal name s e` is identical to `par` except that it takes an extra integer name parameter, for use in identifying the spark location.
- `parLocal name s e` is identical to `parGlobal` except that the sparks it produces are local and cannot be exported to another processor.

4 Visualisation

We provide a set of graphical tools to help analyse the results of the simulation. Most of these tools give information about the granularity of threads. However, we also provide some tools for showing overall activity profiles.

The visualisation tool we have built especially for granularity analysis takes a trace file produced by a simulator run and produces graphical profiles of the number of threads by:

- granularity (pure execution time);
- amount of heap allocation (in words);
- number of local sparks created by the thread;
- number of global sparks created by the thread;

- percentage of the total run time spent communicating.

If preferred, each profile can be specialised for threads created from the following categories of spark:

- those marked `parLocal`;
- those marked `parGlobal`;
- those marked `parGlobal` and exported;
- those marked `parGlobal`, but not exported.

In addition, because each spark site can be named, each kind of profile can be produced individually for just those threads that are produced by a given spark site. This allows us to focus on the most interesting points of thread creation without being overwhelmed by irrelevant information. Execution time, heap, and communications profiles can optionally be produced as cumulative graphs. These have the advantage over the ordinary profiles that the profile is continuous, and therefore does not depend on arbitrarily-chosen groupings of similar sizes of threads.

These profiles let us do the following:

- identify spark sites that generate threads which communicate too much;
- check that the most suitable sparks are being exported;
- check that `parLocal` and `parGlobal` annotations are being used appropriately;
- identify excessively large threads which may be split into smaller ones for improved parallelism;
- identify which threads and/or spark sites cause space problems;
- distinguish the behaviour of threads generated from different spark sites.

In addition, we have also found it useful to display the overall runtime behaviour of a parallel program, using the following three kinds of activity profile:

1. activity of individual threads;
2. activity of individual processors;
3. activity of the machine as a whole.

The processor activity profile can optionally be overlaid with thread migration information.

5 Results

This section gives some results obtained using our simulation tools on three test programs: a simple functional ray-tracer *ray* described in [16], a word searching program *word*, also described in [16], and the following divide-and-conquer factorial program, *factorial*:

```
main inp =
  [AppendChan stdout (show (factorial 5000))]

factorial x = pf 1 x

pf x y
  | x < y     = parGlobal 1 f1
                           (seq f2 (f1+f2))
  | otherwise = x
                where m  = (x+y) `quot` 2
                      f1 = pf x m
                      f2 = pf (m+1) y
```

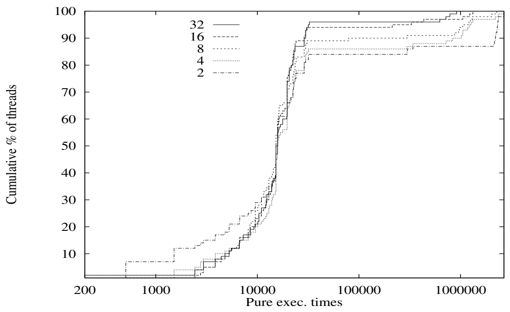
5.1 Accuracy of the Simulation

If we are to have any confidence in the results from our simulator, it is important to calibrate it carefully.

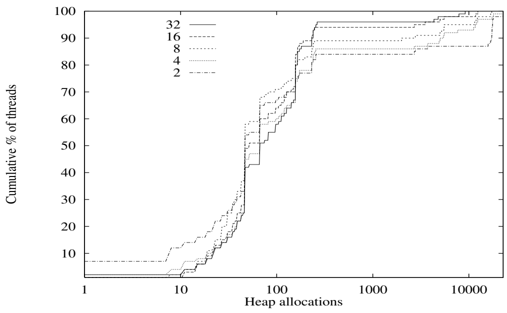
Based on studies of several real Haskell programs using Irlam's SPA instruction tracer for the SPARC, and including cache effects etc., we have assigned the following weights to each instruction category for non-superscalar SPARC processors (those which are typically found in workstations, or low- to medium-end parallel machines).

Category	*Weight*
Arithmetic	1
Floating-Point	1
Load, Store	4
Branch	2

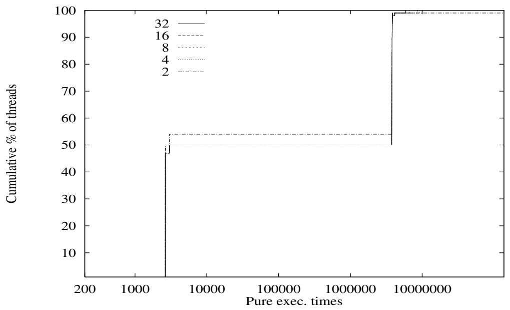
Comparing the unweighted instruction counts from the simulator with those obtained from actual instruction traces reveals that arithmetic instructions are usually within 10% of their predicted value, that loads and stores are usually within 2% of the prediction, but that branches may sometimes vary up to 20%, and floating point instructions up to 14%. Overall, comparing weighted instruction counts against actual clock cycles, we observed deviations between 2% and 15%. We have not yet attempted similar calibration with other architectures.



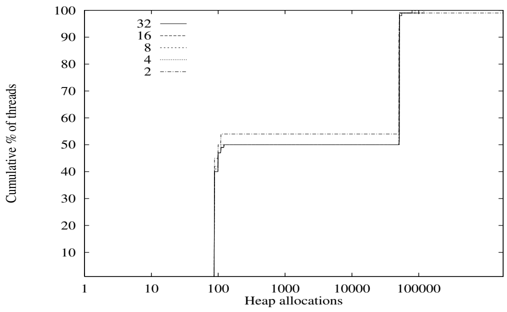
Word Search, Runtimes



Word Search, Heap Allocations



Ray Tracer, Runtimes



Ray Tracer, Heap Allocations

Figure 4: Cumulative time and heap profiles: Word Search and Ray Tracer

The main source of inaccuracy lies in failing to detect optimisations of multiplication and division operations. Since these are, in general, very expensive on SPARC processors, they are often optimised into more efficient operations by the C compiler that serves as the back-end to the Glasgow Haskell compiler. Unfortunately, the function used to assign costs to basic blocks cannot know that these optimisations have been performed.

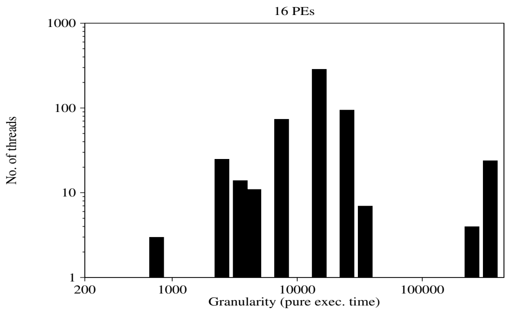
Other sources of inaccuracy are the need to estimate the average size of arbitrary-precision arithmetic values when costing arbitrary-precision operations, and the fact that it is impossible to automatically calculate the cost of a call to a non-Haskell function (e.g. through `ccall` [13]). In the latter case, it is usually possible for the programmer to cost the function explicitly, however.

5.2 Granularity

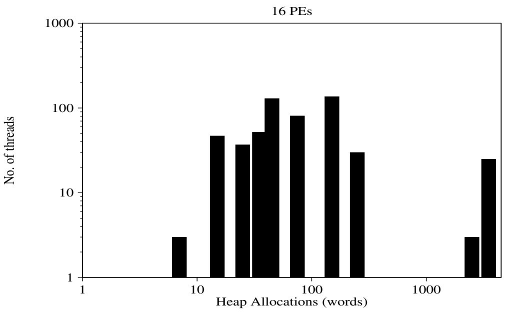
Figure 4 shows the cumulative time and heap profiles for *word* and *ray*, plotted on a logarithmic time scale. In both cases, the correlation between the heap and time profiles is clearly visible. As with the GRIP results reported in [7], there is a strong family resemblance between profiles for varying numbers of processors, with the 2-processor case being the most dissimilar. The cumulative profiles for *ray* also show quite remarkable visual correspondence between the results for different numbers of processors. This is confirmed by the high correlation between the heap and time profiles for each of the test programs.

	Correlation Coefficient		
Procs.	*word*	*factorial*	*ray*
2	0.998784	1.000000	0.999997
4	0.983595	0.999967	0.999934
8	0.978600	0.999992	0.999946
16	0.970150	0.999968	0.999961
32	0.968747	0.999914	0.999966

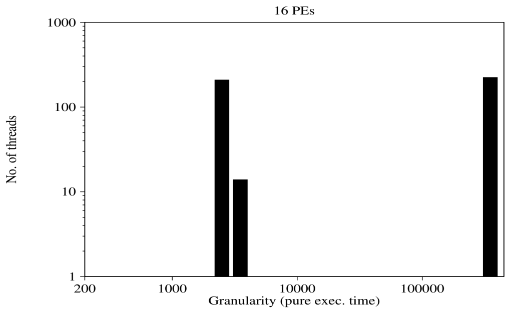
Figure 5 gives a more detailed picture of the time and heap profiles for *ray* and *word* with 16 processors (the profiles for other numbers of processors are similar, as



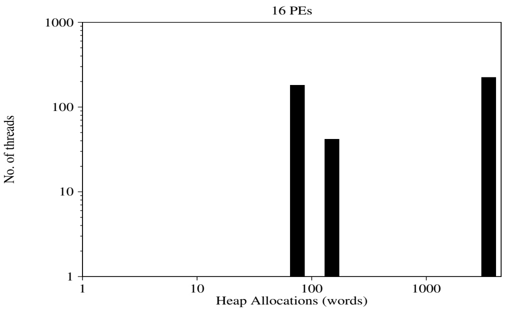
Word Search, Runtimes



Word Search, Heap Allocations

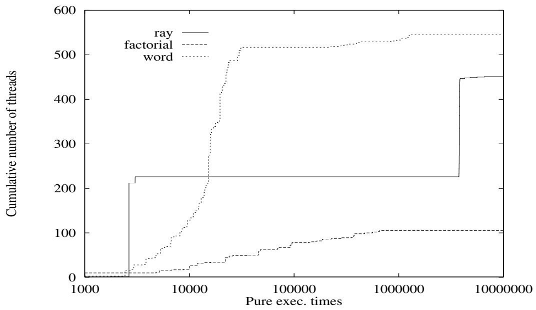


Ray Tracer, Runtimes

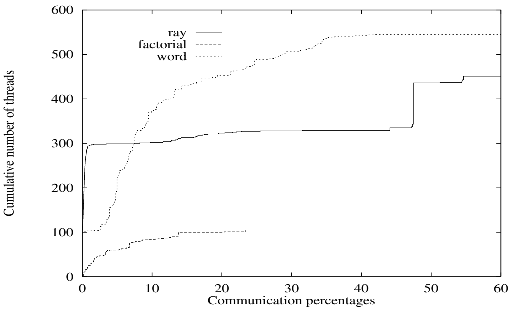


Ray Tracer, Heap Allocations

Figure 5: Runtime and Heap Profiles for Word Search and Ray Tracer



Runtimes

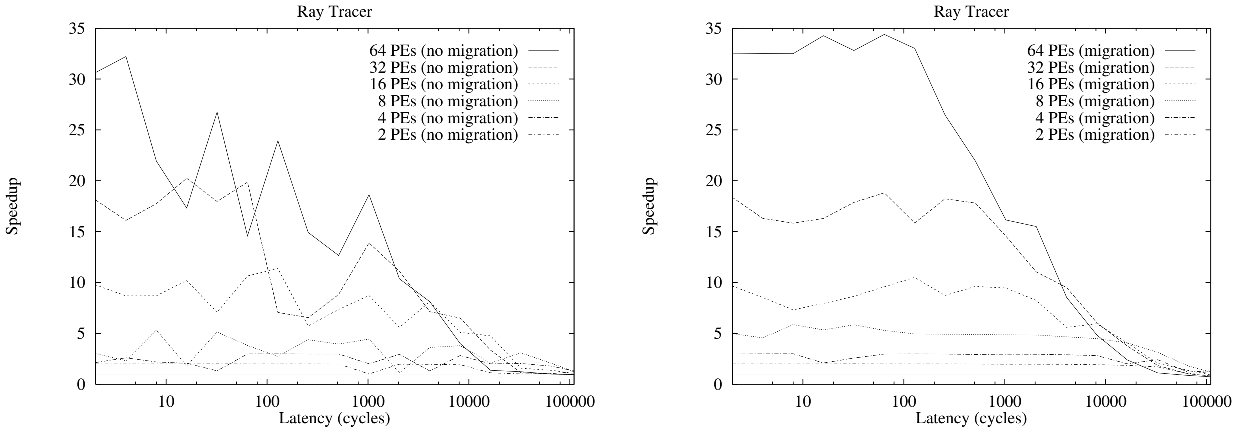


Communication percentages

Figure 6: Cumulative time and communication profiles, 16 processors

can be seen from the cumulative profiles). As would be expected from the cumulative profiles, there are clear similarities between the heap and the runtime profiles in each case.

For *word*, both profiles show a clustering of small and medium sized threads (with many more medium than small threads), then a gap before the two sets of large threads. Significantly, the number of threads in the largest category is approximately the same in both cases. The variance between the profiles is probably best explained by threads which perform the same number of heap allocations having slightly different



Without thread migration — With thread migration

Figure 7: Speedup vs. Latency for Ray Tracer

runtimes, and so being counted in different clusters in the time profile.

For *ray*, there are two principal clusters in both sets of profiles. This is due to the scheme used to parallelise the program: each thread in the cluster of small threads has the sole duty of walking down a list and sparking a thread to evaluate each element of that list in parallel. The cluster of large threads comprises those threads which are sparked by the small threads. The apparent improvement of removing the small threads would actually have a disastrous effect on performance, since this would result in almost no parallelism being created.

Figure 6 shows cumulative runtime and communication percentage profiles for each of the three test programs when run on a 16-processor machine with communications latency similar to that of GRIP. To enable direct comparison of the programs, an absolute number of threads is plotted. The two sets of profiles need to be read together to assess how good a candidate each program is for parallel execution on a machine with high interprocessor latency. The runtime graph essentially shows the raw parallelism that is present in each program, but gives no idea of the cost of exploiting it. The communication percentage graph shows whether threads spend a large or a small proportion of their time communicating, and gives some idea of how well the program will run on machines with higher interprocessor latencies.

The main point of interest from Figure 6 is the pair of curves for the *ray* program. This program has many large-sized threads, and an almost equal number of small-sized threads. There are two corresponding sharp steps in the runtime profile. However, the size of the steps in the runtime profile differs from that for the communications percentage profile, indicating that the two different sizes of threads do not have completely distinct communications behaviour. Although it is not possible to see it from these graphs, the threads with high communication percentages all have small runtimes, so the absolute amount of communication is reasonably small. We therefore expect *ray* to be reasonably tolerant to increased latency.

5.3 Latency and Thread Migration

Although not a primary objective, the simulator has also given us an opportunity to study the effects of latency, number of processors, and thread migration on execution times. Figure 7 shows the effect of varying these parameters on the speedup of the *ray* program. Speedups are measured as absolute speedup over the same program with no parallel overhead. To put these graphs in perspective, a latency of 5–10 cycles corresponds to a typical shared-memory machine, fast distributed-memory machines such as GRIP or the Meiko CS2 have latencies in the 100–500 cycle range, while typical distributed-memory machines such as the IBM SP2 have latencies of 1,000–5,000 cycles. Fully distributed machines connected over an ethernet LAN would typically have latencies of 25,000–100,000 cycles.

The performance when thread migration is enabled is much more consistent than when there is no thread

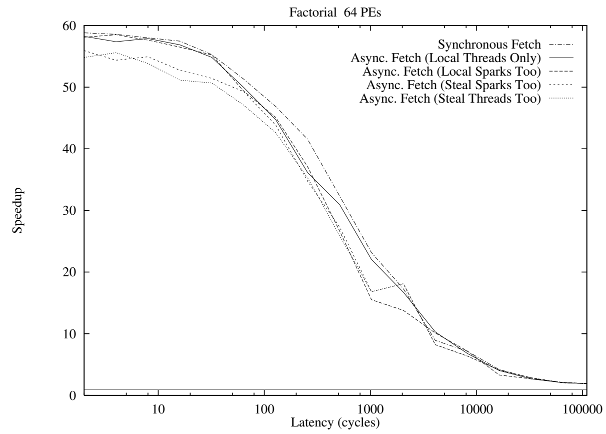


Figure 8: Fetching strategies: *factorial*

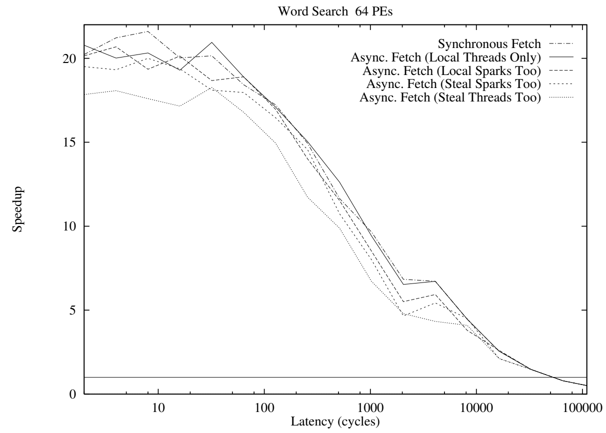


Figure 9: Fetching strategies: *word*

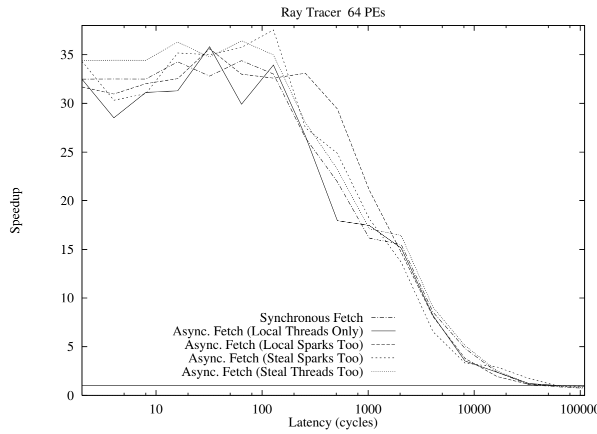


Figure 10: Fetching strategies: *ray*

migration, and generally better at low latencies. This poor speedup at low latencies is a consequence of the fact that when an early-started thread becomes temporarily blocked the processor is able to quickly acquire another thread to run. Without task migration, when the temporarily-blocked thread resumes, both it and the newly-acquired thread are locked to the same processor, leading to a loss of parallelism. At higher latencies it takes longer to acquire a thread, so processors are not able to do so while another thread is temporarily blocked, and the problem does not occur.

5.4 Fetching Strategies

Figures 8–10 compare the 5 different fetching strategies described in Section 2.4 for each of the three test programs. Interestingly, the most basic strategy of simply waiting until data is available actually yields the best performance in almost all cases. The next best strategy is to simply execute a runnable thread if one is available, but not to attempt to find work that might exist elsewhere in the system. The worst strategies are the ones which search most aggressively for work to do whilst waiting for data to be fetched.

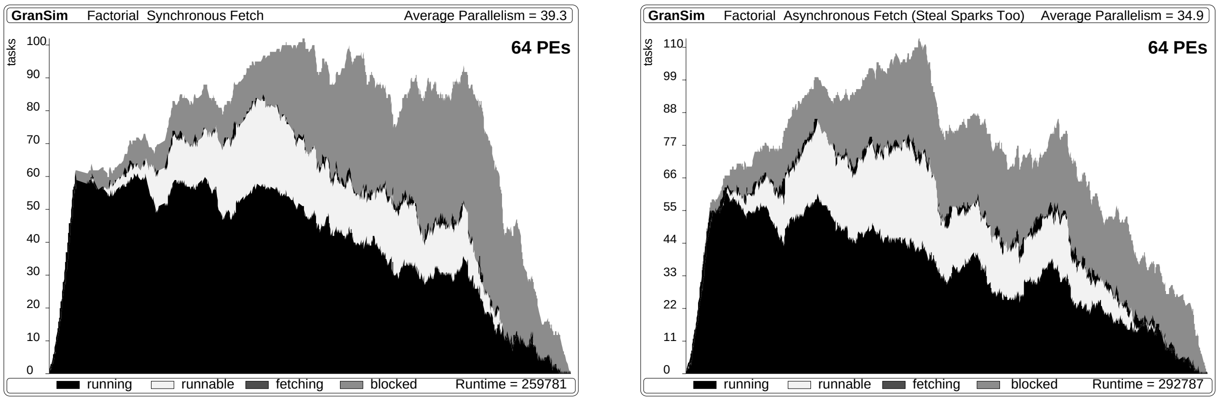
Normal behaviour

Figure 11 shows overall activity profiles for *factorial* on 64 processors with a latency of 256 cycles. This kind of profile shows the total number of running, runnable, fetching and blocked threads at any point in time.

With synchronous fetching, a processor becomes blocked while it fetches a remote node. With asynchronous fetching, however, another thread may be scheduled while the node is being fetched. In general, this leads to a significantly higher total number of threads, and this is borne out by these graphs. It is also clear that, for this program, relatively few threads are fetching data at any given time.

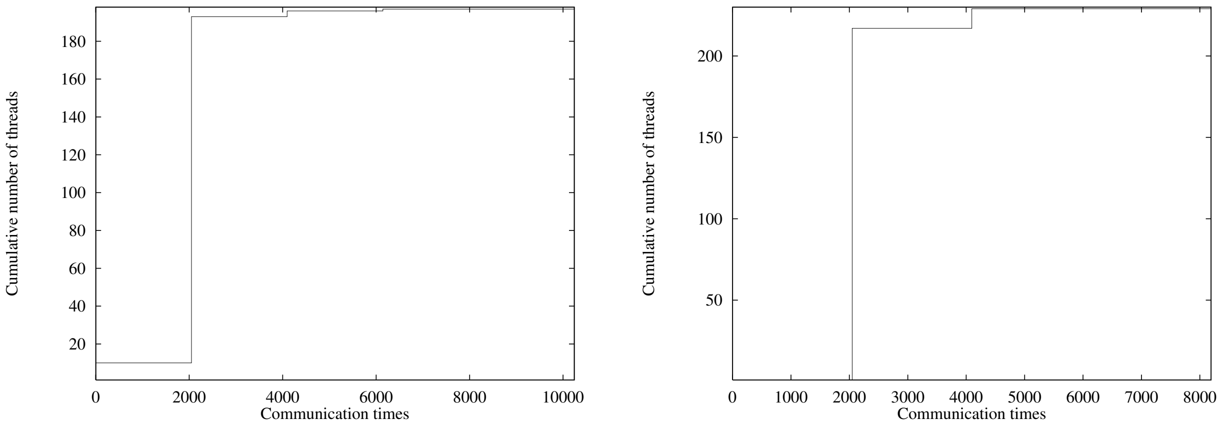
For asynchronous fetching, at any point in time, there are slightly more fetching threads than for synchronous fetching, and in fact this behaviour holds for all the test programs. This is because more sparks are stolen for remote execution. As a consequence, when these sparks are turned into threads, their first action will be to fetch the graph which they need in order to begin execution.

This behaviour is clearly revealed by the corresponding communication profiles (Figure 12). Using synchronous fetching, 10 threads never communicate,



Synchronous Fetch Asynchronous Fetch

Figure 11: Overall Activity Profiles for *factorial*



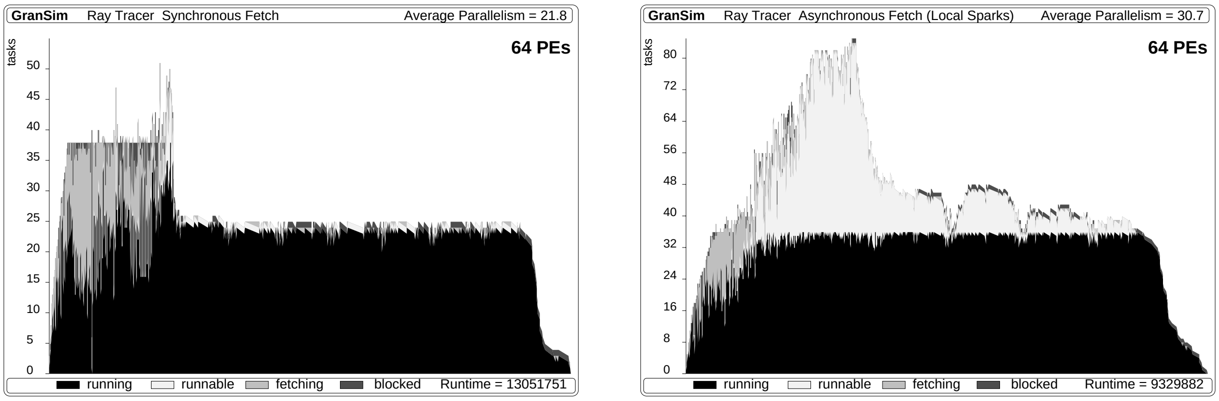
Synchronous Fetch Asynchronous Fetch

Figure 12: Cumulative Communication Costs for *factorial*

and only 3 threads do much communication. In contrast, when asynchronous fetching is used, all threads perform some communication, and 12 threads spend more than 4000 cycles communicating. Overall, with synchronous fetching each thread spends on average 1996 cycles communicating, but with asynchronous fetching this rises to 2155 cycles. This additional communication time is the main reason why asynchronous fetching takes longer than synchronous fetching.

Good asynchronous fetch behaviour

Figure 13 shows similar activity profiles for one interesting point which is revealed by the speedup graphs for *ray* (Figure 10). At this point (latency 400 on 64 processors), the best speedup is given by one of the asynchronous fetch strategies, where we use local threads and sparks. With synchronous fetching, many threads are busy fetching data early in the run. Asynchronous fetching allows new threads to be started, and so overlaps fetching with execution as intended. As a result of this internal parallelism more threads can be run early on, and so the overall execution time can be reduced. This is a clear success for asynchronous fetching.



Synchronous Fetch

Asynchronous Fetch

Figure 13: Overall Activity Profiles for *ray*

6 Conclusions

This paper has introduced an accurate and tunable simulator for a wide range of parallel machine architectures. Unlike most other simulators our objective is to discover information about the granularity of parallel threads. To assist this task, the simulator is provided with graphical visualisation tools which help isolate patterns of granularity, as well as providing detailed information about program behaviour. We have only shown some of the output that these tools can produce (those most relevant to the task at hand, and not those which are most visually attractive).

We intend to use this simulator to explore annotations for granularity control with a view to producing automatic granularity analysis. In order to produce a high-quality analysis we need high-quality performance metrics for each thread that the program produces. Simple overall execution-time, or average thread length metrics are unlikely to locate problems with the placement of annotations. We have already obtained some detailed granularity results for GRIP [7], but these earlier results are limited to allocation-based profiling, and do not allow us to explore latency-related issues. Since we plan to target our compiler for other parallel machines, it is important to obtain as broad a range of results as possible for particular annotations before investing significant effort in re-implementing GRAPH for UMM.

There is, of course, a danger that the design of the simulator may obscure real artefacts or introduce false ones. The only true performance measure is execution on a real machine. We have attempted to avoid these problems as far as possible by building our simulator on a state-of-the-art compiler, by modelling exactly the scheduling algorithm we propose to use for GRAPH for UMM, by careful modelling of significant events such as communication, and by comparing the results of the simulation with equivalent datapoints obtained from real hardware (sequential and parallel).

The results shown in Section 5 confirm earlier results obtained from GRIP. There are clear similarities between granularity profiles for varying numbers of processors, though the profile for the 2-processor case bears the least family resemblance. The similarity of the heap profiles from GranSim with those obtained from GRIP lends extra credence to the accuracy of our simulation.

We have also been able to confirm that thread migration is essential for good parallel performance of coarse-grained applications. This has already been shown experimentally by [9], and theoretically by [1]. This adds to our confidence in the correctness of the simulator.

An important new result to arise from the simulation is that there is a close correlation between time and heap granularity profiles for the programs studied.

Most significant for our work on granularity is the observation that for sensible latencies (i.e. those likely to be encountered in the real world), there is no parallel slowdown for the programs that we have studied. If confirmed, this result will help reduce the cost of static granularity analysis by eliminating the need for lower as well as upper-bound cost estimation (both are done by García et al. for Prolog [11]).

Finally, we have demonstrated that for incremental fetching, where data is fetched only on demand, it is normally better and rarely significantly worse to fetch data synchronously rather than asynchronously *even at high latencies*. We have also shown that it is best to use the least aggressive work location strategies if fetching data asynchronously. This result is at first somewhat counter-intuitive, but can be explained by the typically high cost of context-switching and by the increased communication that results from fetching remote data if more work is exported rather than performed on the processor that generated that work. Since this issue is clearly important, we plan to use GranSim to study this in more detail in the future, by simulating alternative strategies for fetching data, such as exporting some data with a spark, or fetching graph eagerly before it is known to be needed.

References

[1] FW. Burton and VJ. Rayward-Smith. Worst Case Scheduling for Parallel Functional Programs. *Journal of Functional Programming*, 4(1):65–75, 1994.

[2] JM. Deschner. Simulating Multiprocessor Architectures for Compiled Graph-Reduction. In K. Davis and J. Hughes (eds.), *Glasgow Workshop on Functional Programming*, pp. 225–237, Fraserburgh, Scotland, August 21–23, 1989.

[3] MC. van Eekelen, EG. Nöcker, MJ. Plasmeijer, and JE. Smetsers. Concurrent Clean. Technical Report 89–18, Univ. of Nijmegen, October 1989.

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

[5] B. Goldberg. *Multiprocessor Execution of Functional Programs.* PhD thesis, Dept. of Computer Science, Yale University, April 1988.

[6] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In *Workshop on Graph Reduction and Techniques*, LNCS 279, pp. 94–113. Springer, 1987.

[7] K. Hammond, JS. Mattson Jr., and SL. Peyton Jones. Automatic Spark Strategies and Granularity for a Parallel Functional Language Reducer. In B. Buchberger and J. Volkert (eds.), *CONPAR'94*, LNCS 854, pp. 521–532, Linz, Austria, September 6–8, 1994. Springer.

[8] 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.

[9] K. Hammond and SL. Peyton Jones. Profiling Scheduling Strategies on the GRIP Multiprocessor. In H. Kuchen and R. Loogen (eds.), *Fourth Int. Workshop on the Parallel Implementation of Functional Languages*, pp. 73–98, RWTH Aachen, Germany, September 1992.

[10] M. Kesseler. Reducing Graph Copying Costs. In H. Hong (ed.), *PASCO'94*, pp. 244–253, Linz, Austria, September 26–28, 1994. World Scientific.

[11] P. López García, M. Hermenegildo, and SK. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In H. Hong (ed.), *PASCO'94*, pp. 133–144, Linz, Austria, September 26–28, 1994. World Scientific.

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

[13] SL. Peyton Jones and P. Wadler. Imperative Functional Programming. In *Symp. on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

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

[15] 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.

[16] C. Runciman and D. Wakeling (eds.). *Functional Languages Applied to Realistic Examplars: the FLARE Project.* UCL Press, 1994.

[17] P. Trinder, K. Hammond, H-W. Loidl, JS. Mattson Jr., A. Partridge, and SL. Peyton Jones. GRAPHing the Future. In J. Glauert (ed.), *Sixth Int. Workshop on the Implementation of Functional Languages*, Univ. of East Anglia, Norwich, U.K., September 7–9, 1994.