The HasPar Performance Evaluation Suite for GpH: a Parallel Non-Strict Functional Language

K. Hammond[1*], D.J. King[2], H.-W. Loidl[3], Á. J. Rebón Portillo[1] and P.W. Trinder[3]
[1] *School of Computer Science, University of St Andrews, UK (email: {kh,alvaro}@dcs.st-and.ac.uk)*
[2] *Motorola Research Labs., Reading, UK (email: David.King@motorola.com)*
[3] *Dept. of Computing and Electrical Engineering, Heriot-Watt University, Edinburgh, UK (email: {hwloidl,trinder}@cee.hw.ac.uk)*

SUMMARY

The ultimate purpose of parallel computation is to improve performance by exploiting hardware duplication. In order to achieve this improvement, it is essential to have a good understanding of real parallel behaviour. This paper introduces the HasPar integrated suite of performance evaluation tools for Glasgow Parallel Haskell (GpH), a high-performance parallel non-strict functional language. This suite provides a framework for assessing and improving parallel program performance that has been used successfully on a number of large functional programs.

The HasPar suite includes both idealised and realistic simulators for GpH. It also incorporates an instrumented parallel implementation that can be used on a range of architectures including both tightly-coupled multiprocessors and loosely-coupled networks of workstations. An important feature of the tools is that they allow costs to be attributed to the parallel program source using either static or dynamic cost attribution mechanisms, as appropriate. The resulting performance profiles can be visualised in a number of different ways, as illustrated in this paper.

KEY WORDS: Parallelism, Performance Evaluation, Functional Programming, Haskell

1. Introduction

1.1. Parallel Performance Evaluation

Parallelism is typically introduced into a program so as to improve its performance by reducing overall execution time [53]. In order to obtain the best performance, in any other than fully automatic schemes, the programmer must understand and improve the (often complex) parallel behaviour of that program. The purpose of parallel performance evaluation is thus to assist the programmer by providing information about a program's parallel behaviour that can be used to enhance that behaviour. This paper describes the HasPar suite of performance evaluation tools for GpH [68], a small extension of the non-strict purely functional language Haskell [26, 48, 51] to

*Correspondence to: School of Computer Science, University of St Andrews, North Haugh, St Andrews, KY16 9SS, UK.

support parallel execution. The HasPar suite aims to support performance evaluation activities at a high level of abstraction, whilst providing accurate information about actual parallel performance.

Parallel performance evaluation may involve either *monitoring* performance during execution, or *predicting* performance prior to execution [53, 66]. Program monitoring entails recording suitable *measurements* of parallel performance during execution of the program. Performance prediction entails either constructing an abstract *model* of program performance, or measuring a simplified instance of the program (*benchmarking/prototyping*).

Performance modelling can involve either simulation, analytical modelling or sometimes both. *Simulation* usually allows more detail to be incorporated in the model at the cost of generality. Simulation has played a significant role in the development of many parallel systems, and a very large number of simulation tools have consequently been constructed. Some recent examples include the commercial SES/workbench [64], IDRA for reporting ideal speedup in logic languages [11] and the Japanese EXCIT simulator for large-scale parallel programs [31].

In contrast to simulation-based approaches, *analytical cost modelling* allows more general conclusions to be drawn about some behaviour, perhaps even permitting comparisons of dissimilar systems, but normally at the cost of realism. Examples of analytical cost models include work on algorithmic skeletons [65, 3, 42], Petri nets [9], and stochastic models based on Markov chains [52]. Section 6.2 discusses this in more detail.

Measurement-based approaches for monitoring parallel performance include profiling and instrumentation [53]. *Profiling* involves sampling a program's state during execution, whereas *instrumentation* modifies the program to record interesting events as they occur. Profiling is thus less accurate, but less intrusive than instrumentation, but both generally require system-level support. Profiling or instrumentation support is available in many parallel systems, Examples include Quartz [1], AIMS [70], Paradyn [43], XPVM [13] and ParaGraph [24], as dicussed in Section 6.5.

Measurement-based approaches can also be used for performance prediction. Two such, primarily user-level approaches, are benchmarking and prototyping [53]. *Benchmarking* comprises selecting representative sample data that can be used to predict parallel performance for real data sets. *Prototyping* involves constructing a simpler version of a program that can be used to predict the parallel behaviour of the actual, and more complex, program.

The general approaches described above are aimed at either prediction or monitoring, but not both. In some interesting cases, feedback systems can be constructed where predicted performance is compared with actual performance, thereby combining prediction with monitoring. Information about the actual performance can then be used to improve the quality of the prediction, or to modify system parameters. Examples include Fahringer's feedback system [10] or Sarkar's simulated annealing approach [63].

The approach described in this paper involves a combination of modelling and measurement, used at different stages of the program development process for both prediction and monitoring, as appropriate. These mechanisms are used under the control of the programmer rather than automatically.

1.2. Parallel Performance of Non-Strict Functional Languages

The HasPar suite targets GpH (Glasgow parallel Haskell) [68], a simple parallel extension to the standard Haskell language. GpH provides a very high-level parallel programming model, delegating much of the responsibility of managing parallel evaluation to a sophisticated runtime system. For example, synchronisation between parallel threads is achieved by using implicitly shared graph nodes, and there is no explicit communication of data or other values. The task of identifying and specifying potential parallelism is, however, under programmer control. A key feature of GpH is that, like sequential Haskell, it is *non-strict* in the sense that function arguments are evaluated only if they are required. It also maintains the purely functional nature of sequential Haskell: programs are expressed without implicit side-effects.

Evaluating the performance of high-level non-strict languages such as GpH is a hard problem. Even in a sequential non-strict language the models of time and space behaviour are complex compared with strict languages [34]. Whereas in a strict functional language the arguments to a function are evaluated before the function is applied, as in conventional programming languages, in a non-strict functional language a function's arguments may be evaluated only when required in the function body, if at all. It follows that evaluation order and thus execution behaviour frequently do not match the programmer's intuitive model. It can thus be difficult to relate time and space costs to the source code that gave rise to it [8]. In a parallel setting, the problem is compounded by the introduction of multiple threads of control [14].

Furthermore, in the high-level functional language framework in which we are working, many of the traditional parallel performance metrics break down due to their relatively low level. For example, feedback about *precisely when* communication takes place is of much less use if the programmer has no explicit control over communication, though overall communication patterns may still be extremely interesting. We are therefore motivated to work at a much higher level than many parallel profiling tools, which simply record the overall machine state, as exemplified by the ParaGraph [24] system descibed in Section 6.5.4. Rather, we need to relate performance feedback to the parallel program, so that the programmer can take direct advantage of the information which is being provided. We will return to this issue in Section 3.1 where we discuss high-level source-related profiling.

1.3. Structure of This Paper

The remainder of this paper is structured as follows. Section 2 introduces performance evaluation for GpH, discussing the basic GranSim simulator, with the GranSim-Light variant which can be used to investigate parallel behaviour in an idealised setting. Section 3 considers ways in which parallel costs can be systematically related back to the source program, using either parallel *cost centre profiling* or *strategic profiling*. Section 4 discusses performance monitoring in our real parallel system (GUM). Section 5 considers pragmatic issues, including an assessment of the costs associated with the different evaluation tools. Finally, Section 6 discusses related work, and Section 7 concludes.

2. Evaluating the Performance of GpH Programs

This section introduces the main features of Glasgow parallel Haskell (GpH) and describes our basic performance evaluation tools in the context of the GranSim simulators. Subsequent sections introduce refinements that allow attribution of cost at the source code level, and discuss performance monitoring for the GUM parallel implementation of GPH.

2.1. GpH: Glasgow Parallel Haskell

Haskell is a non-strict purely functional language [26, 48, 51], which has been widely adopted by the functional programming community. GpH is a simple extension to sequential Haskell whose implementation builds on the Glasgow Haskell compiler (GHC), a state-of-the-art optimising compiler for the full Haskell language.

The GpH parallel programming model is based on the use of two simple constructs:

```
c  `par` e        -- parallel composition
e1 `seq` e2       -- sequential composition
```

The `par` function is a simple extension to Haskell: the infix application ``c `par` e`` specifies that `c` is a possible candidate for evaluation in parallel with `e`. The `seq` function, however, is part of standard Haskell 98: ``e1 `seq` e2`` specifies that `e1` is evaluated before `e2`.

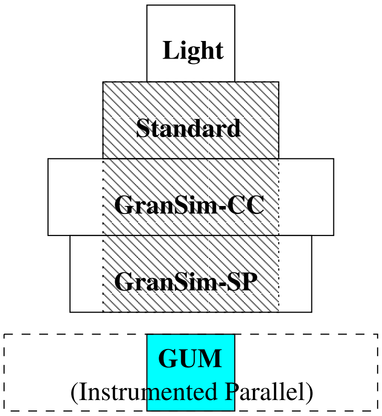
Parallel composition uses the "evaluate-and-die" model developed by Peyton Jones et al. [50]. In this model, `c` (a sub-expression of `e`) is *sparked* as a potential parallel thread, while normal execution continues the evaluation of `e`. If any processor becomes idle following the sparking of `c`, it may steal the *spark* for `c` and evaluate it using a new parallel thread. The result will then be communicated back to the thread that is evaluating `e` as soon as it is needed by that thread. This consequently implements a form of *lazy thread creation* [45]: threads are created if and only if there are idle processors on which they can be executed, and otherwise remain dormant. More detailed descriptions of this mechanism can be found in previous papers (e.g. [68, 18]).

2.1.1. Evaluation Strategies

Evaluation strategies [68] build on the very simple GpH model described above. An evaluation strategy is a function that specifies some kind of abstract sequential or parallel behaviour. Such strategies can be nested if required to form more complex strategies, by using them as arguments to other strategies. New strategies can also be constructed from existing ones using higher-order strategy-forming functions. This forms a powerful, programmable way to introduce new parallel abstractions, such as specific algorithmic skeletons [23], or new user-defined constructs. We have used this ability for a number of large applications [38, 67]. An important property of such strategies is that they are type-safe: all strategies conform to the normal Haskell polymorphic typing rules.

It is possible to attach an evaluation strategy to an arbitrary expression by exploiting the `using` function. So, for example, ``quicksort l `using` parallel all`` specifies that each element of the sorted list should be completely evaluated in parallel (note that the `all` strategy is equivalent to the `rnf` strategy defined in our earlier papers [68]).

The effect of using evaluation strategies is to achieve a clean separation between the algorithmic code that specifies *what* a computation is to produce, and the behavioural code that specifies *how* that computation is to proceed. A major advantage of this



Tool	Description
GranSim-Light	An idealised simulation with zero communication costs and an infinite number of processors [35]
GranSim	A tunable, realistic simulation [19]
GranSim-CC	A detailed simulation costing user functions [20]
GranSim-SP	A simulation costing logical tasks [30]
GUM	An instrumented parallel implementation of GpH [69]

Figure 1. The HasPar Performance Evaluation Suite

high-level approach to parallelism is that the separation of behaviour from algorithm supports prototyping and experimentation. However, this creates a further level of abstraction that must then be bridged in order to understand parallel performance. We will return to this issue in Section 3.2.

2.1.2. The GpH Parallel Execution Model

GpH implements a parallel *graph reduction* model of execution [41], in which parallel threads concurrently modify a logically shared graph structure (effectively a virtual shared heap). Threads are allocated dynamically to idle processing elements (PEs), which comprise a virtual processor plus appropriate local memory and other resources. All inter-thread communication occurs as a consequence of accessing the graph: graph nodes are locked when they are evaluated by a *running thread*; other threads attempting to evaluate such nodes then become *blocked* and are suspended until the evaluating thread updates the node with the result; at that point, the thread becomes *runnable* and may be executed if the PE is idle, or else be placed on the local queue of *runnable* threads. These activities form the basis for the parallel activity profiling that will be introduced in the following sections.

2.2. The HasPar Performance Evaluation Suite

The HasPar suite (Figure 1) is designed to support a simulation-first approach to parallel program development, in which idealised parallelism is gradually refined to suit the target architectures. This approach has the advantage of detecting undesirable parallel behaviour very early, and allowing new hypotheses to be checked relatively painlessly. The suite comprises 5 major tools, ranging from idealised simulation to real parallel execution. The first four tools are, in fact, variants of a single basic simulation tool, collectively called GranSim. GranSim-Light gives a simplified simulation, mainly used for initial design. GranSim-CC (static "cost centre" profiling) and GranSim-SP (dynamic "strategic" profiling) can be used to give more detail than plain GranSim, but otherwise provide identical simulation options (Section 3). This is indicated by hatching in (Figure 1). While in principal, our instrumented parallel system (GUM) could produce more detailed information than any of the simulations, in practice it is confined to producing only a subset of that information (Section 4). All of the tools produce profiling output in a common format that is dumped to a log file. This output can then be visualised in a number of different ways as described in the following sections.

In addition to the specifically parallel tools described above, we have also found conventional sequential profiling using cost centres [62] to be a useful first step in parallelising many programs: generally there is no point in parallelising a program that contains gross sequential inefficiencies. It can, however, be counter-productive to attempt to parallelise a program that has been over-optimised for sequential evaluation — such programs frequently contain unnecessary sequential dependencies that must then be removed from the parallel program.

2.3. GranSim and GranSim-Light — Simulators for Parallel Haskell

GranSim is an event-based simulation of the parallel execution of GpH programs. The GranSim simulation is a very close approximation to the actual GUM implementation: computation costs are simulated by instrumented execution of the actual parallel system code; communication is, however, abstracted under the control of a number of tunable parameters. This combination permits very accurate simulation of computation costs, while supporting a great deal of flexibility over costing interesting communication issues. The high quality of the simulation allows a relatively painless transition from the simulator to the real GUM system.

The main characteristics of GranSim are

- *Flexibility:* GranSim provides a large number of parameters for tuning, for example, the communication behaviour of the system. This makes it possible to model a wide range of parallel architectures.
- *Accuracy:* Realistic code generation via the GHC front-end, combined with the accurate simulation of communication and blocking achieves a high level of accuracy in the simulation. In contrast to many simple simulators (e.g., [57]), computation is measured in machine cycles rather than abstract combinator reduction steps. Real optimised code is instrumented using an instruction cost model, giving a close correspondance between GranSim and GUM profiles (see [35][Section 3.6]).
- *Determinism:* GranSim simulations are repeatable. There is no random component. While this makes the simulator sensitive to timings of particular events, it does allow interesting behaviour to be isolated by repeated experiments, perhaps at different levels of detail.
- *Visualisation:* GranSim is supplemented with a set of visualisation tools in order to aid the programmer in understanding the dynamic behaviour of the parallel program. Since GUM produces the same kinds of profiles as GranSim, the same visualisation tools can also be used for real parallel executions of GpH programs, allowing direct comparison between simulation and actual execution. The visualisation tools are discussed in Section 2.3.2.

The primary deficiency of GranSim is that it makes no attempt either to simulate the memory hierarchy, or to simulate garbage collection behaviour. The former is encapsulated within the instruction cost model. The latter was deliberately omitted from the design in order to avoid introducing a complex new dimension to the simulation. While this is clearly a limitation on the accuracy of the simulation, both in terms of execution costs, and in terms of perturbation of simulated events including communication, incorporating garbage collection costs in the standard simulation would require the programmer to perform an additional range of experiments with different memory system settings. Our belief is this is likely to be most profitably performed on the actual system rather than in the simulation.

A final important restriction is the lack of a topological communication model: GranSim assumes uniform inter-PE latency across the parallel machine. While this may not be a realistic assumption for parallel architectures such as physical hypercubes or meshes, it does represent a common class of architecture. We hope to be able to incorporate a suitable topological model in due course.

The GranSim-Light variant abstracts from communication costs and assumes an infinite number of processors, but retains the full standard execution cost model. It is mainly used to provide information about the design of parallel algorithms by focusing on the amount of parallelism that is inherent in a program [38]. Although GranSim-Light trades flexibility and accuracy in favour of idealised simulation, it does however, retain the same execution cost model as standard GranSim.

2.3.1. The Implementation of GranSim

The GranSim simulator has four primary components:

1. a static execution cost model, which attaches abstract instruction costs to basic blocks;
2. an event-driven threaded runtime system, which simulates multiprocessing;
3. a dynamic communication cost model, which simulates inter-processor communication;
4. instrumented libraries and runtime system, that cost execution dynamically.

The GranSim Execution Cost Model. GranSim is an instrumented version of GHC. To the extent that the simulator executes the exact instructions that would be performed on the host machine (including all optimisations, branches, and library function calls), it is an emulation of the sequential execution behaviour. Rather than costing the exact instructions executed by any particular computer architecture, however, the GranSim execution model is an abstraction of real execution cost in terms of the following 5 classes of instruction: 1) ALU/integer instructions; 2) floating-point instructions; 3) memory load instructions; 4) memory store instructions; and 5) branch instructions. Each of these instruction classes can be given independent costs at run time, so allowing different architectures to be simulated on the host machine without re-compilation or per-instruction cost models. Note that no attempt is made to simulate low-level architectural details such as cache behaviour, instruction pipelining, or multiple-issue. These details are amortised into the overall cost for each class of instruction.

This relatively high level of abstraction allows the simulator to be easily re-targetted to new computer architectures without needing target code generators to be written. It is a reasonable abstraction over modern instruction sets (especially RISC machines), where a design goal is usually to produce a common CPI (cost in terms of Cycles Per Instruction) for each instruction class [47].

Structure of the GranSim Simulation. The GranSim simulation is event-driven. The event queue contains all outstanding events for all simulated PEs ordered by the simulated time of occurrence. Event types include:

FINDWORK:	Search for work on remote PEs
MOVESPARK:	Move a sparked graph node to a new PE after a FINDWORK
MOVETHREAD:	Move a runnable thread to a new PE
START:	Create a new thread from a sparked graph node
RESUME:	Resume a thread after it has been suspended
FETCHNODE	Send a request for a graph node (or sub-graph)
FETCHREPLY	Reply to a FETCHNODE event
GLOBALBLOCK	Record that a thread has become blocked on a remote node
UNBLOCKTHREAD	Unblock a thread that previously blocked on a remote node

Events include information about the PE and thread to which they apply in addition to the simulated time at which they occurred.

Each simulated thread is assigned to a single simulated PE, and is executed as an actual concurrent thread on the host machine. This concurrent thread transfers control back to the event manager when it generates an event, or when it might need to context switch. It follows that each PE must maintain its own independent clock, and that the simulated time on any individual PE will normally be in advance of the global simulated time. Consequently, events for different PEs may be simulated "out-of-order". It is a simple matter to reorder events into an absolute order at the end of the simulation.

The GranSim Communication Model. GranSim simulates a virtual shared heap with implicit communication and local cacheing. The underlying communication model is packet-based, though it would be straightforward to change this to an actual shared-memory model, as used in our new shared-memory implementation [21].

If a thread requires the value of a node that is not cached on the local PE, it must be fetched from the owning PE. The simulation provides a number of options for the resulting communication. Nodes may either be fetched incrementally as they are demanded (*incremental fetching*), or a related sub-graph may be fetched at the same time (*bulk fetching*) [36]. The former is more suitable for low-latency machines; the latter is more suitable for machines with higher latencies. Similarly, there are a number of options for how and whether the system attempts to amortise communications latency by executing existing threads or creating new threads. These options were explored carefully in an earlier paper [36], and we plan to integrate the most successful combination of these into the next version of GUM.

Instrumented Libraries and Runtime System. Since each of the standard libraries is written in Haskell and compiled with full GranSim instrumentation, it follows that they are instrumented in exactly the same way as normal Haskell code. Calls to the operating system (e.g. for I/O) are not costed: achieving accurate costings would amount to providing a cost model for the operating system, which is beyond

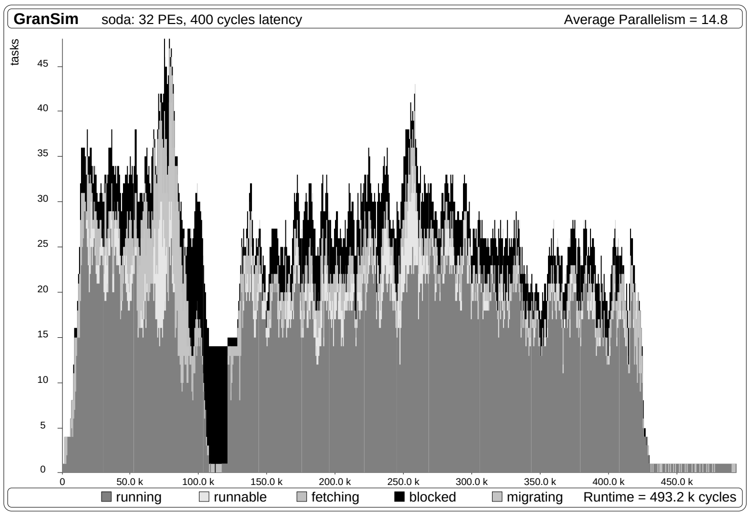


Figure 2. GranSim: Overall activity profile for `soda`

the scope of this simulation. Runtime system code is costed similarly to the standard libraries, except that calls to costing macros must be inserted manually. The same approach can be taken to cost foreign language functions.

2.3.2. Visualising GranSim Activity

The visualisation tools produce two different kinds of profiles: *activity profiles* (e.g. Figure 2) show the number of active threads at a particular point during the execution; *granularity profiles* (e.g. Figure 5) collect information on the size of all executed threads.

GranSim Activity Profiles. For general performance evaluation, activity profiles are normally more useful than granularity profiles. Three different levels of detail can be visualised:

- *Overall activity profiles* present a global picture of the computation, counting the number of running, runnable, blocked, fetching and migrating threads at each point in the execution. Figure 2 shows such a profile for `soda`, a simple test program that performs a parallel search of a grid looking for dictionary words [58]. The x-axis represents execution time, and the y-axis represents individual threads. The different colours in the graph (shown as shades of gray in this paper) represent the number of threads that are in one of the five aforementioned states. The green (lowest, medium gray) area is especially important since it indicates how many threads are running at any point in time. Drops in PE utilisation are shown as dips in the green area, often accompanied by an increased number of blocked threads, visualised as a larger red (highest, dark grey) area.

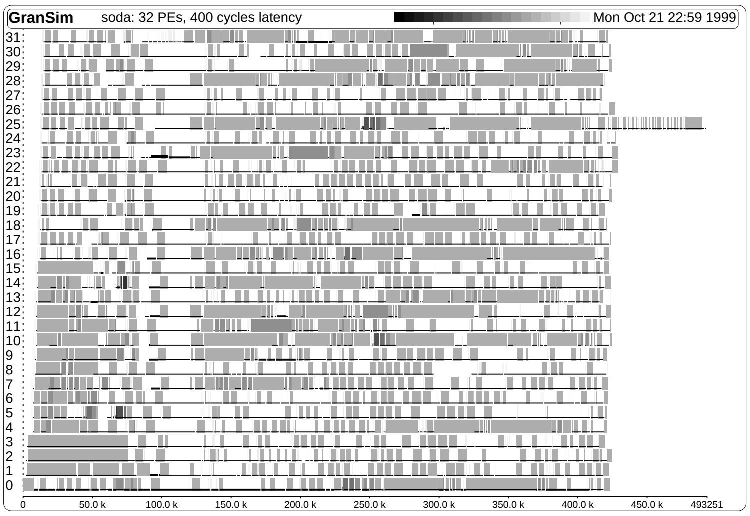


Figure 3. GranSim: Per-PE Activity Profile for `soda`

Such a drop in utilisation can be clearly seen at the 100K instruction cycle mark in Figure 2. Immediately before this happens, there is a dramatic increase in the number of threads that are migrated between PEs (normally a very rare event). This indicates that runnable threads are being moved from otherwise busy PEs to ones that are idle in a desperate, and in this case ultimately futile, attempt to achieve a better load balance.

- *Per-PE activity profiles* are visualised as strip charts, where each strip contains information about the activity of a single PE. Different shades of green (shown as shades of gray in the printed paper) indicate the PE load in terms of the number of runnable threads. Red areas (left blank) indicate idle time. A blue bar inside the strip (shown here in black) indicates the number of blocked threads on the PE. These profiles can be used to obtain additional information about load balance and migration.
  Figure 3 shows the per-PE profile for the same example as the overall activity profile above (Figure 2). Overall, for this example, the load is quite evenly balanced. The profile clearly shows which PEs are idle at the point where the drop in utilisation occurs, but gives no indication of why those PEs are idle. The effect of thread migration is shown by the fact that PE 25 rather than PE 0 is executing the main thread at the end of the computation.
- *Per-thread activity profiles* provide the most detailed performance information. Each thread is represented by a horizontal line, whose thickness and shading shows the state of the thread in terms of the 5 basic activity classes. Figure 4 shows the per-thread profile for PE 0 using the same example as before. There is one long-lived thread (thread 0, the main thread); the remaining threads are fairly short-lived and execute consecutively, so giving a stepped thread profile. The profile also reveals that the execution of these threads is more or less continuous, without long periods of either blocking or communication.

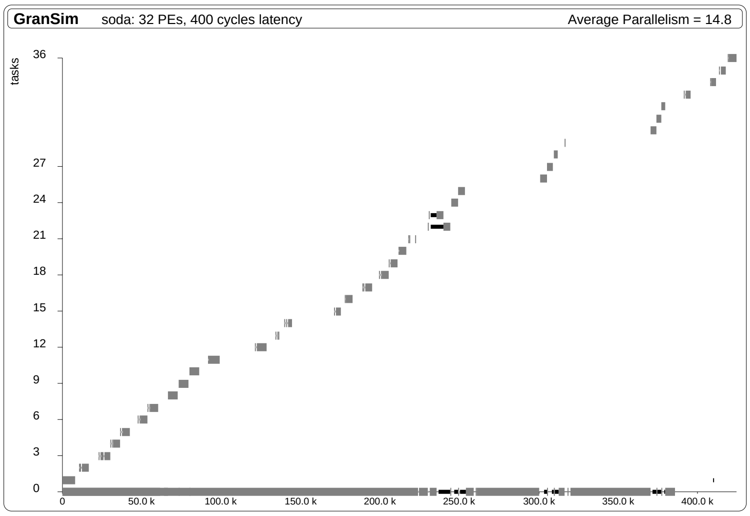


Figure 4. GranSim: Per-thread Activity Profile for `soda`

Although they give the least detailed picture of the parallel computation, we have found that overall activity profiles are by far the most useful of the three types of activity profile. This is because they provide an at-a-glance view of parallel activity. Similar profiles were used by Runciman and Wakeling for the HBCPP simulator [57], and in our work on the parallel GRIP machine [22], and are also found in tools such as ParaGraph [24], which is discussed in Section 6.5.4.

While per-PE profiles permit detailed study of load balance, they are usually too crowded to give a good overview of the computation. Since GpH does not yet provide mechanisms to control locality and distribution, we have not yet used this kind of profile in anger for many programs. We intend to work on this area in the near future. Again, this kind of profile is often supported by standard parallel profiling tools such as ParaGraph.

Finally, although per-thread profiles provide interesting information about thread lifetimes and activity, programs normally produce more threads than can be reasonably visualised simultaneously. An environment that supports the possibility of collapsing threads, such as [7], would make this kind of profile much more useful.

The main shortcoming of all these profiles from an application programmer's perspective is that they do not associate parallel behaviour directly to the source code. This deficiency is made worse by the fact that non-strict evaluation can make even the sequential evaluation order hard to predict. Section 3 introduces two complementary mechanisms that can be used to overcome this deficiency.

Granularity Profiles. Granularity profiles visualise the total runtime of all threads in the form of a histogram. For example, the left hand side of Figure 5 shows a granularity profile for the same `soda` program as before. Here the x-axis, representing runtime, is divided into a number of discrete intervals. The y-axis represents the number of threads whose total runtime fall into a specific interval, thus yielding a

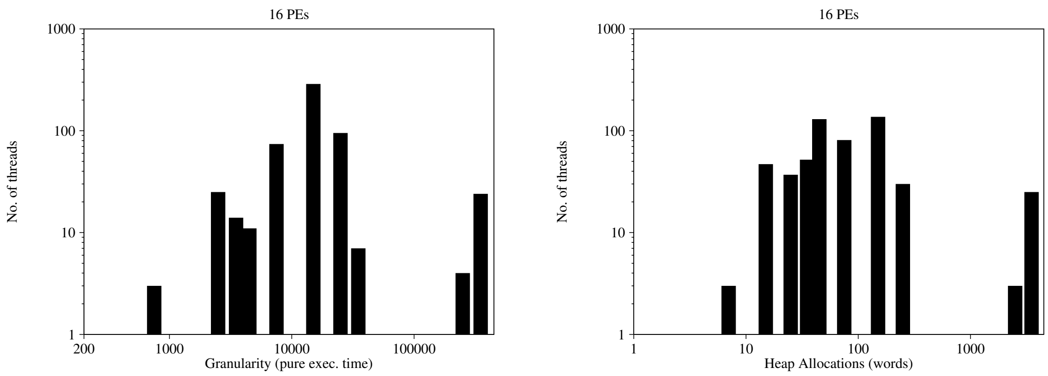


Figure 5. Granularity profiles for thread runtime and heap allocation

histogram. This histogram can then be used, for example, to determine whether the parallelism is excessively fine-grained, with consequent context-switching overhead.

Although these kinds of profiles are most useful for visualising thread runtime, they can also be used to show other aspects of the computation that can be represented using grouped intervals. For example the right hand side of Figure 5 profiles thread *space usage*, in terms of the total heap allocated by each thread. This example displays a close correlation with the runtime profile. We have verified that this is a common feature of such profiles [19].

3. Attributing Cost

As noted earlier, the main deficiency of the GranSim profiles shown in Section 2 is that it can be difficult to relate interesting behaviours to their causes in the program source. This observation has motivated us to extend GranSim with facilities for labelling expressions in the parallel program and for displaying them in the (modified) activity profile.

Attributing execution cost is unfortunately non-trivial in a sequential lazy functional program, since the execution order is determined dynamically. Demand-driven evaluation means that there may be frequent “out-of-line” calls to obtain the values of arguments that in a conventional strict language would have been evaluated before a function call. Higher-order functions and extremely aggressive optimisation add additional complexity. Consequently, simple methods such as placing a marker around a region of interest may be ineffective. The problem is compounded in a parallel setting by the need to track the behaviour of a large number of concurrent threads.

Regardless of how such information is eventually obtained, it represents an essentially static attribution of cost: for example, it may be possible to indicate what function/expression gave rise to what computation, and at what point in the overall execution this happened. In a parallel context, this information is useful in determining which functions gave rise to which parallel computations. More dynamic cost attribution is, however, appropriate when determining which threads of control gave rise to which parallel computations.

The HasPar suite supports both static and dynamic cost attribution: GranSim-CC is a parallel cost-centre profiler (described in Section 3.1); while GranSim-SP is a

Prepared using speauth.cls

dynamic parallel profiler (described in Section 3.2). To our knowledge the HasPar suite is the first such integration of both static and dynamic cost attribution in a parallel simulation setting.

3.1. GranSim-CC — Parallel Cost Centre Profiling

GranSim-CC adapts standard cost centre profiling techniques [62] to the GranSim simulation setting in order to allow dynamic parallel events to be accurately related to the GpH source program.

3.1.1. Cost Centre Profiling

The sequential cost centre profiling scheme [61, 62] allows sections of code to be annotated with a *cost centre* to which all costs associated with executing that section of code are to be attributed. Cost centres are statically scoped and may either be introduced implicitly by the compiler or explicitly by the programmer using an `_scc_` annotation. For example,

```
module M where
  f n = _scc_ "fcost" fbody where fbody = g (...)
  g n = _scc_ "gcost" gbody where gbody = ...
```

attaches the cost centre called `M.fcost` (cost centre `fcost` from module `M`) to the expression `fbody`. All costs associated with executing `fbody` will therefore be attributed to `M.fcost`. Costs associated with executing the subsidiary function `g` will however be attributed to `M.gcost`. If the `_scc_` to `gcost` were omitted then the cost would be attributed to `MAIN.MAIN` instead (cost centre `MAIN` from module `MAIN`).

To avoid unfairly attributing the costs of shared computations to one function rather than another, all potentially shared global constants within a module are assigned to a special cost centre. Of course, if a constant is only used by one function, then it would be reasonable to attribute all of its cost to that function. However, in general, it would require global flow analysis to determine this, and this is not done in the current implementation of cost centres. Following the program run, a cost centre log is produced that shows the percentage of execution time spent in each cost centre.

Parallel Cost Centre Profiling. GranSim-CC uses the basic sequential cost centre mechanism to maintain the current cost centre for each parallel thread. This information can be attached to the execution trace as required to provide precise information about the cause of each event.

3.1.2. Implementation of Cost Centre Profiling

The Glasgow implementation of sequential cost centre profiling is fairly simple in principle, though there are a number of subtle issues concerning precise cost attribution, most notably concerned with attributing costs to global shared constants [62].

A register is used to record the *current* cost centre throughout the program execution. This is updated to reflect the program's new activity when either explicit

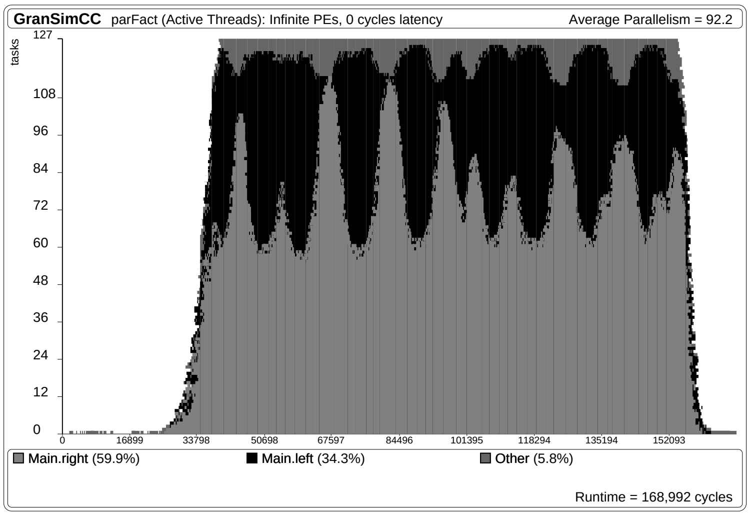


Figure 6. GranSim-CC: Cost-Centre Activity Profile for a Divide-and-Conquer Program

or implicit _scc_ constructs are executed. Interval-based sampling is used to monitor general high-level activity, with event-based sampling being used to monitor low-level activity such as the number of times a particular closure type is used.

GranSim-CC adapts the basic cost centre profiling mechanism to the GranSim simulator setting. Each thread has its own current cost centre register, which must be saved and restored with the rest of the thread's context. Since GranSim traces are event-based rather than sample-based, as for sequential cost centre profiling, each event must also record the current cost centre.

The 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. Although it is not done at present, cost centre information could, if needed, also be recorded for sparks.

3.1.3. Visualising Cost Centre Profiles

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 as shown in Figure 6.

3.2. GranSim-SP — Strategic Profiling

Strategic profiling [30] relates the costs incurred by individual parallel threads to the evaluation strategies that created them (see Section 2.1.1). Since evaluation strategies

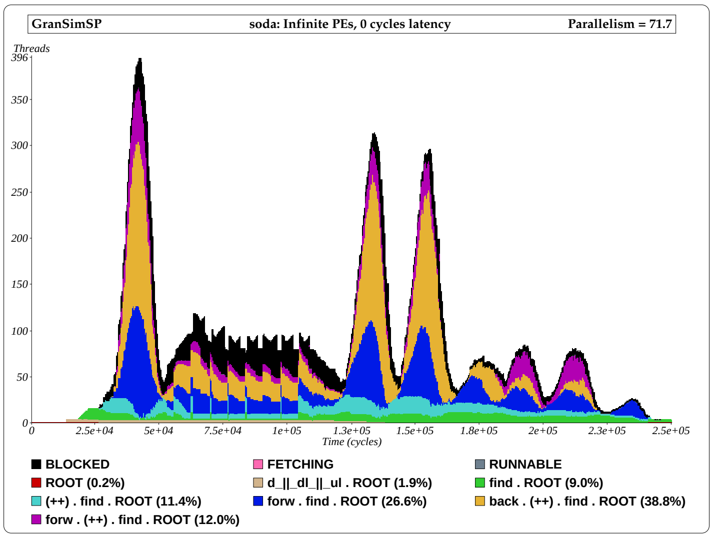


Figure 7. GranSim-SP: Strategic Activity Profile for `soda`

influence parallel behaviour, determining the effectiveness of the strategies that are used in a program can be an important part of the performance tuning process.

The function `markStrat` of type `markStrat :: String -> Strategy a -> Strategy a` is used to name evaluation strategies. It is normally used in conjunction with the `using` function. For example, `expression ‘using‘ markStrat "A" strategyA` labels each and every thread created by `strategyA` with “`A`”.

In addition to recording names for threads, GranSim-SP records the child-to-parent relationship between threads. This information has many uses: one is to provide names for strategies; another, less obvious use, is to allow different call-paths of a strategy to be distinguished. But perhaps the most important use is when a thread blocks waiting for a child thread; strategic profiling allows the creation point of the child thread to easily be determined. This is useful information, since using strategies that avoid thread blocking can often give speedups.

Figure 7 shows a GranSim-SP activity profile for the “soda” word puzzle solver program [58]. This profile clearly distinguishes the threads that relate to the labelled strategies in the program. The profile allows the user to easily determine those stategies that take the most time, and their timing relationships vis á vis the threads created by other strategies.

3.2.1. Strategic Profiling versus Cost Centre Profiling

Although both strategic profiling and cost centre profiling provide user-level information about dynamic behaviour, there is a significant difference in the scoping of `markStrat` and `_scc_`. The scoping provided by `_scc_` is a hybrid of *lexical scoping* [8], which has been found to be the most appropriate for a lazy language like Haskell [61]. In contrast, `markStrat` uses *evaluation scoping*, which is the most appropriate when labelling strategies alone. The difference between the two scoping rules is highlighted

Table I. Differences between GranSim-SP and GranSim-CC

	GranSim-CC	GranSim-SP
Cost attribution	static	dynamic
Scoping	"lexical scoping"	"evaluation scoping"
Refers to	normal user code	strategic code
Needs support in	Code generation and RTS	RTS
Notion of threads	implicit	explicit

by the following example:

```
exprCC `using` _scc_     "A" strategyA     -- cost centre
exprSP `using` markStrat "A" strategyA     -- strategic
```

Under lexical scoping (cost centre profiling) as shown in the first line, the cost of evaluating strategy `strategyA` is attributed to `A`. The cost of evaluating `exprCC` is, however, attributed to the enclosing cost centre. In contrast, under evaluation scoping (strategic profiling) the cost of evaluating both this instance of `strategyA` and `exprSP` is attributed to `A`. Table I summarises the main differences between GranSim-CC and GranSim-SP.

3.2.2. The Implementation of GranSim-SP

GranSim-SP is based on the basic GranSim implementation. Two extra pieces of information are recorded for each spark and thread: the name provided by `markStrat`; and the parent thread identifier. Whenever a call to `markStrat` is encountered, the given strategy name is recorded in the currently active thread's context. When a spark is created, the current strategy name and thread identifier are attached to the spark. When sparks are subsequently turned into threads, the name and thread identifier recorded with the spark are both passed to the freshly created thread; the thread identifier that is recorded is thus that of the parent thread – the one that created the spark. These two pieces of information are recorded by extending the `START` event.

4. Monitoring the GUM Parallel Implementation

The GUM implementation maps GPH's virtual shared graph onto a variety of parallel architectures using a retargetable message-passing interface. This interface has been ported to PVM [12], MPI [40] and proprietory communications systems. We are also investigating a direct shared-memory implementation [21].

GUM provides three types of performance metrics: i) per-PE execution-time profiles; ii) restricted GranSim-style per-PE activity profiles; iii) per-PE space usage, recorded during garbage collection.

4.1. Per-PE Execution Profiles

Because GUM runs on a multitasking multiprocessor that may be executing other applications, monitoring an application's performance introduces a number of

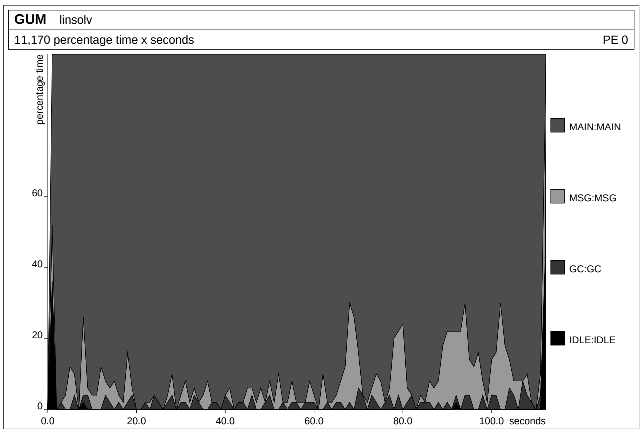


Figure 8. GUM: Per-PE execution profile for `linsolv` *(one PE of four executing)*

problems not encountered in the simulators. Some of these problem are minor, e.g. combining multiple GranSim log files to produce a single overall activity log. Others are more significant, in particular i) maintaining time on distributed multitasking PEs; and ii) ensuring that the cost of measurement has minimal impact on the behaviour of the program.

Maintaining global time is a well-known problem in distributed systems [33]. We use a pragmatic solution that records the PE clock time at a known synchronisation point, then records later times as an offset from this time. We assume that operating system effects do not affect the times recorded in each PE. For accurate performance monitoring, it is also necessary to ensure that the GUM process is the only one executing.

In order to minimise impact on program behaviour, GUM restricts the amount of information that is recorded at execution time. This increases the importance of the simulators and detailed visualisation tools. Some systems such as AIMS [70] (Section 6.5.3) attempt to eliminate such timing errors instead. It would be interesting to implement such techniques for GUM.

GUM uses standard *cost-centre profiling* [61] internally to record the coarse activity of each PE during execution. The results of profiling can then be visualised using standard tools. For example, Figure 8 gives the PE execution profile for one of four PEs evaluating a simple linear equation solver, `linsolv`, on a Sparc-2 workstation. Such a profile allocates time to four cost centres only:

Main time spent executing user code;
Msg time spent packing and unpacking messages etc.;
GC time spent garbage collecting;
Idle time spent in the scheduler, plus all remaining execution time.

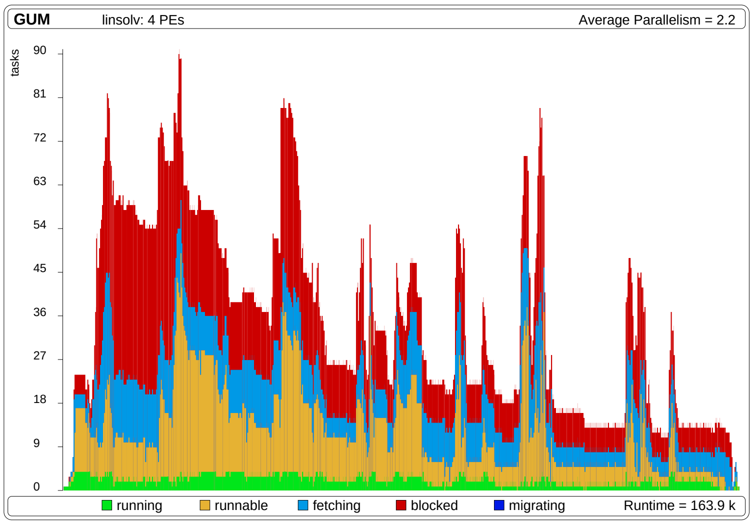


Figure 9. GUM: Overall Activity Profile for `linsolv`

4.2. Per-PE and Overall Activity Profiles

An alternative style of profiling is supported by per-PE activity profiles. In this mode, each PE records a subset of GranSim events, notably thread creation, blocking, and termination, into its own separate log file. The individual log files are merged following the parallel run into a form that can be processed by the GranSim tools described in the previous section. Figure 9 shows the activity profile for the same linear equation solver used in Figure 8.

4.3. Per-PE Garbage Collection Statistics

The garbage collection statistics reported by the sequential runtime system are also useful in GUM. The sequential runtime system records the amount of heap allocated between collections, the live heap following the collection (residency), the count of externally reachable pointers (garbage collection roots), the time spent in each garbage collection etc., as well as overall times for system initialisation, garbage collection and other activities. Times are recorded both as elapsed times and as virtual times. For GUM, all this information is recorded on a per-PE basis. The GUM initialisation time is the time between the PE task starting and the start of mutation. Time recorded as useful activity includes the time spent performing reduction, communicating or even idle, but excludes time spent garbage collecting.

5. Pragmatics

5.1. The Comparative Cost of Each Simulation

This section considers the costs of performing the simulation for a ray tracer that plots the intersection of a number of rays of light with a scene. The ray tracer is

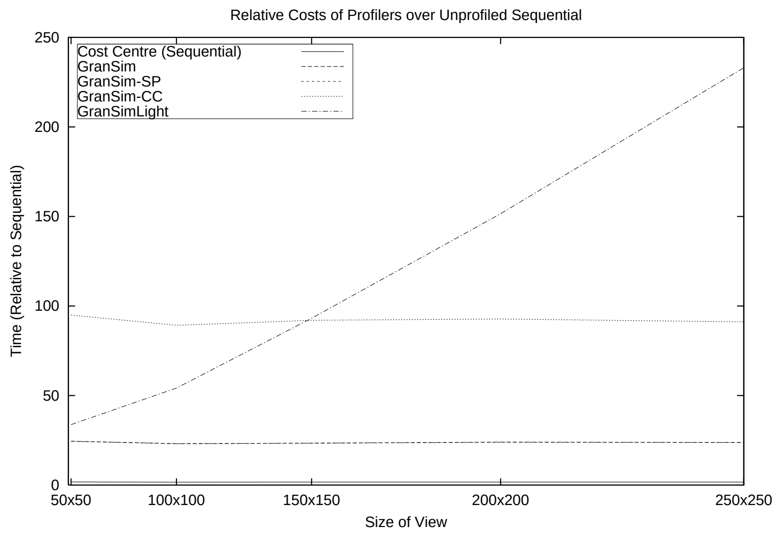


Figure 10. Simulation Time Cost Compared with Sequential Execution: `ray`

moderately sophisticated in that it handles a number of shapes of object, deals with both transparent and opaque objects, and computes reflections on the surface of the objects. The test scene used here comprises three overlapping spheres on a blue background. The parallel version of the program creates one potential thread for each ray that is to be traced with the actual number of threads created being under dynamic runtime control.

All results reported here were obtained on identical 450MHz Pentium II processors running Linux kernel 2.0.36 (RedHat 5.2). All timing results represent the average of 10 runs on a quiescent machine, and record CPU (user time). In order to minimise the effect of garbage collection on the timing results, each run was allocated 200MBytes of dynamic heap memory. The profilers would normally run using considerably less memory, of course.

5.2. The Time Cost of the Simulations

5.2.1. The Relative Costs of the Simulations for Varying Input Sizes

Figure 10 shows the relative costs of each of the profilers in the HasPar suite compared with the fully optimised sequential execution time of the program (whose cost is set to 1 for each data point). In this figure, the x-axis represents the size of the view in pixels with the y-axis giving the execution time relative to the sequential case. For a view comprising a square grid of 50x50 rays, the sequential execution time is 0.245s; for 250x250 rays, the sequential execution time is 6.8s on our test scene. Clearly, it would not be worth parallelising such a small program, however, tests on our 64-processor Beowulf have demonstrated good scalability on larger input sizes. The simulations that have been costed are thus representative of live simulations that might be required by an applications programmer.

It is interesting to observe that, apart from GranSim-Light, the cost of each of the profilers is close to a constant factor of the sequential execution cost. The slightly higher relative costs at the 50x50 view size (most obvious for GranSim-CC) are probably due to additional garbage collection requirements. Generally, sequential cost centre profiling is roughly twice as expensive as optimised sequential execution; GranSim and GranSim-SP both cost roughly 25 times the sequential cost; GranSim-CC is roughly 90 times the cost of optimised sequential code, or 3.5 times the cost of standard GranSim; the cost of GranSim-Light varies between 34 times the optimised sequential cost for a 50x50 view and 265 times the optimised sequential cost for a 250x250 view, or between 1.25 and 10 times the cost of standard GranSim.

The extremely high cost of GranSim-Light profiling for larger problem sizes (which appears to be $O(n\ log(n))$ rather than the $O(n)$ for the other profilers) can be attributed to the increased cost of context-switching between simulated threads, the fact that threads are created eagerly rather than lazily in GranSim-Light, and to the use of fair scheduling for small time slices on a large number of threads (so destroying good cache behaviour for the simulator). Since GranSim-Light is mainly used during the initial parallelisation phase, where it will be used on small data sets, these costs are usually acceptable. Clearly, it could be prohibitive to use GranSim-Light on large data sets.

The lower, but still relatively high cost of GranSim-CC profiling compared with standard GranSim is explained by the need to dump much more information to the log file and by the additional events in the main simulation loop: unlike the sequential cost centre profiler, which performs periodic sampling, each and every change to a cost centre is treated as an event, which is then recorded in the log file. It would be possible to alleviate this cost by, for example, aggregating cost centres, or by restricting the dumping of events to the range of interest (which is rarely the entire program).

Both GranSim and GranSim-SP have very similar costs. This is because GranSim-SP is a very lightweight extension to GranSim. Compared with standard GranSim, GranSim-SP simply needs to record a small piece of information when a thread is marked by `markStrat`, and add a short string identifying the strategy that created the thread to each START event in the log file.

5.2.2. Costs for Varying Numbers of PEs

Figure 11 shows the averaged costs for GranSim (left) and GranSim-CC (right) when varying the number of simulated PEs. The initial sharp rise in the figure represents the cost of doing any parallel simulation (the standard simulation for one PE requires

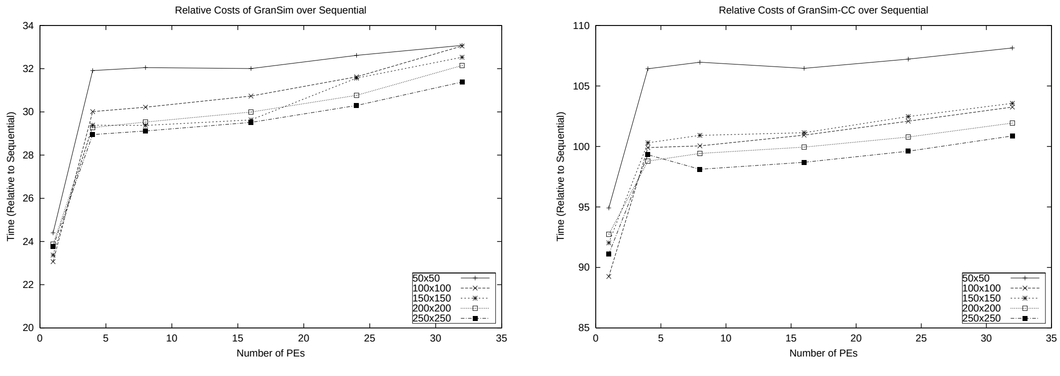


Figure 11. Simulation Time Costs with Varying PEs: `ray`

Prepared using speauth.cls

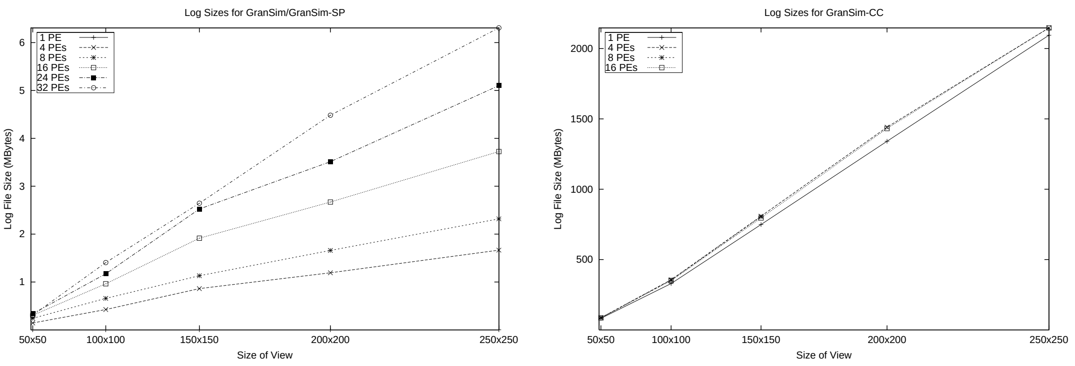


Figure 12. Log File Sizes for Varying Views and Number of PEs: `ray`

only one simulation thread). Thereafter, simulation costs rise slightly with the number of simulated PEs, representing the additional cost of context switching among the simulated threads. Generally, the costs of the simulation are relatively greater for small input sizes than for larger ones. This perhaps represents the cost of garbage collection in the sequential case.

5.3. Log File Sizes

Figure 12 shows the log file sizes when varying the size of the view and the number of PEs for GranSim (left) and GranSim-CC (right). For GranSim, the size of the 1-PE log file is constant, since it records only a START and END event for the main thread. For more processors, the log file size increases with the number of PEs. This is to be expected since if more PEs are available, then more simulated threads can be executed. It is interesting to note that the "knee" of the graph for the 32-PE case occurs later than for fewer PEs. This is perhaps because the system is not saturated with threads up to that point, and more are therefore executed (and so more details recorded in the log file). The log file varies in size between 1Kbytes (for the 1PE cases) to 6Mbytes (for 32 PEs running a 250x250 ray tracer).

The log file sizes for GranSim-CC in contrast are similar in size no matter how many PEs are used. This is because every cost centre change is recorded in the log file, and these will occur for every function executed in both sequential and parallel code regardless of the number of threads that are run. The actual log file sizes are very large (ranging from 80MBytes to 2GB), thus imposing a serious practical limit on the use of unrestricted cost centre profiling, and presumably having a significant impact on time. The approaches that are aimed at reducing the time cost of GranSim-CC should thus also be effective in reducing the log file size.

5.4. Availability

The HasPar suite is available from `http://www-fp.dcs.st-and.ac.uk/haspar/`. The majority of the tools are available on three widely used architectures and platforms: Compaq Alphas running Digital Unix, Sun Sparcs running Solaris and Intel Pentiums running Linux. In our experience, it is straightforward to port the simulation (GranSim) tools, and there should be little difficulty in providing versions for the other systems currently supported by GHC, which include the MIPS and HP-PA architectures as well as FreeBSD and CygWin operating system support for Intel

Pentiums. Porting the actual GUM system can involve slightly more effort, but the fact that it can be built using either PVM or MPI simplifies this task considerably.

6. Related Work

This section considers systems and techniques that are related to those used in the HasPar suite. Sections 6.1 and 6.2 cover simulation and analytical modelling approaches to performance prediction, respectively. Section 6.3 covers performance monitoring for parallel functional programs, while Section 6.4 covers relevant sequential performance monitoring techniques, including alternatives to the cost centre based approach we have used here. Finally, Section 6.5 covers general tools for performance evaluation including a discussion of visualisation techniques that have been used in other settings.

6.1. Performance Modelling: Simulation of Parallel Functional Languages

Many parallel simulation tools have been produced, covering a range of detail from highly abstract (as with GranSim-Light) to complete emulations of some proposed or actual design. Generally the focus of these tools has been on evaluating system characteristics rather than directly assisting the applications programmer. Such system simulations include the ZAPP simulation of a local work offloading scheme [6], or Bennett's cache coherency simulator [5].

Consequently, the emphasis of simulation tools in this area has generally been on programming "in-the-small", providing precise information about detailed and often low-level events. In contrast, we have focused on programming "in-the-large", producing high-level information that can be easily interpreted by the programmer, and used to improve the program.

There are a number of other simulators that can be seen as being aimed at providing user-level information. For example, Roe's simulator [56] for the non-strict language LML resembled GranSim-Light in producing idealised simulations that were aimed at understanding the general characteristics of a parallel program. Each basic block was costed as a single unit cost, an infinite number of processors were modelled, with communication being instantaneous. A round-robin thread scheduler was used, with an eager thread creation model.

The very similar `hbcpp` simulator [57] (integrated with the Chalmers' Haskell compiler) was widely used in the FLARE project [58] to yield information about potential parallel performance, producing overall activity profiles that inspired the ones shown here. `hbcpp` simulates either a finite or an infinite number of processors, using an eager task creation model. Like GranSim-Light, `hbcpp` could be used to test theories about parallel algorithms, or to eliminate serious performance bottlenecks. Because they are aimed specifically at idealised simulation, both Roe's simulator and `hbcpp` were extremely fast, unlike GranSim-Light, which builds on the more complicated internal cost model of the GranSim simulator.

Charles and Runciman have also produced per-thread profiles for GranSim [7]. Unlike the profiles we describe here, these profiles also include the sources of blocking (this requires a small extension to GranSim). Provided there is only limited communication during a parallel execution, this could prove to be a useful performance tool. An interesting feature of this work is that the logs produced by each simulation run are stored in a database. This database can be queried repeatedly in order to create alternative profiles of the same run, with the content of the profile under programmer control. This flexibility contrasts with the GranSim approach where the simulation run must be repeated and new logs created in order to produce different profiles from a fixed set of possibilities. Comparing the approach taken by Charles and Runciman with our own approach, there is a trade between the cost of constructing the database (in both time and space), and the ability to create new profiles (whether in a standard format or problem-specific) from a single simulation run. Performance results suggest that this point is reached if 3 or more profiles are created from a single simulation run.

Finally, another closely related system is the Paradise extension of GranSim to the parallel Eden language [25]. This supports explicit process abstractions, with communication via explicit input and output ports. The performance monitoring tools that are provided are based on those described here, but also include visualisations relating cost to Eden processes. In due course, we hope to produce a combined version of GranSim supporting both Eden and GpH.

6.2. Performance Modelling: Analytical Cost Modelling

Cost modelling has recently received much attention in certain sections of the parallel programming community [65, 39]. Cost models are used to construct a formal abstract representation of a parallel program that can then be used to reason about parallel performance. Typically a cost model will yield a cost equation that is parameterised on details of the communications topology etc. Cost models differ from the simulation models described here in two basic ways: firstly, the models are formal rather than informal; secondly, the models are more abstract than those we have used here.

The purpose of a cost model is not to give a precise costing to a specific parallel program execution, but rather to provide more abstract behavioural information. For example, cost models can be used to answer questions about communications or processor topologies. On a more general level, they can also be used to construct an abstract model of a program's structure, allowing questions to be answered about parallel program design such as choise of algorithm or data structure. Of course, both these types of question can also be answered by simulators such as our GranSim simulator plus suitable prototyping to assess design issues. The fundamental difference is that whereas GranSim will provide answers in terms of machine cycles, a cost model will provide answers in terms of abstract cost, perhaps parameterised in terms of input size, number of processors etc.

6.3. Performance Monitoring: Parallel Functional Programs

The majority of previous work on parallel non-strict functional languages has focused on coarse-grained metrics of parallel performance such as elapsed time. There has thus been relatively little work specifically aimed at performance monitoring tools. Historically, on the GRIP system [49] that preceded the development of GUM, dynamically allocated *activity centres* were introduced at runtime. All subsequent execution costs could then be attributed to this activity centre [22], up to the creation of the next activity centre. This provided a crude way of "marking" dynamic points in a program's execution, similar to the technique described here for strategic profiling.

Similar activity profiling was also used by the WYBERT implementation on a shared-memory machine. This system classified time by normal execution, time used by the runtime system, time spent in garbage collection and idle time. Both GRIP and WYBERT reported overall activity results as a set of profiles, one for each type of activity. Both systems also reported results on either an overall or per-PE basis. In a strict setting, the MultiLisp parallel Lisp implementation [15] includes profiling information that can be visualised by the MulTVision system [17].

6.4. Performance Monitoring: Sequential Functional Languages

6.4.1. Sequential Time Profiling

Long dismissed as an overly difficult problem, execution profiling of sequential lazy functional programs has recently received some overdue attention. In addition to the cost centre profiling mechanism exploited by GranSim-CC [62, 55], Clack et al. contemporaneously investigated a so-called *lexical profiling* technique [8].

Lexical profiling attributes computation costs to those functions or expressions that gave rise to them, a simple intuitive concept. *Cost centre* profiling [62], in contrast, attributes the costs to those *cost centres* that most closely enclose the computation. This allows the flexibility of attaching user-defined cost centres to programs. In its most commonly used form, however, where cost centres are automatically attached to functions and standalone cost centres are rarely used, cost-centre profiling is essentially equivalent to lexical profiling. One major difference is in the attribution of costs to global constants (CAFs): in the lexical profiling scheme, these are attributed to the top-level function or expression to which they belong; in the cost centre profiling case, as we have seen, they are attributed to a per-module CAF cost centre. The latter is usually not particularly informative, and can disguise the true sources of computation cost.

Time profiling has also received attention for strict functional languages. Standard ML of New Jersey provides a profiler based on *inheritance profiling*, in which the costs of evaluating child functions are attributed to their parents [2]. In contrast with the cost centre approach described above, a limitation of inheritance profiling is that the costs of evaluating the arguments to a higher-order function are not usefully attributed.

6.4.2. Sequential Heap Profiling

Recent attention has also focused on the issue of costing heap allocation in sequential functional programs. Both cost centre and lexical profiling techniques can be equally applied to both time and space profiles, and there has been independent interest in dynamic profiling to determine heap node lifetime and liveness properties in order to eliminate “space leaks” and other undesirable space usage behaviour [60, 59]. To our knowledge there has been no attempt to profile parallel heaps in a correspondingly detailed fashion, and the coarse statistics produced by GranSim and GUM are some of the few available for parallel allocation. This area would repay further investigation.

6.5. General Performance Evaluation Tools

In contrast with the situation pertaining to tools that are specific to functional languages, many performance evaluation tools have been developed for more conventional language systems. This section surveys some of the more interesting tools, highlighting features that might usefully be exploited in future versions of the HasPar suite.

6.5.1. Relative Traces

DiP [32] is an integrated performance suite providing an instrumented communications library, a distributed-memory parallel machine simulator, plus visualisation and analysis tools. In contrast to GranSim traces, which are tied to a particular execution cost model, the DiP system [32] generates traces which contain relative rather than absolute times. By applying appropriate execution and communication cost models, the trace results can be transformed to provide information about different architectures. In effect the simulation is partially applied to some application, leaving the cost model to be supplied later as a new parameter. The accuracy of prediction is claimed to be good for regular applications.

One interesting application of relative traces is to allow the use of a *synthetic perturbation algorithm*. This algorithm computes new traces by applying systematic changes to the original. These changes can be used to show the effect of optimising a single function by a certain amount, or to show the effect of different communications latencies etc.

While relative traces are an attractive prospect, since they eliminate the expense of re-creating trace files for different architectures, it is not obvious that the DiP model can be easily adapted to the dynamic setting we require: in our setting, scheduling, thread creation, migration and even communication decisions are inter-dependent. In order to recreate these inter-dependencies, much of the thread level simulation which is currently performed by the GranSim simulation would need to be duplicated by the post-facto trace generator. This would introduce a consistency problem that does not exist at present: because GranSim executes exactly the same code as the real GUM system, we can have a high degree of confidence in the results that we obtain.

DiP profiles are visualised using a single Gantt chart format, whose lines represent either applications, processors, tasks, or threads depending on the level of detail chosen, and which can be coloured according to the state of the object. Filter modules are used to filter trace information so supporting user selection of interesting profiling information.

6.5.2. Detecting Bottlenecks

An underlying principle of high quality performance evaluation tools is that they should focus attention on those parts of a program that will repay optimisation work. It is also highly desirable to divert attention from fruitless areas.

A standard technique that is often supported is critical path analysis to determine the most important dynamic program dependencies [29]. For example, the ParaGraph [24] tool discussed below provides a post-mortem critical path analysis as one of its visualisations.

When used in conjunction with a time-based activity profile that has been related to the source as with GranSim-CC or GranSim-SP, critical path analysis can help isolate the causes of sequential or low-activity bottlenecks by indicating the threads or communications that cause serialisation [44]. In the GranSim setting, where threads

Prepared using speauth.cls

are low-level objects over which the programmer has no direct control, it appears to be much more difficult to make use of such an analysis. Critical path analysis may also produce large trace files since all communications must be traced.

An alternative approach is taken by the Quartz performance tuning tool [1]. The principal metric used by Quartz is *normalised processor time*. This metric takes into account general parallel activity whilst a section of code is being executed. Sections with high normalised processor times are candidates for optimisation, since they are either long-running or highly sequential. Procedures, threads and synchronisation objects are all measured in the same way.

The Paradyn parallel performance measurement tool [43] adopts a more automated approach to detecting performance bottlenecks. The system automatically monitors the program, and inserts instrumentation dynamically. This allows instrumentation precisely when and where required, so reducing the costs of excessive monitoring that can be introduced in systems such as ours.

An especially interesting aspect of the Paradyn system is that it is capable of automatically constructing hypotheses concerning performance bottlenecks that can then be tested automatically, and perhaps refined into more precise hypotheses. The W^3 search model attempts to answer the three crucial questions of *Why* the application is running slowly, *Where* is the bottleneck, and *When* does it manifest itself. At present, these questions are applied to low-level system metrics, but it would be interesting to investigate the application of the same ideas to the higher-level metrics which form the focus of the HasPar suite. In such a system, parallel cost-centre profiling and dynamic strategic profiling would be crucial to allowing the construction of high-level hypotheses.

6.5.3. Intrusion Compensation

A major problem that is faced by many performance evaluation tools is that the profiling process itself can seriously perturb the timing information that is gathered. A major objective of the AIMS system [70] is to eliminate such systematic timing errors for parallel and distributed programs. It does this by using an *intrusion compensation* module that subtracts the cost of profiling from the current clock time when dumping event records.

Although the AIMS system attempts to relate information to the program source, the animated views do not provide sufficient abstraction for use on programs with large numbers of tasks. For such programs, the system provides raw statistical data that can be plotted with a conventional spreadsheet package. To improve portability, the AIMS system uses source-level code instrumentation as in the PICL [24] or Pablo [54] libraries. As with the HasPar suite, instrumentation is inserted automatically by the compiler.

6.5.4. Trace-Driven Profiling Tools and Visualisations

The trace-driven profiling strategy we have adopted is widely used. Two well-known examples are XPVM [13] and ParaGraph [24]. XPVM is a graphical monitor for the PVM message-passing system [13]. The system supports graphical display of task level information, with playback and replay facilities. Utilisation and communications information is provided at the PVM task level. PVM libraries are instrumented to automatically record communications activity without modifying the source code. Since several versions of the GUM system use the PVM communications library, it follows that we can already directly exploit XPVM for fine systems tuning, or to obtain information about general communications behaviour. Since a single PVM task encompasses many GpH threads, however, and since it is not possible to relate profiling information to the GpH source, the system is not suitable for applications level performance monitoring.

ParaGraph [24] is an integrated post-mortem trace inspection tool with a graphical user interface. It provides a number of different profiling capabilities, including critical path analyses, gross activity profiles, communication patterns etc. While the system provides a great deal of data concerning low-level system events, it is not obvious that this data could be usefully related to functional language sources. For system tuning purposes, however, the information provided could be used in the same way as that provided by XPVM, with the advantage that GranSim performance could also be monitored. To achieve this, it would be necessary to modify the HasPar trace output so that it could be used as input to ParaGraph. We have not yet investigated the detailed practicality of such a proposal.

Vista is an ambitious visualisation system that aims to visualise many details of parallel symbolic program performance. A key principle underlying the design of Vista is that it should be extremely flexible: it supports a number of different data views; relationships can be expressed between different data; it aims to include source-related information; and execution traces are specified using a self-describing data format, SDF [16], that can be easily adapted to new settings. The system is interactive and supports both static and dynamic tracing. It thus has many features in common with the tool proposed by Charles and Runciman [7], and could perhaps serve as an alternative visualiser for GUM and GranSim programs.

7. Conclusions and Further Work

This paper has introduced the HasPar performance evaluation suite for the parallel non-strict functional language GpH, a small extension of Haskell. Performance evaluation is a topic of some interest at present, with special emphasis being placed on tools that are both easy to use and easy to understand [46]. In the design of the HasPar suite, we have focused on the latter aspect, whilst also considering the former to some extent.

Providing appropriate performance information in a parallel non-strict functional language such as GpH is especially difficult because such languages deliberately abstract as much as possible of the underlying control structure. This is further exaggerated by our evaluation strategy approach [68], where we aim to separate the description of the program from its parallel behaviour. The HasPar suite has overcome these difficulties by abstracting performance data to a very high level. Information is related to the sources using two novel mechanisms: parallel cost centres for static, *lexically scoped* profiling ("algorithmic" information) and parallel strategy marking for dynamic, *evaluation scoped* profiling ("behavioural" information). To assist the programmer and researcher into parallelism, we have provided a number of abstract visualisation tools. These tools provide information at a number of levels of detail, including overall, per-processor, and per-thread activity profiles.

The paper has used a number of small sample programs to illustrate the differences between the monitoring tools, and to show how each can be used to reveal particular kinds of information. An important facet of our work is that we have not restricted our attention to such demonstrator applications, however, but have also considered much larger parallel programs. The most notable of these is a 6,000 line functional language compiler [28], which was successfully parallelised on a network of workstations. The 47,000 line Lolita natural language processor [37] also yielded useful information on the parallelisation process and indicated that small-scale speedups should be achievable with minimal modification to the program, but the constraints of physical shared-memory on the target architecture mean that we have still to achieve acceptable speedup for this application.

7.1. Further Work

There are a number of directions in which we wish to develop the HasPar suite. These can be divided into work on: designing new tools; closing the gap with analytical modelling; and improving the coverage of the underlying evaluation strategy model. We will now consider each of these directions in turn.

7.1.1. New Tools and Log Formats

Although we have found the tools in the HasPar suite to be effective, there are a number of additional tools that could prove useful if they were incorporated into the suite. We intend to work on a number of new performance monitoring tools in the near future, including:

- Critical task analysis. This tool would identify the critical task (the longest-running sequential task) at each point in the execution, as for the ParaGraph tool described in Section 6.5.2. We have already constructed a prototype of such a tool.
- Locality profiling. Clustered architectures are becoming more widespread. In order to investigate the applicability of our model to such architectures, it will be necessary to extend the tools with information about both task and data locality.
- Cost centre stacks. Jarvis has introduced the notion of cost centre stacks [55] that allow cost centres to be extended with the dynamic notion of a call graph, so allowing different calls to a cost centre to be distinguished in the profile in a similar way to GranSim-SP. It should prove straightforward to incorporate this feature into the next version of the HasPar suite.

We currently use our own format for the log files. A number of benefits would accrue from using more widely used formats.

- **PICL**: Adding support for PICL [24] calls would allow us to access low-level profiling tools providing general information about communication patterns etc. This would primarily be useful for systems-level programming, but might also provide some user-level information.
- **XML**: Jarvis et al have proposed a general self-describing format for profiling-log data using XML [27]. This would allow us to access visualisation tools written by other groups, such as those for BSPlib, and would allow others to use our visualisation tools for their own implementations.

Finally, the visualisation tools described here are essentially batch tools. There is much current interest in the issue of interactive performance evaluation tools [46], and it is clear that these can provide useful rapid feedback. While the focus of this paper has been on mechanisms to support the development of parallel application through providing appropriate performance information, we have also developed interactive environments that allow the user to focus attention on an interesting part of the execution or to otherwise direct the performance evaluation process. We intend to incorporate these tools into the full suite in the near future.

7.1.2. *Analytical Modelling*

Although this paper has focused on measurement and simulation, we regard analytical modelling as an important component of the parallel programming process. There are a number of ways in which our support for modelling could be improved. For example, we have developed an operational semantics for our basic `par` and `seq` constructs [14]. This allows us to prove equivalences between programs, and to understand some aspects of the parallel behaviour. We are in the process of extending this semantics to cover more issues, such as placement and communication [4]. Relating this semantics to the performance monitoring tools described here would allow us to validate the model against the simulation and actual implementation.

7.1.3. *New Applications*

Finally, we are investigating a number of new large-scale irregular applications, with a view to improving the coverage of our evaluation strategy approach and the associated tools. The most significant of these applications are the radiosity method for global scene illumination, AI search techniques, and matching genetic sequences using complex criteria.

ACKNOWLEDGEMENTS

This work was generously supported by research grants GR/M 32351, GR/M 55633, GR/M 93379 from the UK's Engineering and Physical Science Research Council (EPSRC) and by British Council/DAAD Research Grant no. 1097. We are grateful for the helpful comments provided by Greg Michaelson, Chris Clack and Aled Sage on earlier versions of this paper.

REFERENCES

1. T.E. Anderson and E.D. Lazowska. Quartz: a Tool for Tuning Parallel Program Performance. In *Proc. ACM SigMetrics Conf. on Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.
2. A.W. Appel, B.F. Duba, and D.B. MacQueen. Profiling in the Presence of Optimisation and Garbage Collection. part of the Standard ML of New Jersey compiler distribution, 1988.
3. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A Structured High Level Programming Language and its Structured Support. *Concurrency — Practice and Experience*, 7(3):225–255, May 1995.

4. C. Baker-Finch. Parallel Lazy Abstract Machines. In G.A. Michaelson and P.W. Trinder, editors, *Proc. 1st. Scottish Functional Programming Workshop, August 1999*, Stirling, August 2000. Heriot-Watt University. To appear.
5. A.J. Bennett and P.H.J. Kelly. Locality and False Sharing in Coherent-Cache Parallel Graph Reduction. In *Proc. PARLE '93 – 1993 Conf. on Paralel Architectures and Reduction Languages Europe*, pages 329–340, Munich, Germany, June 1993.
6. F.W. Burton and M.R. Sleep. Executing Functional Programs on a Virtual Tree of Processors. In *Proc. 1st. Conf. on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, 1981.
7. N. Charles and C. Runciman. Chapter 13: Performance Monitoring. In Kevin Hammond and Greg A. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 233–246, September 1999.
8. C. Clack, S. Clayman, and D. Parrott. Lexical profiling: Theory and practice. *Journal of Functional Programming*, 5(2):225–277, April 1995.
9. K.J. Danhof and M.R. Zargham. A Petri Net Model of Parallel Processing: Performance Analysis and Program Verification. *J. Combined Information System Science (LNCS)*, 14(2):73–94, 1989.
10. T. Fahringer. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Press, 1996.
11. M. Fernández, M. Carro, and M. Hermenegildo. IDRA (Ideal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proc. EuroPar '96*, volume 1124 of *Lecture Notes in Computer Science (LNCS)*, pages 724–734. Springer-Verlag, August 1996.
12. G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Menchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing.* MIT press, 1994.
13. G.A. Geist, J. Kohl, , and P. Papadopoulos. Visualising, Debugging and Performance in PVM. In *Proc. Visualisation and Debugging Workshop*, October 1994.
14. J.G. Hall, C. Baker-Finch, P.W. Trinder, and D.J. King. Towards an Operational Semantics for a Parallel Non-Strict Functional Language. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *Proc. IFL '98 — 1998 Intl. Workshop on Implementation of Functional Languages, London, UK*, volume 1595 of *Lecture Notes in Computer Science (LNCS)*, pages 55–72. Springer-Verlag, September 1998.
15. R.H. Halstead Jr. An Assessment of MultiLisp: Lessons from Experience. *International Journal of Parallel Processing*, 15(6), December 1986.
16. R.H. Halstead Jr. Self Describing Files + Smart Modules = Parallel Program Visualisation. In *Theory and Practice of Parallel Programming*, number 907 in Lecture Notes in Computer Science (LNCS), pages 253–283. Springer-Verlag, November 1994.
17. R.H. Halstead Jr. and D.A. Kranz. A Replay Mechanism for Mostly Functional Parallel Programs. Technical Report CRL 90-6, Digital Cambridge Research Labs., November 1990.
18. K. Hammond. Evaluation Strategies for Irregular Parallel Programming. In *Proc. PDSCIA '99 — 1999 Intl. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Computations (PDSCIA '99), Sendai, Japan*, Sendai, Japan, July 2000. World Scientific. To appear.
19. K. Hammond, H.-W. Loidl, and A.S. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *Proc. HPFC '95 — 1995 Conf. on High Performance Functional Computing, Denver, CO*, pages 208–221, April 1995.
20. K. Hammond, H.-W. Loidl, and P.W. Trinder. Parallel Cost Centre Profiling. In *Proc. 1997 Glasgow Workshop on Functional Programming, Ullapool, Scotland*, pages 51–72, September 1997.
21. K. Hammond, S. Marlow, and S.L. Peyton Jones. A New Shared Memory Implementation of Haskell. In preparation, November 1999.
22. K. Hammond and S.L. Peyton Jones. Some Early Experiments on the GRIP Parallel Reducer. In *Second Intl. Workshop on the Parallel Implementation of Functional Languages, Univ. of Nijmegen, The Netherlands*, pages 51–72, June 1990.
23. K. Hammond and A. Rebón Portillo. HaskSkel: Implementing Algorithmic Skeletons in a Non-Strict Functional Language. In *Proc. IFL '99 — 1999 Workshop on Implementation of Functional Languages, Lochem, The Netherlands*, Lecture Notes in Computer Science (LNCS). Springer-Verlag, 2000. To appear.
24. M. Heath and J. Etheridge. Visualising the Performance of Parallel Programs. *IEEE Software*,

pages 29–39, September 1991.

25. F. Hernández, R. Peña, and F. Rubio. From GranSim to Paradise. In G.A. Michaelson and P.W. Trinder, editors, *Proc. 1st. Scottish Functional Programming Workshop, August 1999*, Stirling, August 2000. Heriot-Watt University. To appear.
26. Report on the Programming Language Haskell Version 1.2. *ACM SIGPLAN Notices*, 27(5), May 1992.
27. S.A. Jarvis, S. Marlow, and S. Peyton Jones a E. Wilcox. Standardising Compiler/Profiler Log Files. In *Draft Proc. IFL '98 – 1998 Intl. Workshop on the Implementation of Functional Languages, London, UK*, 1998.
28. S.H. Junaidu, A.J.T. Davie, and K. Hammond. Naira: A Parallel^2 Haskell Compiler. In *Proc. IFL '97 – Intl. Workshop on the Implementation of Functional Languages, St Andrews, UK*, number 1467 in Lecture Notes in Computer Science (LNCS), pages 215–231, St Andrews, Scotland, September 1997. Springer-Verlag.
29. T. Kerola and H. Schwetman. Monit: a Performance Monitoring tool for Parallel and Pseudo-Parallel Programs. In *Proc. ACM SigMetrics Conf. on Measurement and Modeling of Computer Systems*, May 1987.
30. D.J. King, J.G. Hall, and P.W. Trinder. A Strategic Profiler for Glasgow Parallel Haskell. In K. Hammond, A.J.T. Davie, and C. Clack, editors, *Proc. IFL '98 — 1998 Intl. Workshop on Implementation of Functional Languages, London, UK*, volume 1595 of *Lecture Notes in Computer Science (LNCS)*, pages 90–104. Springer-Verlag, September 1998.
31. K. Kubota, K. Itakura, M. Sato, and T. Boku. Practical Simulation of Large-Scale Parallel Programs and Its Performance Analysis of the NAS Parallel Benchmarks. In *Proc. EuroPar '98*, volume 1470 of *Lecture Notes in Computer Science (LNCS)*, pages 244–254. Springer-Verlag, September 1998.
32. J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. DiP: A Parallel Program Development Environment. In *Proc. EuroPar '96, Lyon, France*, number 1124 in Lecture Notes in Computer Science (LNCS). Spinger-Verlag, August 1996.
33. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):559–565, July 1978.
34. J. Launchbury. A Natural Semantics for Lazy Evaluation. In *Proc. POPL '93 — 20th ACM Symposium on Principles of Programming Languages, San Francisco, Carlifornia*, pages 144–154, Charleston,Texas, 1993.
35. H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, 1998.
36. H.-W. Loidl and K. Hammond. Making a Packet: Cost-Effective Communication for a Parallel Graph Reducer. In *Proc. IFL '96 – 1996 Intl. Workshop on the Implementation of Functional Languages, Bonn/Bad-Godesberg, Germany*, pages 184–199, September 1996.
37. H.-W. Loidl, R. Morgan, P.W. Trinder, S. Poria, C. Cooper, S.L. Peyton Jones, and R. Garigliano. Parallelising a Large Functional Program; Or: Keeping LOLITA Busy. In *Proc. IFL'97 — 1997 International Workshop on the Implementation of Functional Languages, St Andrews, UK*, volume 1467 of *Lecture Notes in Computer Science (LNCS)*, pages 199–214. Springer-Verlag, September 1998.
38. H.-W. Loidl and P.W. Trinder. Engineering Large Parallel Programs. In *Proc. IFL '97 – 1997 Intl. Workshop on the Implementation of Functional Languages, St Andrews, UK*, number 1467 in Lecture Notes in Computer Science (LNCS), pages 178–197. Springer-Verlag, September 1997.
39. W. F. McColl. Scalable Computing. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science (LNCS)*, pages 46–61. Springer-Verlag, 1995.
40. Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. Supercomputing '93*, pages 878–883. IEEE Computer Society Press, 1993.
41. G.A. Michaelson and K. Hammond. Chapter 2: Fundamentals. In K. Hammond and G.A. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 31–61, September 1999.
42. G.A. Michaelson, A. Ireland, and P.J.B. King. Towards a Skeleton Based Parallelising Compiler for SML. In C. Clack, K. Hammond, and A.J.T. Davie, editors, *Proc. IFL '97 — 1997 International Workshop on the Implementation of Functional Languages, St Andrews, UK*, volume 1467 of *Lecture Notes in Computer Science (LNCS)*, pages 539–546. Springer-Verlag, 1997.

43. B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R. B. Irvin, K.L. Karavanic, K. Kunchithapadam, and Tia Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11), November 1995.
44. B.P. Miller and C.-Q. Yang. IPS: an Interactive and Automated Performance Measuring Tool for Parallel and Distributed Programs. In *Proc. 7th. International Conf. on Distributed Computing Systems*, September 1987.
45. E. Mohr, D.A. Kranz, and R.H. Halstead. Lazy Task Creation – a Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.
46. C.M. Pancake. Human Factors Issues for Parallel Performance Tools. In *Proc. Euro-Par '99*, number 1685 in Lecture Notes in Computer Science (LNCS), pages 44–60. Springer-Verlag, 1999.
47. D.A. Patterson and J.L. Hennessey. *Computer Organisation & Design: The Hardware/Software Interface*. Morgan Kaufmann, 1998.
48. J.C. Peterson and K. Hammond (eds.). Report on the Programming Language Haskell Version 1.4. Technical report, Yale University, 1997.
49. S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – a High-Performance Architecture for Parallel Graph Reduction. In *Proc. FPCA '87 – 1987 Conf. on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science (LNCS), pages 98–112. Springer-Verlag, 1987.
50. Simon L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proc. FPCA '89 – 1989 ACM Conf. on Functional Programming Languages and Computer Architecture, London, UK*, pages 184–201, New York, NY, 1989. ACM.
51. S.L. Peyton Jones and R.J.M. Hughes (eds.). Haskell 98: A Non-strict, Purely Functional Language. Technical report, Yale University, 1999.
52. B. Plateau, J.M. Forneau, and K.H. Lee. PEPS: A Package for Solving Complex Markov Models of Parallel Systems. In R. Puigjaner, editor, *Proc. 4th Intl. Conf. on Modelling Tools and Techniques for Comp. Perf. Eval.* Plenum Publishers, 1988.
53. M.J. Quinn. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, 1987.
54. D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera. Scalable Performance Analysis: the Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conference*. IEEE Computer Society, 1993.
55. R.G.Morgan and S.A.Jarvis. Profiling Large-Scale Lazy Functional Programs. *Journal of Functional Programming*, 8(3):201–237, May 1998.
56. P. Roe. *Parallel Programming using Functional Languages*. PhD thesis, Dept. of Computing Science, Univ. of Glasgow, February 1991.
57. C. Runciman and D. Wakeling. Profiling Parallel Functional Computations (Without Parallel Machines. In *Proc. 1993 Glasgow Workshop on Functional Programming, Ayr, Scotland*, WICS, pages 236–251. Springer-Verlag, July 5–7 1993.
58. C. Runciman and D. Wakeling, editors. *Applications of Functional Programming*. UCL Press, 1995.
59. Colin Runciman and Niklas Röjemo. Two-Pass Heap Profiling: a Matter of Life and Death. In *Proc. IFL '96 – Intl. Workshop on the Implementation of Functional Languages, Bonn/Bad-Godesberg*, number 1268 in Lecture Notes in Computer Science (LNCS), pages 222–232. Springer-Verlag, 1996.
60. Colin Runciman and David Wakeling. Heap profiling of Lazy Functional Programs. *Journal of Functional Programming*, 3(2):217–245, April 1993.
61. P.M. Sansom and S.L. Peyton Jones. Time and Space Profiling for Non-strict Higher Order Functional Languages. In *Proc. POPL '95 — 22nd ACM Symposium on Principles of Programming Languages, San Francisco, Carlifornia*, January 1995.
62. P.M. Sansom and S.L. Peyton Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.
63. V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. MIT Press, 1989.
64. Scientific and Engineering Software Inc. SES/workbench Reference Manual, Release 3.1. 1996.
65. David B. Skillicorn. Chapter 8: Cost Modelling. In Kevin Hammond and Greg A. Michaelson, editors, *Research Directions in Parallel Functional Programming*, pages 207–218, September 1999.
66. A.B. Tayyab. *Performance Evaluation for a Class of Parallel Computations*. PhD thesis, Department of Electrical and Computer Engineering, University of Iowa, 1993.

67. P.W. Trinder, E.W. Barry, K. Hammond, S.B. Junaidu, H.-W. Loidl, and S.L. Peyton Jones. Engineering Large Scale Parallel Programs. *Concurrency: Practice & Experience*, 2000. To appear.
68. P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
69. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S. Partridge, and S.L. Peyton Jones. GUM: A Portable Parallel Implementation of Haskell. In *Proc. PLDI '96 — 1996 ACM Conf. on Programming Language Design and Implementation, Philadelphia, PA*, pages 78–88, May 1996.
70. J.C. Yan, S. Sarukkai, and P. Mehra. Performance Measurement, Visualisation and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software — Practice and Experience*, 25(4):429–461, April 1995.

Prepared using speauth.cls