A Strategic Profiler for Glasgow Parallel Haskell

David J. King[1], Jon Hall[1], and Phil Trinder[2]

[1] Faculty of Maths and Computing, The Open University
Walton Hall, Milton Keynes MK7 6AA
{d.j.king, j.g.hall}@open.ac.uk
http://mcs.open.ac.uk/{djk26, jgh23}
[2] Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS
trinder@cee.hw.ac.uk, http://www.cee.hw.ac.uk/~trinder/

Abstract. Execution profiling plays a crucial part in the performance-improving process for parallel functional programs. This paper presents the design, implementation, and use of a new execution time profiler (GranSim-SP) for Glasgow Parallel Haskell (GpH). Parallelism is introduced in GpH by using *evaluation strategies*, which provide a clean way of co-ordinating parallel threads without destroying a program's original structure. The new profiler attributes the cost of evaluating parallel threads to the strategies that created them. A unique feature of the strategic profiler is that the call-graph of evaluation strategies is maintained, allowing the programmer to discover the sequence of (perhaps nested) strategies that were used to create any given thread. The strategic profiler is demonstrated on several examples, and compared with other profilers.

1 Introduction

Profiling tools that measure and display the execution behaviour of a program are well established as one of the best ways of tracking down code inefficiencies. Profiling tools are particularly useful for very high-level languages where the relationship between source program and execution behaviour is far from apparent to a programmer, compared to conventional languages like C, for instance. Moreover, parallelism and laziness in a language magnifies the disparity between source program and execution behaviour by the sheer number of activities carried out in an chaotic order.

With all this in mind, a new profiler has been developed (which we're calling GranSim-SP) for the non-strict parallel functional language Glasgow Parallel Haskell (GpH)—an established parallel derivative of Haskell. The profiler allows for a direct correspondence between source program and a visualisation of its execution behaviour. This is done by allowing programmers to augment the parallel co-ordination parts of their code (known as *evaluation strategies*) with labels. Each label corresponds to a unique colour in a graphical visualisation of a program's execution behaviour, thus allowing the threads created by different strategies to be distinguished.

This paper has the following structure. In Section 2 GPH, and evaluation strategies are briefly explained. Then in Section 3 a description is given of the GPH runtime system for GRANSIM that GRANSIM-SP is built on top of. In Section 4, the new combinator for annotating strategies is given, and a direct comparison is made with the GRANSIM profile in the previous Section. In Section 5 the implementation of GRANSIM-SP is described, and in Section 6 motivating examples are given justifying why GRANSIM-SP is useful. In Section 7 GRANSIM-SP is compared and contrasted with GRANSIM-CC another profiler for GPH, and some imperative profilers are described. Finally, Section 8 discusses the paper and gives future work.

2 Glasgow Parallel Haskell

Glasgow Parallel Haskell provides parallelism by adding to Haskell a combinator for parallel composition, `par`, that is used together with sequential composition, `seq`, to control parallelism. The denotational behaviour of these combinators is described by the following equations:

$$\texttt{x `seq` y} = \begin{cases} \perp, & \text{if x is } \perp \\ \texttt{y}, & \text{otherwise} \end{cases} \qquad\qquad \texttt{x `par` y = y}$$

Operationally, the combinator `seq` arranges for its arguments to be evaluated in sequence, `x` then `y`, and `par` arranges for its arguments to be evaluated in parallel, `x` on a separate processor from `y`. These combinators are explained in more detail elsewhere, for example, see Loidl [8].

2.1 Evaluation Strategies

Adding `par` and `seq` combinators to programs can easily obfuscate the code—mixing computation with parallel co-ordination. In view of this, Trinder et al. [15] introduced *evaluation strategies* which are lazy higher-order polymorphic functions that allow expressions to remain intact, but with suffixes that suggest their parallel behaviour. For example, an expression that maps a function over a list may be parallelised in the following way:

```
map f xs `using` strategy
```

The function `strategy` describes how to parallelise `map f xs`, and `using` is a composition function with the following definition:

```
using :: a -> Strategy a -> a
e `using` s = s e `seq` e
```

Evaluation strategies can be user-defined, but a lot of commonly occurring strategies are provided in a library module bundled with GPH.

2.2 The need for GranSim-SP

When parallelising a program there are a number of different choices that can be made. Evaluation strategies make it easier to focus on the parallel co-ordination routines, but a programmer still has to choose which expressions to parallelise, and which strategies to use. The purpose of GranSim-SP is to give direct feedback to the programmer on the effectiveness of different choices in strategies.

3 The runtime system for GpH

The compiler for GpH that GranSim-SP has been built on top of is an extension of the Glasgow Haskell compiler, GHC (Peyton Jones et al. [11]). The compiler supports parallelism using the GUM runtime system (Trinder et al. [16]) and a parallel simulator called GranSim (Loidl [7, 8]). Our strategic profiler GranSim-SP is an extension of GranSim, and hence its name.

The runtime system that provides parallelism and parallel simulation is based on the sequential threaded runtime system for GHC that was developed to provide concurrency (Peyton Jones et al. [10]). Threads are best thought of as virtual processors. They are represented in the runtime system by TSOs (Thread State Objects) which hold a thread's registers together with various statistics about a thread. Threads are created by the `par` combinator, and die when they have completed their work. At any instant a thread will be in one of five states: running, runnable (awaiting resources to become available), blocked (awaiting for the result of evaluating another thread), fetching (awaiting the arrival of a result from another processor), or migrating (moving from a busy processor to an idle processor).

3.1 Recording events

The parallel simulator GranSim (Loidl [7, 8]) records information in a log file about each thread when they change from one state to another (see Figure 1).

```
PE 18 [612386]: SCHEDULE        39
PE 17 [612288]: BLOCK           15
PE 17 [612398]: REPLY           1f
PE 17 [612410]: START           88
PE 18 [614185]: FETCH           39
PE 18 [614335]: SCHEDULE        16
PE  1 [614172]: FETCH           6a
PE  1 [614322]: SCHEDULE        68
PE 17 [614254]: RESUME(Q)       15
PE 20 [614481]: END             3c
```

Fig. 1. An extract from a GranSim log file.

Each line of the log file contains the processor number, the time (given in number of instructions), the state that a thread is changing to, followed by the thread number. Some other statistics about threads are also recorded, but they have been omitted here since they are of less interest to us.

3.2 Graphical visualisations of log files

Log files of programs can contain so much data that it is almost impossible to discover useful information from them without a tool; such as one that can give a graphical representation of the data. GranSim provides tools for doing this, the most useful of which shows how many threads there are, and what state they are in (running, runnable, fetching, blocked, or migrating) at every point in time during execution. An example visualisation of a GranSim log file is given in Figure 2 for the following program.

Example 1.

```
main = print ((a, b) `using` parPair rnf rnf)
    where
      a = expA `using` stratA
      b = expB `using` stratB
```

Note in Example 1 that the definitions for `expA`, `expB`, `stratA`, and `stratB` have been left out for brevity, and that `parPair rnf rnf` is a strategy to reduce the components of a pair to normal form in parallel.

The graph in Figure 2 plots tasks (y-axis) against time (x-axis). Two colours are shown in the graph: black representing blocked threads, and grey representing running threads. Other colours representing runnable, fetching, and migrating do not appear in the profile because the simulation was done with GranSim-Light (which models an idealised machine with an unlimited number of processors and zero communication cost). Note that these graphical profiles are usually viewed in colour, with detail improving with size.

4 Attributing threads to strategies

With the basic `par` and `seq` combinators GranSim records the number of threads and the state they are in, but does not attribute the created threads to any parts of the source program. There are some advanced combinators, however, such as `parGlobal` (see Loidl [7, 8]), that provide a limited capability allowing sparks to be associated with a numerical value which is carried through to the log files. With GranSim-SP a more sophisticated combinator is provided, namely `markStrat`, for annotating all the threads created by a strategies, it has the following type:

```
markStrat :: String -> Strategy a -> Strategy a
```

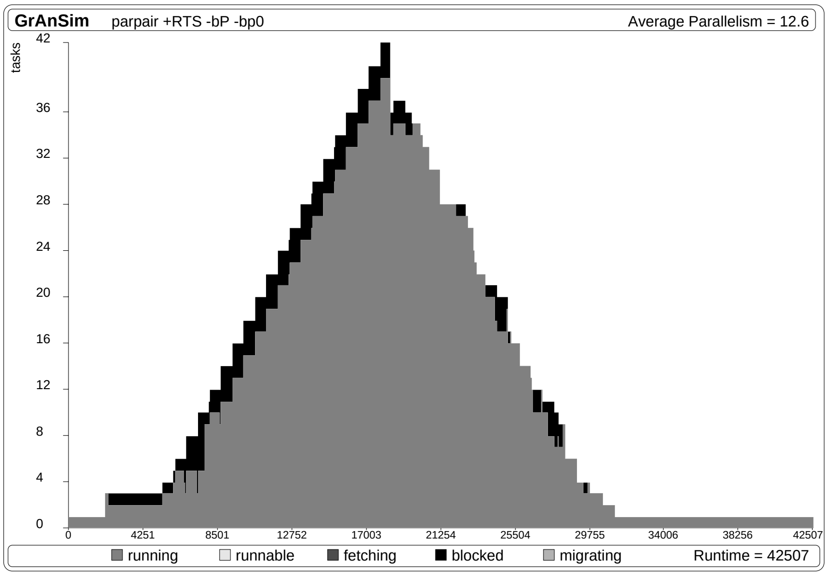

Fig. 2. A GranSim profile for Example Program 1.

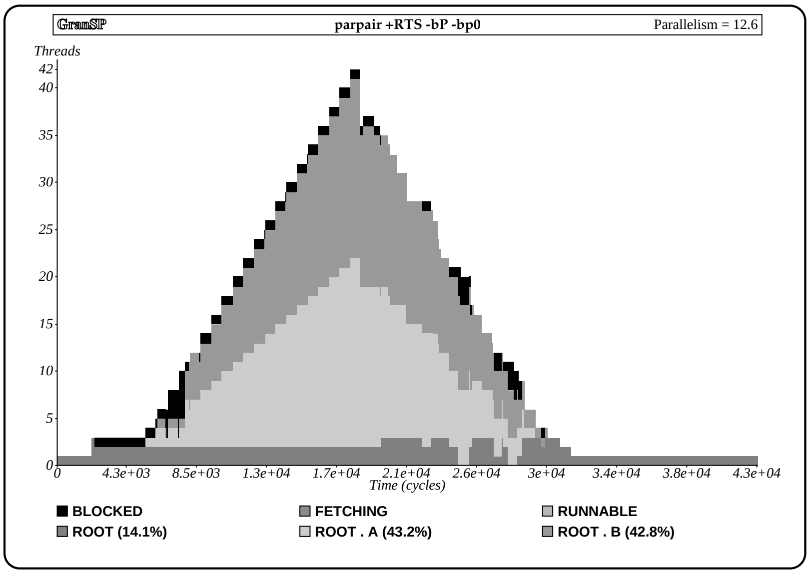

Fig. 3. A GranSim-SP profile for Example Program 2.

the first argument of `markStrat` is the string label that is used to annotate all the threads created in the second argument which must be a strategy. For example, taking the same program (Example 1), strategies `stratA` and `stratB` could be annotated as follows:

Example 2.

```
main = print ((a, b) `using` parPair rnf rnf)
    where
      a = expA `using` markStrat "A" stratA
      b = expB `using` markStrat "B" stratB
```

Three strategies are used in this program, and two have been annotated. The graph in Figure 3 was generated from the log file for the above program. The strategies labelled `A` and `B` are distinguished by different colours (albeit shades of grey) in this paper.

4.1 Recording the relationships between threads

In addition to recording a strategy name for each thread, GranSim-SP also records the child-to-parent relationships between threads. This information has several uses: one is to use the call-path for thread names; 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; in this case the blocked thread can trace its parentage making it possible to discover where it was created in the source program. This is useful information, since using strategies that avoid thread blocking can often give speedups. Morgan and Jarvis [9] have also had success in adding similar relationships to sequential cost centre profiling. They record *call-stacks* of names that corresponds to the function call-path of cost-centres.

The following is an example of a program where two strategies labelled `F` and `G` both use a further strategy labelled `H`.

Example 3.

```
main = print ((f 32, g 16)
            `using` markStrat "parPair" (parPair rnf rnf))

f x = map h [x, 2] `using` markStrat "F" (parList rnf)
g x = map h [x, 3] `using` markStrat "G" (parList rnf)

h x = x + sum (map loop (replicate 8 64)
              `using` markStrat "H" (parList rnf))
      where  loop n = if n==0 then 0 else loop (n-1)
```

In Example Program 3 the thread relationships that are recorded are described in Figure 4. The relationships are also used to build the names in the

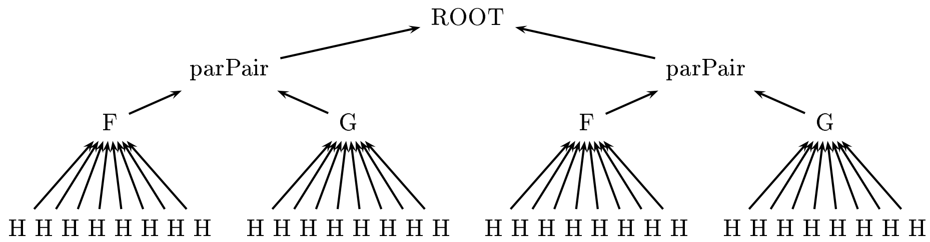

Fig. 4. The thread structure recorded by GranSim-SP for Example Program 3.

graphical profiles. See Figure 5 for a per-thread activity profile, and Figure 6 for an overall activity profile of Example Program 3.

In Figure 4 each node corresponds to a parallel thread. The arrows are upwards, since GranSim-SP only records a parent name for each thread. Maintaining the reverse relationship from parent-to-child would require more storage space, and the relationship can be reproduced anyway.

Per-thread activity profiles A per-thread activity profile plots threads against time, where the threads are coloured depending on their state and which strategy they correspond to. The threads are also sorted by their creation time, and by strategy. Figure 5 is a per-thread activity profile for Example Program 3. Threads are only in two states in this example, running (shown in shades of grey), and blocked (shown in black). The two blocks of threads at the top of the graph represent the threads created by strategy `H`, and are named `ROOT.parPair.F.H` and `ROOT.parPair.G.H`, which describes the two call-paths of strategy `H`. Per-thread activity profiles are more useful with GranSim-SP than with GranSim because the threads are sorted by strategy and are labelled, making it easier to interpret the profile.

Overall activity profiles An overall activity profile plots thread activity against time, and is perhaps the most often used with GranSim. Figure 6 depicts an overall activity profile of Example Program 3. In an overall activity graph, the threads are separated into the five thread states with the running state plotted below other states. With GranSim-SP the running state is broken down into different coloured areas representing each strategy, and again these are sorted by strategy name. The advantage of an overall activity profile over a per-thread activity profile is that it is easier to see the amount of a particular type of activity (e.g. parallelism, or blocking) at any given point in time. The disadvantage is that it is not possible to see which threads are blocking at any point in time.

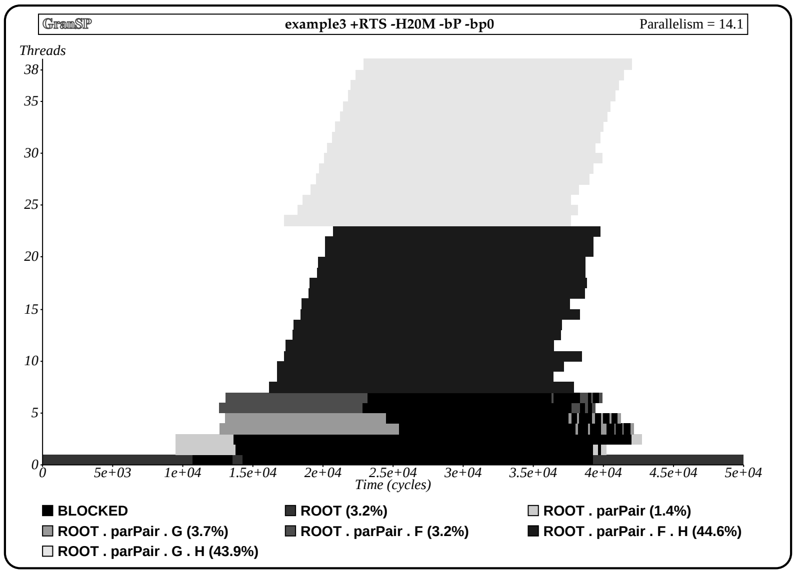

Fig. 5. A GranSim-SP per-thread activity profile for Example Program 3.

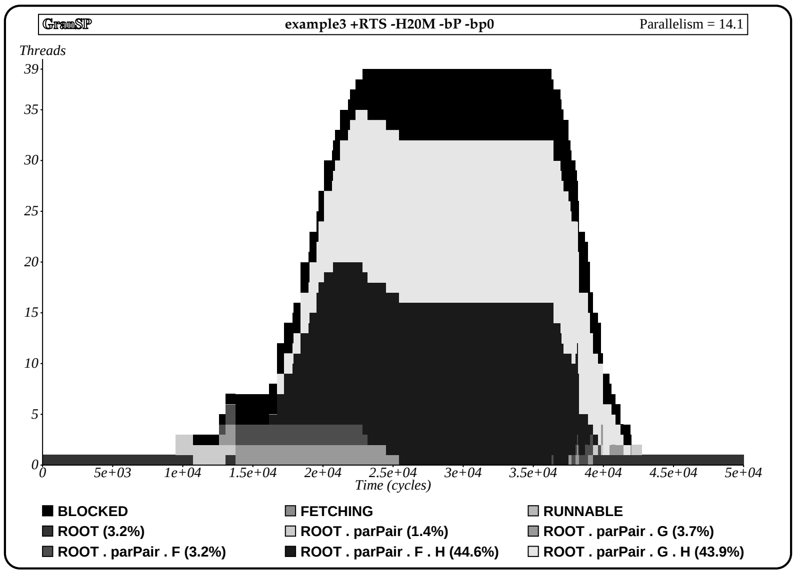

Fig. 6. A GranSim-SP profile for Example Program 3.

5 Implementation

Our implementation of `markStrat` is a small addition to the on-going GpH project. From a user's point of view the function `markStrat` has been added, whose behaviour is to label threads with a given strategy name. When running a program the log files are the same as before (see Figure 1), but with a name and parent identifier on `START` events. The parent identifier is the thread name (a unique hexadecimal number) of the parent thread. For example, the start of the log file for the program in Section 4.1 is given in Figure 7:

```
PE  0 [0]:      START        0      (ROOT , -1)
PE  0 [10519]:  START        1      (parPair , 0)
PE  0 [10540]:  START        2      (parPair , 0)
PE  0 [11790]:  BLOCK        0
PE  0 [13632]:  START        3      (G , 2)
PE  0 [13653]:  START        4      (F , 1)
PE  0 [14040]:  START        5      (F , 1)
PE  0 [14563]:  RESUME       0
PE  0 [14693]:  BLOCK        2
PE  0 [14040]:  START        6      (G , 2)
PE  0 [15333]:  BLOCK        0
PE  0 [14765]:  BLOCK        1
PE  0 [17094]:  START        7      (H , 4)
PE  0 [17669]:  START        8      (H , 4)
PE  0 [18081]:  START        9      (H , 5)
PE  0 [18656]:  START        a      (H , 5)
```

Fig. 7. A log file extract for Example Program 3 that is used by GranSim-SP.

GranSim-SP was implemented by changing the runtime system for GpH, and by writing new visualisation tools. What follows is a sketch of the implementation in the runtime system. First of all extra fields were added to the data structures representing sparks and threads for (*i*) strategy name, and (*ii*) parent identifier. Whenever `markStrat` is evaluated the accompanying label is recorded in the currently active thread's data structure (i.e. the TSO described in Section 3). When a spark is created the strategy name and thread identifier of the current thread is saved in the spark data structure. At the point when sparks are turned into threads the name and thread identifier, held in the spark data structure, are passed to the new thread; this thread identifier being that of the parent thread. Finally, the strategy name and parent identifer are output on the `START` event, which is the most convenient place for the visualisation tools.

The overheads incurred by our implementation are negligible. Indeed, when simulating, no overheads are recorded in the log file. The implementation has not yet been tested on a real parallel machine, but again the overheads should be minor and consistent (i.e. the same overhead is incurred for each thread).

6 Is GranSim-SP useful?

The bottom-line is: *does* GranSim-SP *help a programmer to track down inefficiencies*? A question like this can only be answered properly after many programmers have used GranSim-SP on many different applications over a reasonable period of time. So far, however, the signs are good. GranSim-SP has shown to be useful in understanding the behaviour of a number of non-trivial examples including: a canny edge detector, a ray tracer for spheres, a naughts-and-crosses program, and a word-search program. The word-search program is a good example to look at closely, because it has been well-studied by Runciman and Wakeling ([12], Ch. 10), and it uses non-trivial forms of parallelism. The program searches for a number of words in a grid of letters. The words may appear horizontally, vertically, or diagonally, and either forwards or backwards. The program is made parallel by producing different orientations of the grid in parallel, and then by searching these orientations in parallel. A GranSim profile of the (version 6) program produced by Runciman and Wakeling is given in Figure 8. Originally, Runciman and Wakeling used the HBC-PP compiler, which is a version of the Chalmers HBC compiler that provides a parallel simulator. The profile in Figure 8 is not identical to the original, since GranSim was used to produce it, but it is pleasingly similar, which demonstrates a consistency between the simulators GranSim and HBC-PP.

Taking the original soda program it was re-written to use strategies, and then profiled with GranSim-SP. Using GranSim-SP revealed that the peaks in Figure 8 corresponded to searching the grid in different orientations in turn, i.e. sequentially. This was perhaps not the intention of Runciman and Wakeling, but was caused by a subtlety in their code. By searching all the orientations in parallel it was possible to increase the parallelism and reduce the runtime by approximately 25%, as depicted in Figure 9.

One feature of our profiler that is particularly useful is the recording of parent/child relationships. A common inefficiency is that there is more blocking going on than there needs to be. Blocking is caused by a thread waiting for a result from a child thread, and this can sometimes be avoided by changing a program's evaluation order. Since the parent/child relationships between threads are recorded by GranSim-SP it is possible to locate precisely where the blocking occurs.

7 Related work

In this section a comparison is given with GranSim-CC a parallel cost centre profiler, and some imperative profilers are discussed.

7.1 Comparing GranSim-SP with GranSim-CC

Independently, Hammond et al. [5] have been working on GranSim-CC, a parallel cost-centre profiler. Their work combines sequential cost centre profiling (Sansom and Peyton Jones [13]) with the Glasgow Parallel Haskell runtime system.

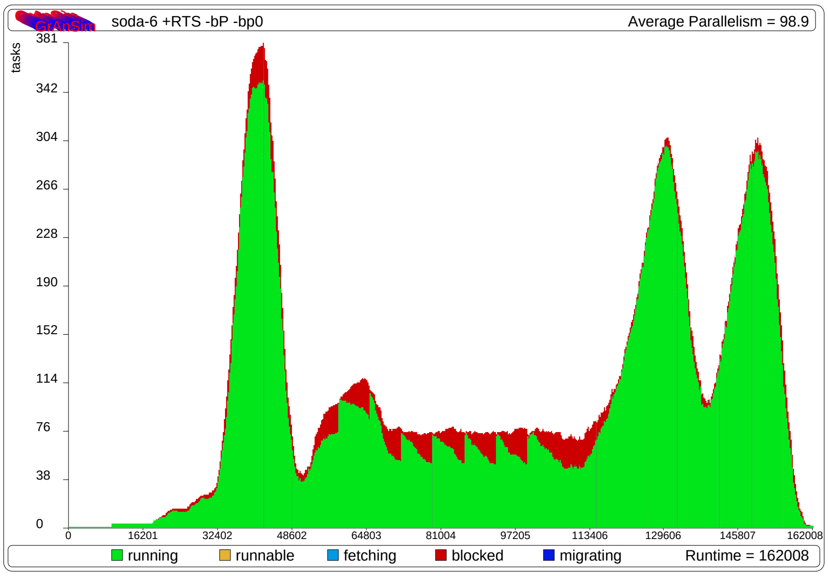

Fig. 8. A GranSim profile for the original soda program.

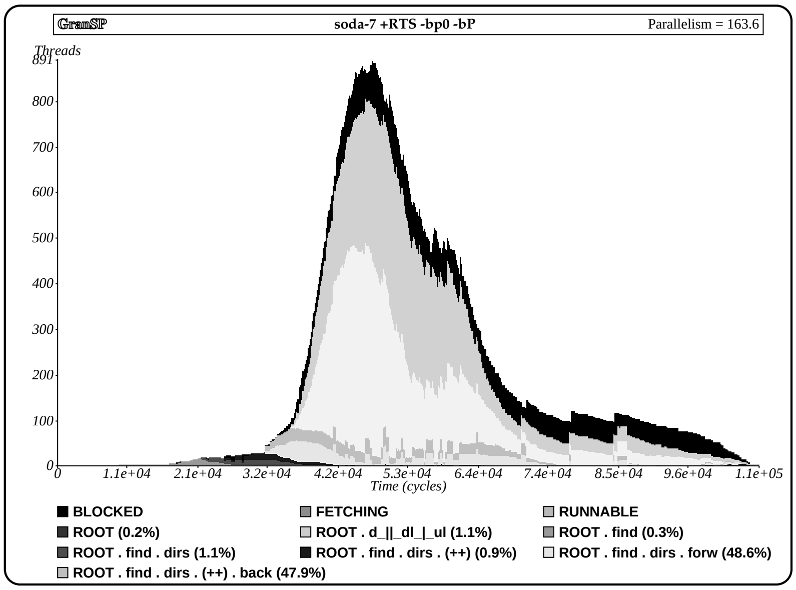

Fig. 9. A GranSim-SP profile for an improved version of the soda program.

GranSim-CC uses the set-cost-centre primitive `_scc_` to annotate expressions. When an expression with `_scc_` is evaluated the given label is recorded in the log file. It is then possible to produce a profile that divides up activity by the cost of these expressions, and attributes them to the expression labels. An example of a GranSim-CC profile for Example Program 3 is shown in Figure 10. This profile should be compared with 2, which is a GranSim-SP profile of the same example. The program had to be modified for GranSim-CC, which involved using `_scc_` around expressions instead of `markStrat` on strategies. In this instance there is a similarity between the GranSim-CC and GranSim-SP profiles which shows a consistency between the two systems. There is a difference, however, which is that GranSim-SP is giving more detail because it is able to distinguish between two separate call-paths to the same function, whereas GranSim-CC is unable to. This highlights one advantage of GranSim-SP over GranSim-CC which is that it records parent-to-child relationships between threads, and makes good use of them. GranSim-CC is also a much more complex system, and as noted by Hammond et al. [5], it hasn't been fully developed. The main advantage of GranSim-CC is that any expression can be annotated, whereas GranSim-SP only allows strategies to be annotated. Therefore, both systems are useful in their own right, and both can provide different and useful information.

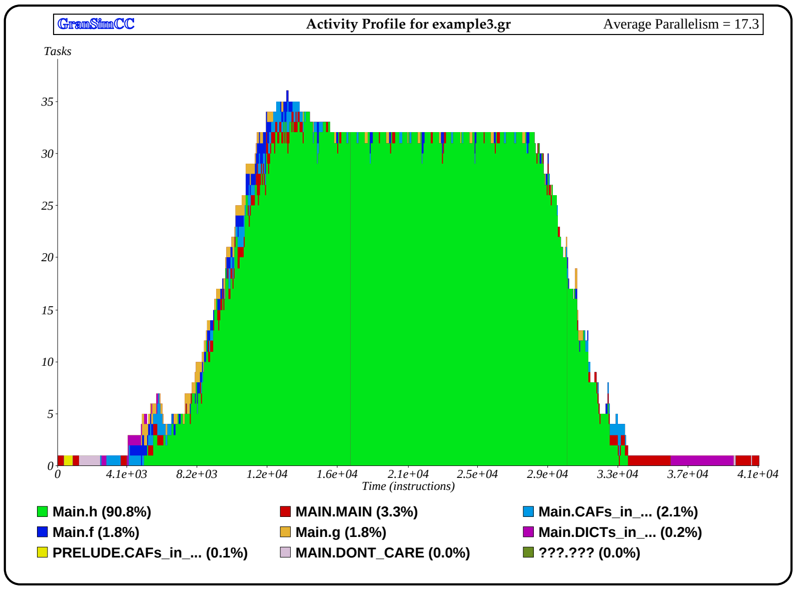

Fig. 10. A GranSim-CC profile for Example Program 3.

Lexical scoping versus evaluation scoping Another difference between using `markStrat` in GranSim-SP and `_scc_` in GranSim-CC is in their scoping rules. For example, in the following expression:

```
map f xs `using` _scc_ "A" strat
```

The cost of evaluating `map f xs` would be attributed to the enclosing cost centre, and not to `A`, because a form of *lexical scoping* is used with GranSim-CC (see Clack et al. [2] for a full explanation of lexical profiling). Essentially, lexical scoping attributes costs to the enclosed expression. If `markStrat` was used in the above example instead of `_scc_`, then all the threads that are sparked when the strategy `strat` is applied to the map expression are labelled. This form of scoping is *evaluation scoping*, and is precisely the semantics one would want for `markStrat`. See the cost centre work of Sansom and Peyton Jones [13] for a discussion about different scoping rules. Sansom and Peyton Jones found that a hybrid of lexical scoping was the most appropriate for the sequential cost centre profiling of Haskell.

7.2 Imperative profiling tools

There are several profiling tools available for imperative parallel languages. One example is *Jumpshot* which is provided with an implementation of MPI (Message Passing Library) called `mpich` (Gropp and Lusk [3]). Jumpshot is an interactive tool written in Java$^{\mathrm{TM}}$, that animates a program's execution displaying parallel time lines with process states (e.g. running, or communicating) for each processor. Another tool is XPVM (Kohl and Geist [6]) which provides a graphical monitor and console to PVM. XPVM is written in Tcl/Tk and is similar to Jumpshot in that it provides an interactive environment showing an animation of the activity on each processor. It's also possible to use XPVM to trace which instruction caused a particular activity. XPVM and Jumpshot differ from GranSim-SP in that they don't relate a section of program source to the displayed histogram. This is perhaps because with imperative parallel languages the evaluation order is static, and the parallelism is explicit, so it is easier to relate source to profile.

8 Discussion and future work

Discussion GranSim-SP has been described in this paper which is an addition to the available profiling tools for Glasgow Parallel Haskell. Its purpose is to help a programmer understand, and thereby improve their parallel Haskell program. The implementation of the profiler was straightforward and fits comfortably into the runtime system for the parallel simulator GranSim. GranSim-SP enables the programmer to annotate threads giving a direct correspondence between program and profile. The usefulness of GranSim-SP was demonstrated on several simple examples, as well as some moderately sized ones.

Future work One obvious improvement is to get GranSim-SP working with the parallel runtime system, so far our efforts have been concentrated on the parallel simulator GranSim. There should not be any major obstacles with this. GranSim-SP has been tested on several examples, and this experimentation plays a crucial role in uncovering shortcomings and ideas for improvement. One improvement is to provide a compiler option to automatically annotate strategies. Automatic annotation has shown to be useful with sequential cost centre profiling, especially for large examples, where the programmer wants to get results quickly without having to make lots of annotations. The only real usefulness of using `markStrat` instead of an automatic name generator is that the programmer is given freedom to use any desired name, and he or she may want to use a common name for more than one strategy.

One unique feature of the sequential cost centre profiling work of Sansom and Peyton Jones [14], is that they give a formal specification for attributing costs to costs centres, and show it equivalent to an operational semantics. The same could be done to justify the behaviour of `markStrat`. First of all though, an operational semantics is needed to model GpH, which is far from trivial. Hall et al. [4] are currently working on such a semantics.

Producing better visualisations of profiles is also of interest to us. The two forms of profile that are shown here, namely the per-thread activity profile, and the overall activity profile, are useful, but why restrict ourselves to these? For example, it may be useful to mask away some strategies, or to zoom in on one part of a profile. This may be of particular value for large scale parallel programs, where there are thousands of active threads. It would also be useful to provide a programmer with the actual call-graph. The call-graph information is currently provided in the log files, and it is used to generate the profiles, but the complete graph is not displayed.

All the above features would best be provided by an interactive graphical interface like those provided by *Jumpshot* and *XPVM*. Charles and Runciman [1] have begun work on such a project for GpH. Their particular interest is to translate log files to a database format, and then to query the information. This high-level interactive approach appears to be a good way forward, and may be of use for profiling other languages, sequential or imperative.

Acknowledgments

This work has been supported by the Research Development Fund at The Open University under the project title APSET (A Parallel Software Engineering Tool). We would also like to thanks everyone that has helped to improve our work, especially the GpH gurus Kevin Hammond and Hans-Wolfgang Loidl.

References

1. N. Charles and C. Runciman. An interactive approach to profiling parallel functional programs. In *The Draft Proceedings of the International Workshop on the Implementation of Functional Languages*, London, Sept. 1998.

2. C. Clack, S. Clayman, and D. Parrott. Lexical profiling: Theory and practice. *Journal of Functional Programming*, 5(2):225–277, Apr. 1995.
3. W. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago, July 1998.
4. J. G. Hall, C. Baker-Finch, P. Trinder, and D. J. King. Towards an operational semantics for a parallel non-strict functional language. In *The Draft Proceedings of the International Workshop on the Implementation of Functional Languages*, Sept. 1998.
5. K. Hammond, H.-W. Loidl, and P. Trinder. Parallel cost centre profiling. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sept. 1997.
6. J. A. Kohl and G. A. Geist. *XPVM 1.0 User's Guide*. Oak Ridge National Laboratory, Nov. 1996.
7. H.-W. Loidl. *GranSim's User Guide*. Department of Computing Science, University of Glasgow, 0.03 edition, July 1996.
8. H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming*. PhD thesis, Department of Computing Science, University of Glasgow, Mar. 1998.
9. R. G. Morgan and S. A. Jarvis. Profiling large-scale lazy functional programs. *Journal of Functional Programming*, 8(3), May 1998.
10. S. L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *23rd ACM Symposium on Principles of Programming Languages*, pages 295–308, St Petersburg Beach, Florida, Jan. 1996.
11. S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: A technical overview. In *Proceedings of the UK Joint Framework for Information Technology, Technical Conference*, pages 249–257, Keele, Mar. 1993.
12. C. Runciman and D. Wakeling. *Applications of Functional Programming*. UCL Press Ltd, 1995.
13. P. M. Sansom and S. L. Peyton Jones. Time and space profiling for non-strict, higher-order functional languages. In *The 22'nd Symposium on Principles of Programming Languages*, pages 355–366, San Francisco, California, Jan. 1995. ACM SIGPLAN-SIGACT.
14. 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), Jan. 1997.
15. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, Jan. 1998.
16. P. W. Trinder, K. Hammond, J. S. Mattson, A. S. Partridge, and S. L. Peyton Jones. GUM: a portable implementation of Haskell. In *Proceedings of Programming Language Design and Implementation*, Philadelphia, USA, May 1996.