From GranSim to Paradise[*]

Félix Hernández[1], Ricardo Peña[2], Fernando Rubio[3]

Universidad Complutense de Madrid, E-28040 Madrid, Spain

[1] e-mail: `fhernand@eucmos.sim.ucm.es` Fax: 34-91-3944602
[2] e-mail: `ricardo@sip.ucm.es` Phone: 34-91-3944313
[3] e-mail: `rubiod@eucmax.sim.ucm.es` Phone: 34-91-3944350

Abstract. We describe PARADISE (PARAllel DIstribution Simulator for Eden) a simulator developed to profile the execution of programs written in the parallel functional programming language Eden [BLOMP96], [BLOMP97]. Eden extends the lazy functional language Haskell by syntactic constructs to explicitly define processes.
Paradise is a substantial modification of GranSim [HLP94], [Loi96], [Loi98], a tool to study the dynamic behaviour of GpH (Glasgow Parallel Haskell [THJ$^+$96]) programs.
We describe the basic differences, in the one hand between GpH and Eden, and in the other hand between GranSim and Paradise. After that, we present the new facilities we want for our simulator and the current status of the implementation. Examples of actual and foreseen graphics are given along the text.

1 Introduction

The parallel functional programming language Eden [BLOMP96], [BLOMP97], [BKL$^+$98b], [KOMP98] extends the lazy functional language Haskell by syntactic constructs to explicitly define processes. Section 3 gives a brief account of the main Eden features. Eden's compiler is implemented by modifying the Glasgow Haskell compiler GHC [JHH$^+$93], and Eden's runtime system has been implemented by taking as starting point the GUM [THJ$^+$96] runtime system. In [BKL98a], more details are given.

The basic reason for using a parallel language is to increase the efficiency of programs by using several processors. Unfortunately, it is not simple to write efficient parallel programs, or to reason about their runtime behaviour. Some kind of feedback is needed in order to know whether our programs are well parallelized and to understand the reasons of possible inefficiencies.

One way of obtaining that feedback is the use of a simulator such as GranSim [HLP94], [Loi96], [Loi98]. GranSim is a tool to study the dynamic behaviour of GpH (Glasgow Parallel Haskell, [THJ$^+$96]) programs. It can simulate GUM —GUM is the runtime system of GpH— and a wide range of architectures by adjusting parameters such as number of processors, communication latencies and other.

In this paper, we describe PARADISE (PARAllel DIstribution Simulator for Eden), a substantial modification of GranSim, developed to deal with Eden programs instead of with GpH programs.

The organization of the paper is as follows: in Section 2 we summarize the main characteristics of GpH and of GranSim; in Section 3, an explanation of Eden's features relevant to this paper is given together with a simple Eden parallel program which will be used in subsequent sections to illustrate different aspects of Paradise; Section 4 explains the main differences between GranSim and Paradise and presents the current and future functionalities of this tool. Some preliminary graphics are also shown. Section 5 gives an account of the current state of the implementation and Section 6 draws some conclusions.

[*] Work partially supported by the spanish projects CAM-06T/033/96 and CICYT-TIC97-0672.

2 GranSim

Paradise is closely related to GranSim, the Glasgow parallel Haskell (GpH [THJ+96]) simulator. This system is a fundamental milestone in the analysis of functional parallel programs, providing an accurate and flexible way of studying the dynamic behaviour of GpH programs.

GpH extends Haskell by adding a new parallel operator (`par`) that, in combination with the `seq` operator allows the programmer to control the parallel behaviour of the programs. The declarative semantics of these operators is

```
⊥ `seq` y = ⊥
x `seq` y = y
x `par` y = y
```

In operational terms, `seq` forces the evaluation to WHNF[1] of its first argument, and then proceeds with the evaluation of its second argument. Operator `par` is just an annotation whose meaning is: the first argument can be evaluated in parallel with the second one, but it is not mandatory (the RTS[2] will decide whether a new thread should be created or not).

The simulator supports extensive tuning of the simulated architecture, having parameters such as number of processors, communication latencies, and others. Additionally, an ideal simulation mode (the *light* mode) simulating an unlimited number of processors is provided in order to study the parallelism upper limit of a program.

The implementation is based on a threaded runtime system that collects lot of runtime information, such as the start and end time of each thread, the moments in which they get blocked or resumed, etc. This information is stored in a log file, and then is postprocessed in order to obtain different graphics.

The core structure of the implementation is a 'global event queue' which stores the list of events which will occurr in the future, ordered by the moments in which they have to be treated. Basically, for each type of message that can be passed from one processor to another, a different kind of global event exists, such as START a new thread, or MOVESPARK from one processor to other. Therefore, to simulate sending a message, GranSim adds a new global event in the appropriate position of the global event queue, so that it is treated in the right moment.

GranSim provides several visualization tools to help analyze the simulation results. Two kinds of graphical profiles are supplied: activity profiles and granularity profiles. Activity profiles give an overview of the evolution of parallel programs along the time. Three different views are provided:

1. Per-thread activity profile.
2. Per-processor activity profile.
3. Global activity profile.

The *per-thread activity profile* shows each generated thread providing information about its state (running, runnable, blocked, fetching or migrating) along the time. The *per-processor activity profile* shows the evolution in time of the status of each processor (active or idle), its workload and the number of blocked threads. The *global activity profile* shows the evolution in time of the number of threads in each possible execution state (running, runnable, blocked, fetching or migrating).

Granularity profiles depict the threads grouping them by execution time or allocated heap size.

An important limitation of GranSim is that simulation profiles cannot be directly related to the source code, preventing the programmer from quickly identifying the sources of inefficiency.

[1] Weak Head Normal Form.
[2] Runtime System.

For instance, the *per-thread activity profile* is not very useful unless the number of threads is rather small. A key issue in the design of Paradise has been to overcome this constraint.

3 Eden

The parallel functional programming language Eden extends the lazy functional language Haskell by syntactic constructs to explicitly define and communicate processes.

There exist a new expression `process x -> e` of a predefined type `Process a b` to define a *process abstraction* having variable `x::a` as input and expression `e::b` as output. Process abstractions of type `Process a b` can be compared to functions of type `a -> b`, the main difference being that the former, when instantiated, are executed in parallel. Additionally, when the output expression is a tuple, i.e. `e::(t1,...,tn)` a separate concurrent thread is created for the evaluation of each tuple element. When the input (resp. output) of a process is a tuple, we will refer to each tuple element as a *channel*. If it is not a tuple, we will say that the process has a single input (resp. output) channel.

A *process instantiation* is achieved by using the predefined infix operator `(#)::Process a b -> a -> b`. Each time an expression `e1 # e2` is evaluated, a new process is created. We will refer to the latter as the *child* process, and to the owner of the instantiation expression as the *parent* process. The instantiation procotol deserves some attention in order to understand the differences between GranSim and Paradise:

- Closure `e1` together with all its dependent closures are *copied* unevaluated to a new processor and the child process is created there to evaluate it.
- Once created, the child process starts producing eagerly its output expression.
- Expression `e2` is eagerly evaluated in the parent process. If it is a tuple, an independent concurrent thread is created to evaluate each component.

Once a process is running, only fully evaluated data objects are communicated through channels. The only exception are lists: they are transmitted in a *stream*-like fashion, i.e. element by element. Each list element is first evaluated to normal form and then transmitted. Concurrent threads trying to access not yet available input will be suspended. This is the only way of synchronizing Eden processes.

We can summarize the main differences between Eden and GpH as follows:

1. Process abstractions in Eden are not just annotations but first class values which can be manipulated by the programmer (i.e. communicated through channels, stored in data structures, and so on).
2. In Eden, the concept of a virtually shared global graph does not exist. Each process evaluates its outputs autonomously with respect to other processes. The entire graph needed by a newly instantiated process is copied into its heap *before* it starts running. In some cases, this can even lead to some duplication of work.
3. In Eden, lazy evaluation is changed to eager in two cases: processes are eagerly instantiated when a binding `o = e1 # e2` is found while evaluating a `letrec` binding list. Also, instantiated processes produce their output even if it is not demanded. These semantics modifications are aimed at increasing the degree of parallelism and at speeding up the distribution of the computation. Notice that, if not appropriately used, these features can lead to speculative work.
4. In general, a process is implemented by several threads concurrently running in the same processor.

The following Eden program specifies a parallel mergesort algorithm, where a parameter `h` controls the granularity by specifying the desired height of the process tree to be generated. It will be used in the following sections to illustrate different aspects of Paradise.

```
parMsort h = process xs -> ys
   where ys | h == 0 = seqMsort xs
            | h > 0  = ordMerge (parMsort (h-1) # x1) (parMsort (h-1) # x2)
                       where (x1,x2) = unshuffle xs
seqMsort []  =  []
seqMsort [x] =  [x]
seqMsort xs  =  ordMerge (seqMsort x1) (seqMsort x2)
                where (x1,x2) = unshuffle xs

unshuffle = foldr assign ([],[]) where assign x (xs1, xs2) = (xs2, x:xs1)
```

Notice the difference between parameter `h`, which must be known at instantiation time, and the input `xs`, i.e. the list of values to be sorted, which will be gradually provided during the execution of the process. Ignoring parameter `h`, the similarity between the sequential and parallel versions is remarkable. While the former makes two function applications —function `seqMsort`—, the latter produces two process instantiations —process abstraction `parMsort (h-1)`. Function `unshuffle` splits a list into two by putting odd position elements into one list and even position ones into another. Function `ordMerge` merges two ordered lists into a single ordered list and is not shown.

4 Paradise

4.1 Basic Paradise

As a first approach, we want to obtain the same kind of graphics that GranSim provides. That is, we want to view the global behaviour of the program (see Figure 1), the behaviour per processor, per thread and also the granularity graphics. In other words, we want to transform GranSim into an Eden simulator, instead of a GpH simulator.

In addition to some low level differences between the underlying implementation of Eden and GpH, the main conceptual differences we have to deal with are the following:

– In Eden, is the programmer who decides if a thread has to be created or not. The RTS is not allowed to change this decision. In contrast, in GpH the programmer only suggests which parts of the code can be executed in parallel, but is the RTS who decides whether a thread is actually created or not.
– Eden needs a fair scheduler, while GpH does not.
– The load balancing is quite different in Eden. In GpH, idle processors ask for work to the rest of processors, and in case they find some potential work (in form of sparks), they 'steal' it. On the other hand, Eden does not allow stealing sparks (nor threads): at process creation time, the creator decides which processor is going to evaluate the new process (and all its threads), and that decision cannot be changed afterwards.
– Eden allows duplicating work in different processors, so that one closure can be evaluated more than once. This implies that, when copying a part of the closures graph to a remote processor, we must perform a real copy, and not just a simulation (as GranSim does). Otherwise, different processors could be sharing the evaluation of the same closure, and that is neither possible nor desirable in Eden.
– A new type of global event needs to be introduced for each Eden message which can be passed between processors (these are mainly to create new processes and to send values from one process to another, but also some others to correctly simulate the protocols of Eden's RTS).

As GranSim is highly configurable, and one of its options is to automatically convert every spark into a thread, the first point is quite easy to solve. GranSim is also prepared for using a fair scheduler, and so only a couple of changes have been needed. So, the main part of the work has been the implementation of the other three points.

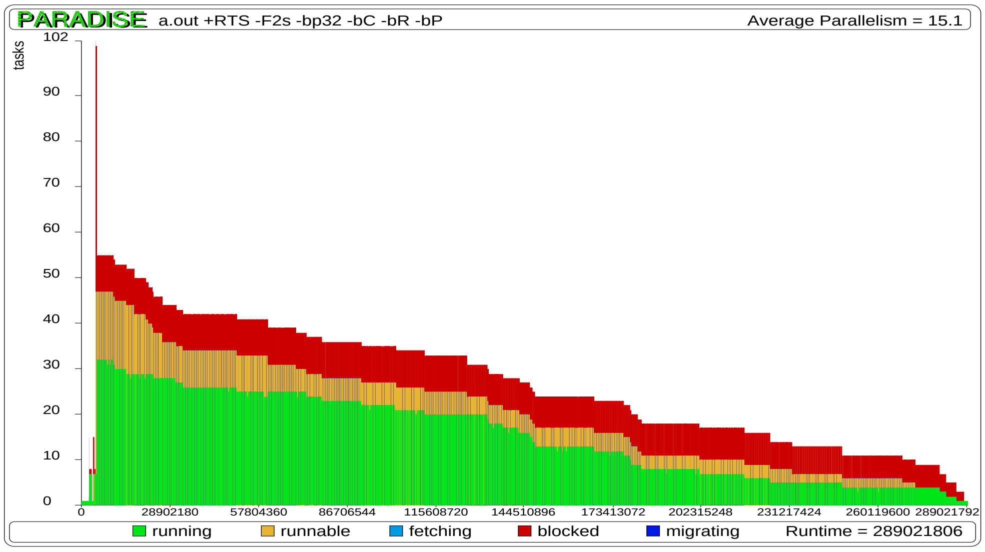

Fig. 1. Global behaviour of the minimax algorithm when simulating 32 processors for playing Checkers.

4.2 Full Paradise

With the graphics of the previous section, it is not easy to know which part of the source code is the responsible of possible inefficiencies. We can solve this by adding a *mark system* which relates the output to the source code, and by developing new *visualization tools* that use that relationship to generate graphics with more information.

It has been said in Section 3 that an Eden program can lead to speculative work. For the programmer, it is not easy to predict the amount of speculative work done by his program, as this can depend on RTS features he cannot control. An Eden profiler should provide the feedback the programmer needs, in the form of *speculation graphics*, so that he can correct his programs when they are speculating a lot.

The mark system The basic idea to relate the output to the source code is to give names to the threads (by adding a new *mark* field to each TSO[3]), and to introduce a predefined function `mark` which is very similar to the `markStrategy` function introduced in the Strategic Profiler [KHT98]. The declarative meaning of our `mark` function is

```
mark :: String -> a -> a
mark name expr = expr
```

but its real behaviour at runtime, is producing a side effect consisting of the following modification of the name of the current thread:

```
currentTSO.mark <- name ++ "." ++ currentTSO.mark
```

The thread evaluating the function changes its name, appending a new one to its past name.

The type of `mark` is very general, and it could be used anywhere in the source code, but we should restrict ourselves to use it only where it makes sense. The methodology is that we should only use it just after a new thread is created. The programmer may introduce the marks in three different points, depending on the information he wants to obtain:

[3] TSO: Thread State Object. It is a register that stores all the relevant information about a thread.

process abstractions: if he just wants to know how much parallelism generates each process abstraction, he can associate a name to each of them:

```
myProcess = mark "mp" (process x -> ...)
```

Using this kind of annotations, we can detect the amount of parallelism generated by each process abstraction, as all the threads of the same abstraction will be marked with the same name.

process instantiations: sometimes we are interested not only in distinguishing different abstractions, but also different instantiations of the same process abstraction. For instance, if we have a mergeSort process, we could use a different mark for left instantiations and for right instantiations:

```
parMsort h = process xs -> ys
   where ys | h == 0 = seqMsort xs
            | h > 0  = ordMerge (mark 'L' (parMsort (h-1)) # x1)
                                (mark 'R' (parMsort (h-1)) # x2)
                       ...
```

Note that it is possible to distinguish not only between left and right processes, but also between internal nodes and leaves. The reason is that `mark` does not completely change the name of the thread, but only appends a new name. This means that each thread remember its full instantiation path. So, if the depth of the instantiation tree is two, the leaves are labeled with names such as "L.R.Main"[4], "L.L.Main", etc, while internal nodes are named "L.Main" or "R.Main". Therefore, we could use some kind of wild card in the visualization tools to collapse all leaves into a single class, and all internal nodes into another one. This is explained in the next subsection, and the result can be seen in Figure 2).

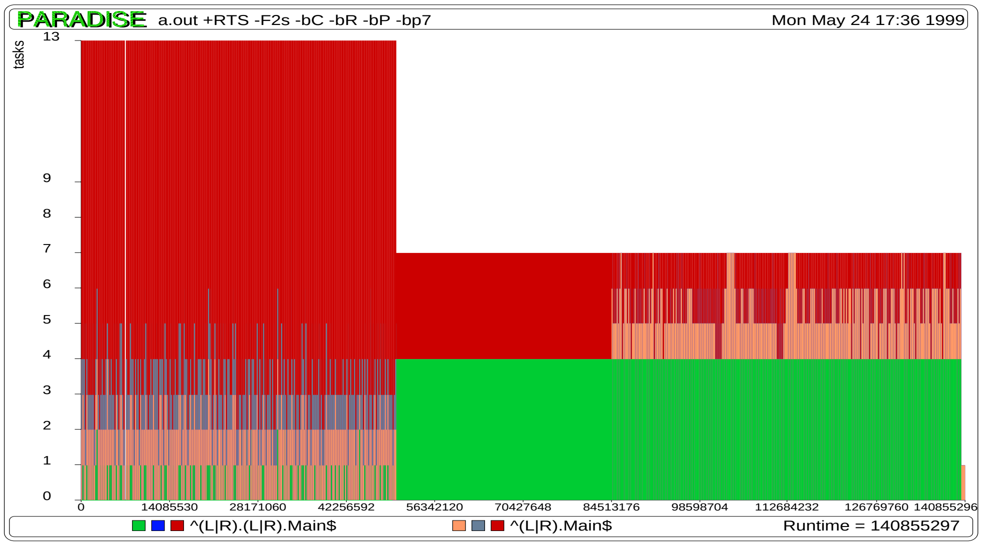

Fig. 2. Global behaviour of the mergeSort algorithm when simulating 7 processors. The three colors associated to each mark represent (from left to right) the running, runnable and blocked threads. It is clear that leaves do most of the work, so that the processors associated to the internal nodes are idle most of the time.

[4] Initially, the main thread is labeled with the name "Main". Therefore, all the threads inherit this initial name.

threads: each process may contain several threads, and we may be interested in distinguishing each of the threads of a process abstraction or instantiation. This is quite easy, as Eden threads are easily identifiable: we have a thread for each element of the output tuple of a process. So, if we have a process with three output channels, we can associate a different name to each of the threads by writing

```
p = process (in1,in2) -> (mark "o1" out1, mark "o2" out2, mark "o3" out3)
```

Analogously, we can give different names to each of the threads responsible of sending values to a newly instantiated process:

```
p # (mark "i1" in1, mark "i2" in2)
```

Paradise Visualization Tools GranSim visualization tools have been adapted to Paradise, and therefore activity and granularity profiles are available for the analysis of Eden programs. In addition to GranSim graphics, Paradise provides new ones that take advantage of the mark system. Now, the global behaviour of our programs is not divided into a fixed number of bands (running, runnable and blocked), but in a variable number, as we have three bands for each different mark (see Figure 2). The behaviour per thread is also shown in a different way now, because we can see the mark associated to each thread (see Figure 3).

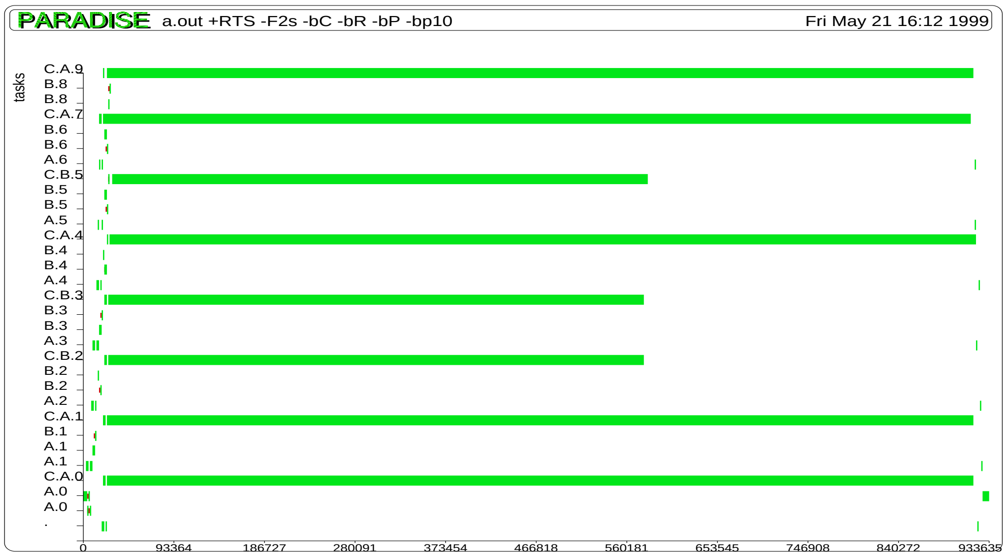

Fig. 3. Per thread graphic sorted by processor and by lexicographical order. The last number of each name indicates the processor.

Sometimes we do not want to see all the information provided by the mark system. That is why Paradise visualization tools allow the following transformations:

unification: marked entities matching a regular expression are shown under the same mark.
filtering: all marked entities matching a regular expression are not rendered.
processor labeling: marks are extended with their host processor number.
time zoom: only activities in a given time interval are shown.
ordering: the threads are lexicographically ordered and/or ordered by processor (in the *per thread graphic*).

The mark system allows us to give different names to each process instantiation or even to each thread, but it can be difficult to extract information from a graphic with too many bands. That is why we allow the *unification* of a family of marks. For instance, in Figure 2 we use regular expressions[5] to join all the internal nodes of the mergeSort topology in a unique mark, and all the leaves in another one. We can use any regular expression, obtaining a very flexible way of joining threads.

Filtering a family of threads is another useful way of restricting the output of the tool. For example, if we detect an inefficiency in a family of marks, we can forget about the rest, studying only the interesting part of the results.

By using unification and filtering, we can obtain different views of the information without changing the source code in order to introduce or eliminate marks. So, we do not need to recompile and rerun our programs, saving much time.

Processor labeling gives us a way to relate *per processor* information with *per thread* and *global behaviour* graphics. This can be quite useful, as inefficiencies may be due to the load balancing, and not to the source code. Note that when the processor number is appended to the marks, all the power of the previously mentioned transformations can be used, so that it is easy to join or to eliminate threads located in the same processor.

Another feature supported by Paradise visualization tools is the *time zoom*. GranSim and Paradise's simulation of programs intensive in communications and with long execution times can generate very complex profiles whose representation can exceed the screen resolution. To solve this problem, Paradise activity profiles can post-process simulation results to restrict the graphical representation to any concrete period of time.

Speculation Eden is not a purely lazy language, as each thread evaluates its output even if nobody demands it. So, the programmer must take care of this circumstance in order not to write programs speculating a lot.

For this reason, Paradise will provide a *speculation graphic* (Figure 4). The idea is that for each mark[6] we want to know the evolution in time of the amount of data that has been sent as output of threads of this mark, and that have not been consumed yet by the receiver. By doing so, we can see not only how much speculation produces each mark, but also the 'races' between processes. That is, even if there is no speculation at all, we can see whether a producer is very fast with respect to the consumer (we will see a graphic that end with zero, but that is quite wide during the computation), or whether it is very slow (the graphic will show a zero-wide band). In the latter case, we will detect that the producer is a bottleneck.

This graphic does not show the real amount of time spent performing speculative work, but the amount of data that has been speculatively sent. Even though this is not the most desirable output, we think it can be quite useful.

One possible implementation consists of adding two fields to each TSO. The first one (the *produced* field) contains the amount of data that has been sent by the thread through its output channel, while the second one (the *consumed* field) contains how much of these data have actually been used by the receiver. Sampling these fields at regular intervals, we can easily obtain the desired graphics. The main difficulty is to maintain the correct values in the fields. To do so, we need to add a new *speculation* field to each closure. This field is zero if the closure has been locally created or if it has already been used. Otherwise (i.e. it has been sent by a remote thread, and it has not been used yet), it contains the sender thread identity. With this information, the following actions will be done in order to keep the correct values in the TSO fields: (1) Each time a message is sent, the corresponding field is increased (this part is easy,

[5] ^(L|R).Main$ must be read as: it starts with an 'L' or and 'R', then there is a '.', and finally appears 'Main'. If we only use (L|R), this means: all the marks that includes an 'L' or an 'R' in any position.

[6] This amounts to say for each process abstraction, process instantiation, thread, or processor.

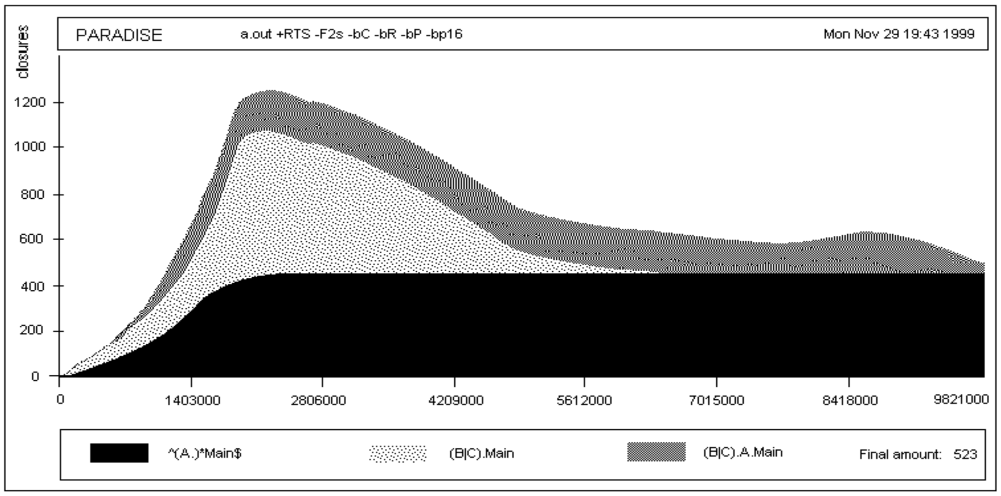

Fig. 4. Speculation graphic.

as there are only two routines sending values), and the sent closures are marked with sender's thread identity. (2) When a thread uses a closure, if its speculation field is not zero, it is set to zero and the corresponding *consumed* field in sender's thread TSO is increased.

5 Current state of the implementation

At the moment, we have almost fully implemented what we call *Basic Paradise*, i.e. we are able to obtain the same graphics as GranSim. The only thing not yet implemented is the *light* mode to simulate an ideal setup with infinite processors. We still have some problems with the copy of closures.

We have also implemented the visualization tools described in Section 4.2 (but the speculation graphics generator). We have not yet implemented the `mark` function, only a very primitive approximation to it.

6 Related work, future work and conclusion

There are another two works, the parallel cost center profiler, or GranCC [HHLT97], and the strategic profiler [KHT98] which relate GranSim output to the source code. The main difference between our approach and that of GranCC, is that the former uses evaluation scoping, while the latter uses lexical scoping, which does not make much sense for Eden processes, as we are interested in knowing exactly what evaluates each process (a detailed explanation of the differences between lexical and evaluation scoping can be found in [CCP93]).

In comparison with the Strategic Profiler, our `mark` function is similar to their `markStrategy`, the main difference being that the latter marks all the threads created as a consequence of applying a 'strategy', a programming concept related to skeletons that normally involves many parallel activities. Our function is more flexible in the sense that we can mark each process abstraction, each process instantiation or even each thread with a different name.

In the next future, we hope to complete the implementation of all the described facilities not implemented yet and to be able to provide more examples and more interesting graphics.

Also, we will be in a position to compare the simulated results with the actual figures obtained in real parallel executions.

We believe that Paradise is going to be quite useful for the Eden programmer. For the moment, it has been useful for us, as it has helped us in the development of the Eden compiler. The use of Paradise has suggested an automatic transformation for eagerly instantiating processes, and this transformation has increased the parallelism degree of several of our programs. These programs includes calculating Mandelbrot sets, a checkers player, and the mergeSort algorithm.

In conclusion, we think that —as it has been the case with GpH— the use of a simulator is a very valuable tool to know the detailed behaviour of parallel programs, providing an unquestionable feedback both for the development of parallel algorithms and for the development of the compiler of such programs.

References

[BKL98a] S. Breitinger, U. Klusik, and R. Loogen. From (sequential) Haskell to (parallel) Eden: An implementation point of view. In PLILP'98. Springer LNCS 1490, pages 318–334, 1998.

[BKL+98b] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Peña. DREAM: the Distributed Eden Abstract Machine. In *Implementation of Functional Languages, IFL'97. St. Andrews, Scotland, LNCS 1467*, pages 250–269. Springer-Verlag, 1998.

[BLOMP96] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report, Bericht 96-10, revised version, Philipps-Universität Marburg, Germany, 1996.

[BLOMP97] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *Workshop on High-level Parallel Programming Models, HIPS'97. In conjuntion with the IEEE International Parallel Processing Symposium, IPPS'97*, pages 120–124. IEEE Computer Science Press, 1997.

[CCP93] C. Clack, S. Clayman, and D. Parrott. Lexical profiling: Theory and practice. Journal of Functional Programming, 1993.

[HHLT97] K. Hammond, C. V. Hall, H.W. Loidl, and P. W. Trinder. Parallel cost centre profiling. Available from authors, 1997.

[HLP94] K. Hammond, H. W. Loidl, and A. Partridge. Visualising granularity in parallel programs: A graphical winnowing system for Haskell. Department of Computing Science. University of Glasgow, 1994.

[JHH+93] S. L. Peyton Jones, C. V. Hall, K. Hammond, W. D. Partain, and P. L. Wadler. The Glasgow Haskell Compiler: A Technical Overview. In *Joint Framework for Information Technology, Keele, DTI/SERC*, pages 249–257, 1993.

[KHT98] D. J. King, J. Hall, and P. W. Trinder. A strategic profiler for Glasgow parallel Haskell. Proceedings of the IFL-98, 1998.

[KOMP98] U. Klusik, Y. Ortega-Mallén, and R. Peña. Implementing Eden - or: Dreams Become Reality. In *Implementation of Functional Languages, IFL'98, London, Sept. 1998. LNCS 1595, Springer-Verlag*, pages 1–16, 1998.

[Loi96] H. W. Loidl. Gransim user's guide. Department of Computing Science. University of Glasgow, 1996.

[Loi98] H. W. Loidl. Granularity in large-scale parallel functional programming. Ph.D thesis. Department of Computing Science. University of Glasgow, 1998.

[THJ+96] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 1996.