An AMPLE Implementation

Ben Lippmeier and Clem Baker-Finch

Department of Computer Science
Australian National University, Canberra, ACT 0200, Australia
Ben.Lippmeier@anu.edu.au, Clem.Baker-Finch@anu.edu.au

Abstract. Ample is an Abstract Machine for Parallel Lazy Evaluation. Ample was created as an experimental environment in which to study the behaviour of lazy functional programs when running on parallel machines with various characteristics. Ample is designed to be highly configurable and all configuration parameters may be set interactively. This configuration extends to the number of processors in the machine, the communication latency and the method used for spawning and unblocking threads. Ample includes tools to generate graphs of profiling data and allows program execution to be traced step by step. Ample has been written entirely in Haskell and has been constructed in a modular way. Ample is intended to be modified by end users in order to investigate their own machine designs.

1 Introduction

During the design of a system for parallel evaluation it is useful to profile the performance of several archetypal programs. This is done in order to determine whether the decisions taken at design time will lead to the efficient execution of programs once the system has been implemented. Unfortunately, profiling support is usually something which is added *after* large parts of a system have already been implemented. Although it may be possible to hand-evaluate trivial programs at design time, answering seemingly straight forward questions such as "How many processors can this program make use of?" is surprisingly hard to do without having access to a good run-time profiler. [5]

One possibility is to make use of an abstract machine in order to profile small programs and test out ideas. By using an abstract machine, important questions regarding the behaviour of a system may be answered without having to build a native implementation. Compared to a native implementation, it is also easier to modify an abstract machine as the design changes and to reason about how it operates.

Ample implements the parallel lazy abstract machine described in [1]. This machine is based on the operational semantics for parallel lazy evaluation given in [2] as well as Sestoft's derivation of a (sequential) lazy abstract machine [6]. These are SECD style environment machines which incorporate a model of parallelism consisting of a pool of threads that communicate via a common heap.

Two versions of Ample have been implemented, based on Sestoft's Mark-1 and Mark-3 machines respectively. The Mark-1 machine uses direct substitution to perform function application whereas the Mark-3 machine uses a separate environment to model this substitution. The two versions accept a similar source language which differs only in details specific to the abstract machine used. This paper focuses on the version based on Sestoft's Mark-3 machine.

Ample implements three different execution modes: single threaded, fully speculative and explicitly parallel. In the single threaded mode a sole thread exists and reductions are performed in a sequential manner. In the fully speculative mode one thread evaluates the function body while a new thread is created to evaluate the argument. In the explicitly parallel mode parallelism is controlled with the `par` and `seq` combinators in a manner similar to GpH [7].

Ample allows configuration parameters to be tuned interactively. This configuration extends to the number of processors in the machine, the communication latency and the method used for spawning and unblocking threads. Ample also provides a rich set of profiling tools that can be used to study the effect that these parameters have on the run-time behaviour of the machine.

Ample has been written entirely in Haskell and has been constructed in a modular way. This modularity is intended to allow the user to modify Ample to meet their own requirements.

2 Core language

Ample accepts programs written in a small lazy functional language. This language is based on the untyped lambda calculus, augmented with mutually recursive let-bindings, integers, constructors and the `par` and `seq` combinators. An abbreviated syntax for this language is shown in figure 1. The Ample compiler translates program source into the Ample *core* language by converting it to the normalized lambda calculus while replacing variable names with de Bruijn indices as per Sestoft's Mark-3 machine. The compiler also builds environment trimmers for let expressions and constructors [6].

At run-time the program exists as a graph of 'cells' which are stored in a static heap. Each cell is referenced by its offset (pointer) into the heap and the edges of the graph are represented by including these pointers in its cells.

$$\begin{array}{lll}
\mathrm{e} & = & \lambda \mathrm{x}\,.\,\mathrm{e} \mid \mathrm{e}_1\ \mathrm{e}_2 \mid \mathrm{x} \\
 & \mid & \textbf{let}\ \{\ \mathrm{x}_i\ \overline{\mathrm{y}}_i = \mathrm{e}_i\ \}\ \textbf{in}\ \mathrm{e} \\
 & \mid & n \mid \mathrm{e}_1\ \mathit{op}\ \mathrm{e}_2 \\
 & \mid & \mathrm{C}_i\ \overline{\mathrm{y}}_i \mid \textbf{case}\ \mathrm{e}\ \textbf{of}\ \{\ \mathrm{C}_i\ \overline{\mathrm{y}}_i \rightarrow \mathrm{e}_i\ \} \\
 & \mid & \mathrm{e}_1\ \textbf{par}\ \mathrm{e}_2 \mid \mathrm{e}_1\ \textbf{seq}\ \mathrm{e}_2
\end{array}$$

Fig. 1. abbreviated syntax for the Ample source language

3 The AMPLE abstract machine

The AMPLE abstract machine is a combination of Sestoft's Mark-3 machine [6] and the Parallel Lazy Abstract Machine described in [1]. The machine is defined by a set of state transition rules, the full set of which is given in the appendix. In single threaded mode the machine operates as per Sestoft's Mark-3 machine and the two parallel modes are defined by modifying the set of rules for single threaded mode.

3.1 Sestoft's Mark-3 machine

Sestoft's Mark-3 machine of consists of a *control* C, an *environment* E, a *stack* S and a *heap* H.

The control holds the sub-expression currently being evaluated. In the AMPLE machine the control is a pointer to one of the program cells described in the previous section. Each cell represents one node of the program graph. In this configuration the control is equivalent to the program counter in an imperative machine.

The environment eliminates the need to modify the live program during *β-reduction*, as is done by Sestoft's Mark-1 machine. In the Mark-3 machine the environment is a stack of heap pointers p. Where the Mark-1 machine would perform the direct substitution $e[y/x]$, modifying the original expression e, the Mark-3 machine inserts y together with a copy of its associated environment into the heap. The resulting heap pointer is then added to the environment for e. At compile time each named variable x is replaced by an integer offset into the environment – so that when the machine requires the value of that variable it can be obtained by following the associated heap pointer.

The heap now maps pointers p onto *closures* (e, E) which represent the result of substituting values referenced by the environment E into the expression e.

The stack represents the context of the evaluation. For example, to evaluate an application $e\,x$ the corresponding heap pointer p_x is pushed onto the stack before proceeding to evaluate e to an abstraction $\lambda e'$. At this point p will be available on the top of the stack and will be 'substituted' into e' by adding it to the current environment. In a similar fashion the stack will hold the case alternatives while a case object is being evaluated, as well as the second argument of `seq` during the evaluation of the first.

The stack may also contain update markers. When the machine arrives at a variable x an update marker for the associated heap pointer $\#p_x$ is pushed onto the stack. The closure at p_x is then evaluated to whnf, after which the marker indicates that the heap should be updated with this result.

3.2 Parallel evaluation

AMPLE uses a model of parallelism that consists of a set of threads which communicate via a common heap. Each thread contains its own control, environment and stack and is labeled with an index t. The transition rules for the appropriate evaluation mode are applied to each thread separately and a given thread becomes inactive when no rule applies. The transition rules are given in the appendix.

3.3 Fully speculative evaluation

In fully speculative mode the evaluation of an application $e_1\, x$ proceeds by evaluating both e_1 and the closure referenced by x in parallel. As per the **fs-app**$_1$ rule, the parent thread t pushes p_x onto it's stack and continues with the evaluation of e_1. A new thread t' is created which commences evaluation of the closure to whnf and will update the heap once finished. The heap element at p_x is overwritten with a blackhole $\bullet$ to indicate that this closure is presently being evaluated. The **pe-var**$_1$ rule, which also evaluates a closure to whnf, exhibits similar behaviour.

If another thread requires the value of a closure that is presently being evaluated then that thread will be *blocked.* Blocking is handled by the **pe-var**$_2$ rule which sets the thread's control to the blocked state $\square$ and appends it's index to the associated blocking queue. When the evaluation of a closure is complete the **pe-var**$_3$ writes it's value back to the heap as well as unblocking any threads that were waiting for it.

One final rule **fs-app**$_3$ is required so that for an application $e_1 x$, if the closure referenced by x is already being evaluated then no new thread is created.

The evaluation of a program in fully speculative mode results in the creation of many fine-grained threads which quickly become blocked. Although impractical as far as a 'real' implementation is concerned, when a program is evaluated in fully speculative mode with the communication latency set to zero it's speedup relative to the sequential mode is constrained only its inherent data dependencies. This is useful for determining an upper limit to the amount of parallelism which can be gained from a particular program.

3.4 Explicitly parallel evaluation

In explicitly parallel mode threads are created with the `par` combinator. The expression e_1 `par` e_2 causes a new thread to be created which evaluates the closure for e_1 in parallel with e_2. In the 'mostly-implicit' parallelism offered by GpH the reduction of e_1 `par` e_2 merely records that e_1 *may* be evaluated in parallel. This operation is called *sparking* and the job of determining when, if ever, to create an actual thread for e_1 is left to the run-time system.

In contrast, the Ample machine *always* creates a new thread. All threads which are not presently blocked are members of the *runnable* pool. For each machine step, the scheduler chooses n threads from this runnable pool – where n represents the number of processors in the machine. These threads form the list of *active* threads, and the state transition rules are applied to these threads only.

Expressions of the form e_1 `par` e_2 are handled by the **par-app**$_1$ rule who's operation is similar to that of **fs-app**$_1$. The **par-app**$_2$ rule performs the same role as **fs-app**$_3$ in preventing a new thread from being created when the associated closure is already being evaluated.

3.5 Thread synchronisation

If one were to implement an abstract machine for single threaded evaluation then the natural way of expressing the reduction rules would be to make use of the pattern matching mechanism provided by a standard functional language. Each rule would be a pattern for a function, probably called `reduce`, which matches on the appropriate machine state and returns an updated state.

```
reduce :: (Control, Env, Stack, Heap)
       -> (Control, Env, Stack, Heap)
```

Unfortunately, this straight forward implementation is not as useful when applied to a machine consisting of a set of threads – each with its own control, environment and stack – which communicate via a common heap or otherwise affect the global machine state. In the definition of the Parallel Lazy Abstract Machine given in [1] the **fs-union** rule is used to combine the effects of all reductions performed during a given computation step. When implementing this rule it is important to ensure that the effect each thread has on the global machine state is not visible to other threads until the beginning of the next step. This behaviour *must* be enforced in order to preserve causality between steps.

To achieve this we must somehow separate the aspects of the reduction which are local to a specific thread from the aspects that are global in nature. The challenge is to maintain the simplicity of the original `reduce` function while avoiding the introduction of low level details that are beyond the scope of an abstract machine. One option that we certainly *don't* want to entertain is to keep a separate instance of the global machine state for each thread and then somehow combine them all at the end of the step.

In separating the local and global aspects of a reduction it is important to ensure that the atomic nature of the original rule is preserved. An example of what would happen if this property were violated can be derived by inspecting the **pe-var**$_1$ and **pe-var**$_2$ rules shown in section 2 of the appendix.

These two rules overlap in all parameters of the local thread state. When a thread arrives at this state the determination of whether it should be reduced via **pe-var1** or **pe-var2** can only be made after inspecting the appropriate element in the heap.

It is critical that other threads do not modify this heap element during the reduction. Consider the following sequence of events,

1. Thread A reads the heap element at p_x, sees a black-hole and determines that another thread is in the process of evaluating that closure.
2. Thread B, which had been evaluating the closure at p_x, finishes its computation and updates the heap. This causes all threads present in the blocking queue at p_x to be unblocked.
3. Thread A, thinking that p_x is still unevaluated, writes a black-hole back into the heap along with the original blocking queue, with its own index at the front.

This behaviour is clearly incorrect. Depending on the specific program, either the closure at p_x will be re-evaluated or thread A will never be unblocked. The root of the problem is that **pe-var**$_1$ and **pe-var**$_2$ are atomic operations and must be treated as such.

Message passing With this in mind, one way of solving the problem would be for the thread to send a message which invokes an atomic operation on the heap. On arriving at a state as per **pe-var**$_1$ / **pe-var**$_2$ the thread would send the following message,

```
if       the heap element at $p_x$ is a closure
then     overwrite it with a black-hole and send back the closure.

else if  the heap element at $p_x$ is a black-hole
then     add the thread index $t$ to the blocking queue and send back
         a token indicating that this was the action taken.

else     no rule applies, send back an error token.
```

The thread would then inspect the return message to determine how to proceed with the reduction.

Although this method ensures that reductions are implemented as atomic operations, the abstract nature of the machine has been spoilt. In the message passing model a specific reduction must be broken down into two distinct stages, one before the heap access and one after. This destroys the simplicity of the original `reduce` function.

Staggered update The approach taken in Ample is to modify the original `reduce` function so that any modifications that would otherwise be made to the global machine state are deferred until all threads have been reduced.

With this approach `reduce` retains its original form, with the addition of a list of *reduction modifications* which describe the effects that the reduction has on the global state.

```
reduce :: (Control, Stack, Env, Heap)
       -> (Control, Stack, Env, Heap, [ReduceMod])
```

Only after `reduce` has been applied to all active threads are the resulting reduction modifications applied to the global state. An exception lies in the `let` rule which must make two separate accesses to the heap – one to reserve heap elements for each of the bindings and one to update the elements with closures for these bindings. This is necessary because the closures for recursive bindings reference each other so their associated heap pointers must be known before they can be constructed. As the mere reservation of heap elements provides no information to other threads this operation can be performed during `reduce`. Updating the elements with closures is still performed using reduction modifications.

This scheme results in the required machine behaviour, provided the reduction modifications are applied in a certain order. Heap updates must be performed first, followed by additions to the blocking queues, followed by unblocking and removal from the blocking queues, followed by the spawning of new threads. These modifications are labeled `Update`, `Block`, `Unblock` and `Spawn` respectively.

3.6 Parameterisation of the abstract machine

Once the `reduce` function has been applied to all active threads the resulting list of reduction modifications is broken down into its component types. Although the `Update` and `Block` modifications must be applied immediately, the `Unblock` and `Spawn` modifications may be written to a queue in order to delay their application. By setting the number of steps that these modifications must spend in the queue before they are applied, a model of communication latency is introduced into the machine.

Greiner and Blelloch note that several parallel functional language implementations reactivate blocked threads sequentially. As a consequence, apparently parallel programs can degrade to effectively sequential performance [3]. By setting the delays for the `Unblock` and `Spawn` queues appropriately, this and other behaviours may be studied.

Ample also allows the number of active threads to be limited to a maximum value. Doing this simulates a machine with a finite number of processors. Different methods for scheduling the pool of runnable threads among these processors may also be studied.

4 Profiling

Step counting Ample keeps track of the number of steps performed *vs* the number of times each reduction rule was applied. This information can be used to determine the speedup of a parallel program compared to its sequential version.

Ample also counts the number of steps that each thread spent blocked as well as the number of steps during which *all* non-idle threads were blocked. This last figure represents time that the machine spent with no work to perform as it was waiting for internal communication to complete. As such, time spent in this state may indicate that the parallelism was too fine-grained compared to the latency of the machine or that the run-time system was over-eager in creating new threads. Figure 4 shows an example where this is the case.

Thread activity A thread activity plot shows the state each thread was in throughout the reduction. In this paper the states are indicated with a thick line, a medium line, a thin line and no line for steps that a thread spent active, runnable but not active, blocked and inactive respectively. Plots generated interactively are shown with the states colour coded.

When a thread finishes a unit of work it transitions from the active to the inactive state. Before doing this it writes its result back to the heap which causes all threads waiting on that result to be to be unblocked. When working with smaller expressions it is possible to use the thread activity plot to see how a given thread unblocks others. After doing this the user can instruct Ample to trace the reduction around that point in order to determine exactly what part of the expression was being evaluated.

Thread count The thread count plot shows how many threads were in each state for each step of the reduction. The thread count plot has the same profile as the thread activity plot but is more useful for gaining insight into exactly how many machine resources were being used.

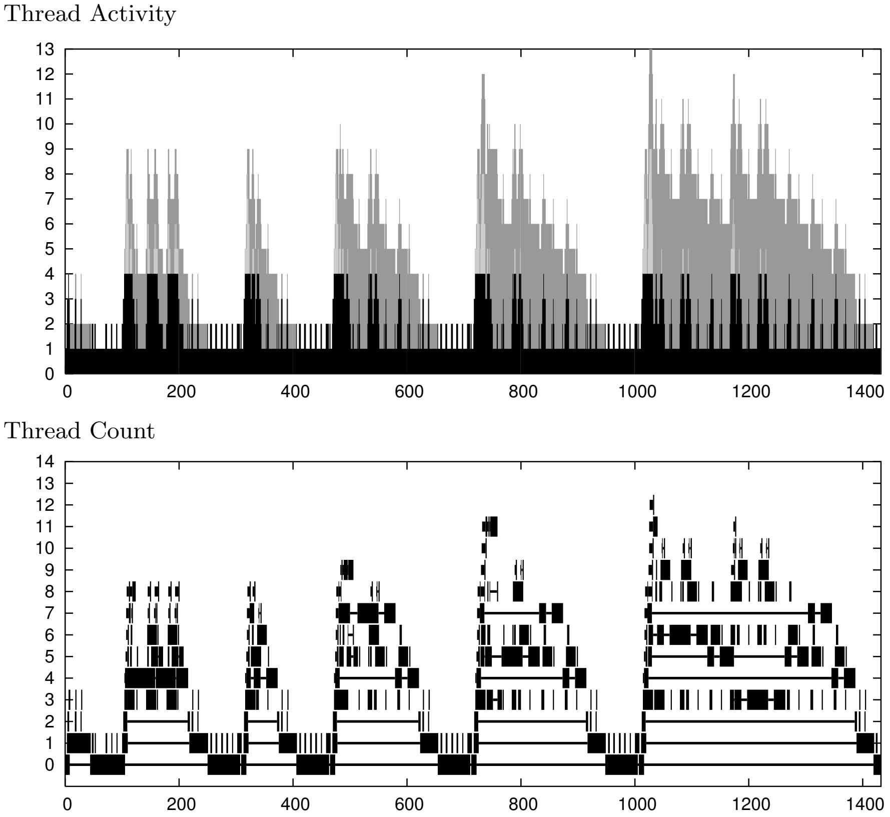
Length of the spawn and unblocking queues The length of the spawn queue corresponds to the number of threads that have been created but are waiting to be added to the runnable pool. The length of the unblock queue corresponds to the number of `Unblock` modifications that have been generated but are waiting to be applied. In a low latency machine each of these plots will appear as a series of impulses, as these modifications are applied soon after they are created. In a high latency machine each element will remain in the queue for longer and new elements will tend to pile on top of old ones.

Heap usage A plot of heap usage and the number of allocations performed at each step can be used as an indication of how much work the machine is doing. It is also possible to instruct Ample perform a garbage collection after each step, producing a plot of the number of active heap elements at each stage of the reduction. This is a good way of testing code transformations designed to save heap space.

5 Examples

5.1 Fully speculative evaluation

Figure 2 shows the machine profile when evaluating the first five prime numbers using the Sieve of Eratosthenes. The expression evaluated was `print (take 5 primes)` where the source for `primes` is shown in figure 3 and `take` is defined in the usual way. For this evaluation the number of active threads was limited to four and the communication latency was set to zero.


Thread Activity

Thread Count

Fig. 2. Profile of the evaluation of `(take 5 primes)` in fully speculative mode

```
primes          = primes'  $ from positiveInts 2;
primes' xx      = case xx of { (x:xs) -> x : primes' (sieve x xs); };

sieve   n xx    =
 case xx of {
        []      -> [];
        (x:xs)  -> if x % n == 0 then     sieve n xs
                   else               x : sieve n xs; };
```

Fig. 3. Ample source for the `primes` function

The evaluation is driven by the `print` function which prints out the elements of the list one at a time. `print` makes use of the `seq` combinator to ensure that characters are output in the correct order. This results in obvious peaks in machine activity, corresponding to the nth prime number being evaluated, interspersed by periods of low activity when the values are printed out.

At the beginning of each iteration many threads are created. Although most of these threads become blocked straight away, the threads that finish their work become inactive and their indices are re-used. The visual activity in rows three and six in the thread count plot are the result of this behaviour. The alternative would be to increment the thread index for each new thread created, in a manner similar to GranSim's 'per-thread activity profile' [4], although this approach would become unwieldly with such a large number of fine grain threads.

Figure 5 shows a tally of how many times each reduction rule was invoked during the evaluation. Somewhat surprisingly, the rule used most frequently is **pe-var3/Cnstr** which updates the heap with a construct. On the other hand, the fact that the vast majority of reductions represent the 'overhead' of lazy evaluation – as opposed to operations on base types – is less surprising.

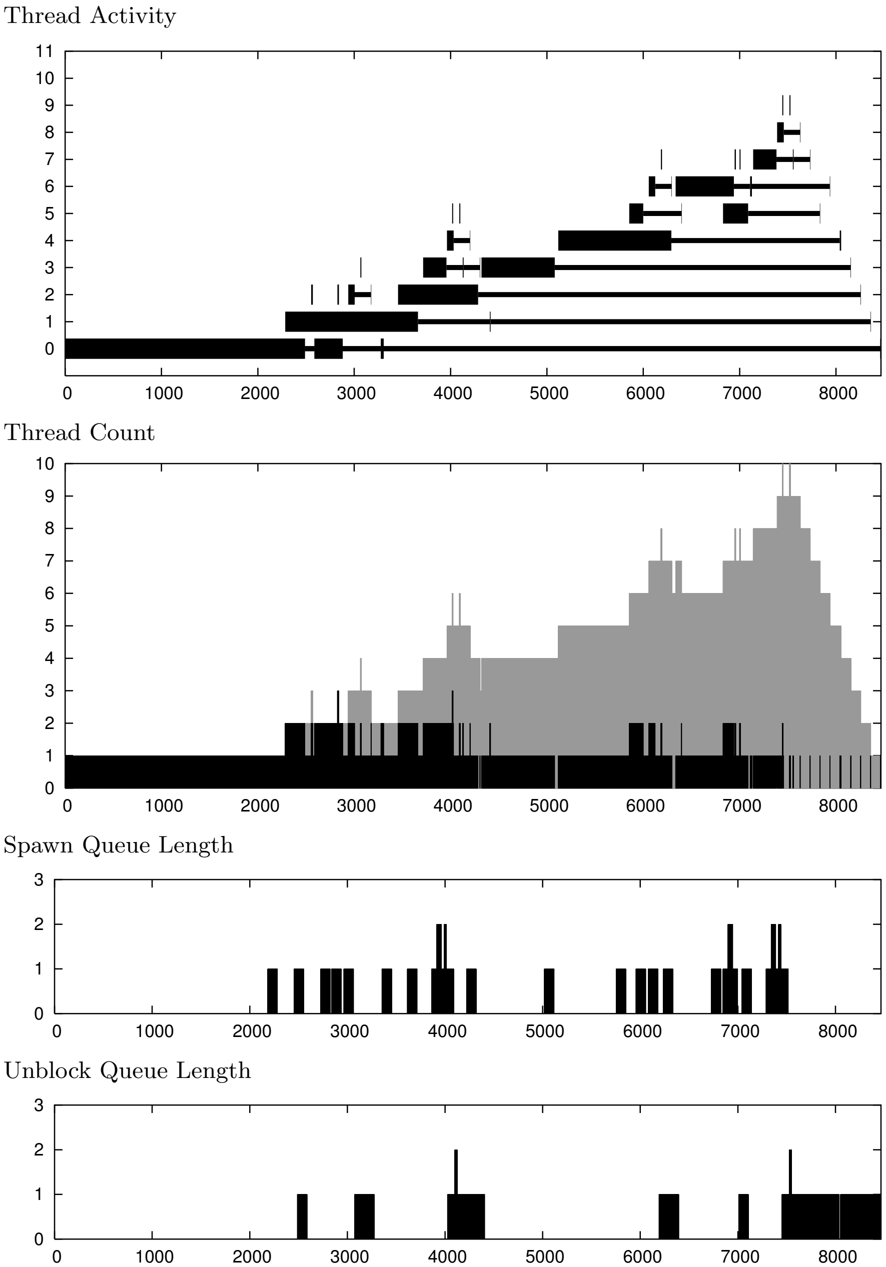
5.2 Explicitly parallel evaluation

Figure 4 shows the machine profile when evaluating an expression that inserts a list of elements into a binary tree and then determines the size of that tree. The expression evaluated was `(parTreeSize tree)` where the source for both `parTreeSize` and `tree` is shown in figure 6. Although the tree is constructed in a sequential manner, `parTreeSize` creates a new thread to determine the number of elements in each branch. In this example 23 elements are inserted into the tree, though they are not all shown in the source. The communication latency has been set to 100 cycles.

As little work is done at the nodes of the tree the threads created for each branch quickly become blocked. From the profile we see that in the final stages of the reduction the number of active threads alternates between one and zero. This corresponds to the period where the sub-counts for each branch of the tree propagate back up to the main thread. As the threads corresponding to branches higher up in the tree have been blocked while waiting for lower threads to complete this information is carried by the machines unblocking mechanism. It takes 100 cycles for each of these values to be communicated and while this communication is taking place there is no other work to be done. This causes the number of active threads to fall to zero.

6 Related Work

The GranSim series of profilers are run-time profilers for Glasgow Parallel Haskell (GpH) [4] [5]. GpH is a mostly-implicit parallel extension of Haskell98 and makes use of the Glasgow Haskell Compiler (GHC) for the front end compilation while providing a new parallel run-time system named GUM [8]. GranSim


Thread Activity

Thread Count

Spawn Queue Length

Unblock Queue Length

Fig. 4. Profile of the evaulation of `(parTreeSize tree)` with communication latency set to 100 cycles

Rule	*n*	*%*
pe-var3/Cnstr	450	0.17
pe-var1	338	0.13
app2	273	0.10
stop	272	0.10
fs-app1	271	0.10
case1	233	0.09
case2/match	232	0.09
pe-var3/Lambda	159	0.06
let	139	0.05
pe-var2	68	0.03
const/int	47	0.02
prim/intEq	25	0.01
seqPush	19	0.01
seqEval/Cnstr	19	0.01
prim/intMod	16	0.01
prim/intAdd	11	
const/char	10	
print/char	10	
prim/intSub	7	
print/int	5	
pe-app3	2	
case2/default	1	

Fig. 5. Tally of the number of times each reduction rule was taken during the evaluation of `(take 5 primes)`

```
treeInsert tree iKeyVal =
 case iKeyVal of {
    (iKey, iVal) ->
     case tree of {
        []                      -> (iKey, iVal, [], []);
        (key, val, left, right) ->
            if      iKey == key then KeyExists
            else if iKey < key  then
                 (key, val, treeInsert left iKeyVal, right)
            else (key, val, left, treeInsert right iKeyVal); }; };

parTreeSize tree =
 case tree of {
    []                      -> 0;
    (key, val, left, right) ->
     let {
        sizeLeft    = parTreeSize left;
        sizeRight   = parTreeSize right;
     } in
        sizeLeft par (sizeRight par (1 + sizeLeft + sizeRight)); };

elements    = ["perch", "barb", "tetra", "discus"   ... ]
elementIxs  = [5,       10,      4,      3          ... ]

tree = foldl treeInsert [] (zip elements elementIxs)
```

Fig. 6. Ample source for the `parTreeSize` and `tree` functions

comes in a number of flavours including GranSim-Lite which can be used to simulate an idealised machine with infinite processors and no communication cost.

The primary benefit that Ample has over GranSim and other such systems is that being an abstract machine, Ample it is much easier to modify and reason about. On the other hand, low level aspects such as graph packing techniques and caching strategies are beyond the scope of an abstract machine and are best left for such a system.

7 Conclusion

When working with an abstract machine, design ideas can be experimented with without the overhead of building a complete language implementation. We have seen that the careful analysis of the state transition rules given in [1] and [6] has resulted in a modular system which provides useful information for small functions. Future work on Ample would allow the reduction rules take different numbers of steps so that machine behaviour closer to a native implementation could be simulated. Ample could also serve as the basis for an abstract machine for distributed processing. In such a machine, processing elements would consist of separate instances of the Ample abstract machine, connected by their existing IO mechanism.

References

1. Clem Baker-Finch. Parallel lazy abstract machines. In *Proceedings of the First Scottish Functional Programming Workshop*, pages 33–42, 1999.
2. Clem Baker-Finch, David King, and Phil Trinder. An operational semantics for parallel lazy evaluation. In *ACM-SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 162–173. ACM, 2000.
3. John Greiner and Guy Blelloch. A provably time-efficient parallel implementation of full speculation. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 309–321, January 1996.
4. King D. J., Hall J. G., and Trinder P. W. A strategic profiler for Glasgow Parallel Haskell (GpH). In *Proceedings of the 10th Int. Workshop on Implementation of Functional Languages*, pages 90–104, September 1998.
5. Hans-Wolfgang Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.
6. Peter Sestoft. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264, May 1997.
7. Philip W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.
8. Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones. GUM: a portable implementation of Haskell. In *International Workshop on the Implementation of Functional Languages*, Bastad, Sweden, September 1995.

A AMPLE Reduction Rules

A.1 Rules for single threaded evaluation

app$_1$

$$\begin{array}{lllrrl}
 & (& \text{ExpVar e x}, & [..., \text{p}_x, ...], & \text{S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}, & [..., \text{p}_x, ...], & \text{p}_x\text{: S} \ ) & \text{H}
\end{array}$$

app$_2$

$$\begin{array}{lllrrl}
 & (& \text{Lambda e}, & \text{E}, & \text{p}_x\text{: S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}, & \text{p}_x\text{: E}, & \text{S} \ ) & \text{H}
\end{array}$$

var$_1$ (enter)

$$\begin{array}{lllrrl}
 & (& \text{Var x}, & [..., \text{p}_x, ...], & \text{S} \ ) & \text{H} \ [\text{p}_x \rightarrow \text{(e', E')} \ ] \\
\Rightarrow & (& \text{e}, & \text{E'}, & \#\text{p}_x\text{: S} \ ) & \text{H}
\end{array}$$

var$_2$ (update)

$$\begin{array}{lllrrl}
 & (& \textit{whnf}, & \text{E}, & \#\text{p : S} \ ) & \text{H} \\
\Rightarrow & (& \textit{whnf}, & \text{E}, & \text{S} \ ) & \text{H} \ [\text{p} \rightarrow (\textit{whnf}, \text{E'})]
\end{array}$$

$$\begin{array}{lllll}
\text{where} & \text{E} & = [\text{p}_0, ... \ \text{p}_a, ... \ \text{p}_n] & & \\
 & \text{E'} & = [\text{p}_0, \ldots \text{p}_a \ldots \text{p}_n], & \text{when } \textit{whnf} & = \text{Lambda e} \\
 & & \mid \ [\text{p}_0, \ldots \text{p}_a], & \text{when } \textit{whnf} & = \text{Cnstr m a}
\end{array}$$

let

$$\begin{array}{lllrrl}
 & (& \text{Let } [(\text{b}_i, \text{t}_i)] \ \text{e}_0 \ \text{t}_0, & \text{E}, & \text{S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}_0, & \text{E'} \mid \text{t}_0, & \text{S} \ ) & \text{H} \ [\text{p}_i \rightarrow \text{b}_i, \text{E'} \mid \text{t}_i \ ]
\end{array}$$

where $\text{E'} = [\text{p}_n, ... \ \text{p}_1] \ {+}{+} \ \text{E}$

case$_1$

$$\begin{array}{lllrrl}
 & (& \text{Case (e, t) alts}, & \text{E}, & \text{S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}, & \text{E}, & (\text{alts, E} \mid \text{t}) : \text{S} \ ) & \text{H}
\end{array}$$

case$_2$

$$\begin{array}{lllrrl}
 & (& \text{Cnstr name}_i \ \text{a}_i, & [\text{p}_1, ... \ \text{p}_a, ... \ \text{p}_n], & (\text{alts, E}) : \text{S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}_i, & ([\text{p}_i, ... \ \text{p}_a] \ {+}{+} \ \text{E}) \mid \text{t}_i, & \text{S} \ ) & \text{H}
\end{array}$$

seqPush

$$\begin{array}{lllrrl}
 & (& \text{Seq e}_1 \ \text{e}_2, & \text{E}, & \text{S} \ ) & \text{H} \\
\Rightarrow & (& \text{e}_1, & \text{E}, & (\text{SSeq e}_2 \ \text{E}) : \text{S} \ ) & \text{H}
\end{array}$$

seqEval

$$\begin{array}{llrrl}
 & (\ \textit{whnf}, & \text{E}, & (\text{SSeq e}_2\ \text{E'}) : \text{S}\ ) & \text{H} \\
\Rightarrow & (\ \text{e}_2, & \text{E'}, & \text{S}\ ) & \text{H}
\end{array}$$

constant{Int|Char}

$$\begin{array}{llrrl}
 & (\ \{\text{Int}|\text{Char}\}\ \text{v}, & \text{E}, & \text{S}\ ) & \text{H} \\
\Rightarrow & (\ \{\text{RetInt}|\text{RetChar}\}, & \{\text{TagInt}|\text{TagChar}\} : \text{v} : \text{E}, & \text{S}\ ) & \text{H}
\end{array}$$

print{Int|Char}

$$\begin{array}{llrrl}
 & (\ \text{PrimFunc "print"}, & \{\text{TagInt}|\text{TagChar}\} : \text{v} : \text{E}, & \text{S}\ ) & \text{H} \\
\Rightarrow & (\ \text{Cnstr "Done" 0}, & \text{E}, & \text{S}\ ) & \text{H}
\end{array}$$

prim_intAdd

$$\begin{array}{llrrl}
 & (\ \text{PrimFunc IntAdd}, & \text{TagInt : n1 : TagInt : n2 : E}, & \text{S}\ ) & \text{H} \\
\Rightarrow & (\ \text{RetInt}, & \text{TagInt : (n1 + n2) : E}, & \text{S}\ ) & \text{H}
\end{array}$$

prim_intEq

$$\begin{array}{llrrl}
 & (\ \text{PrimFunc IntEq}, & \text{TagInt : n1 : TagInt : n2 : E}, & \text{S}\ ) & \text{H} \\
\Rightarrow & (\ \text{Cnstr}\ \textit{name}\ \text{0}, & \text{E}, & \text{S}\ ) & \text{H}
\end{array}$$

where $\textit{name}$ = "True" | "False"

A.2 Modifications for parallel evaluation

pe-var$_1$ (enter)

$$\begin{array}{llrrl}
 & (\ \text{Var x}, & [...,\ \text{p}_x,\ ...], & \text{S}\ ) & \text{H}\ [\text{p}_x \rightarrow (\text{e, E})] \\
\Rightarrow & (\ \text{e}, & \text{E}, & \#\text{p}_x\text{: S}\ ) & \text{H}\ [\text{p}_x \rightarrow \bullet_{[]}]
\end{array}$$

pe-var$_2$ (block)

$$\begin{array}{llrrl}
 & (\ \text{Var x}, & [...,\ \text{p}_x,\ ...], & \text{S}\ ) & \text{H}\ [\text{p}_x \rightarrow \bullet_{ts}] \\
\Rightarrow & (\ \Box, & [\,], & \text{S}\ ) & \text{H}\ [\text{p}_x \rightarrow \bullet_{t:ts}]
\end{array}$$

pe-var$_3$ (unblock)

$$\begin{array}{lllrrl}
 & (& \textit{whnf}, & [...\ \text{p}_x,\ ...], & \#\text{p}_x\text{: S}\ ) & \text{H}\ [\ \text{p}_x \rightarrow \bullet_{[t1,\ldots tn]}\ ] \\
 & \{(& \Box, & [\,], & \text{S}_i\ )_i\ \} & \\
\Rightarrow & (& \textit{whnf}, & [...\ \text{p}_x,\ ...], & \text{S}\ ) & \text{H}\ [\ \text{p}_x \rightarrow (\textit{whnf},\ [...\ \text{p}_x,\ ...])_{[]}\ ] \\
 & \{(& \textit{whnf}, & [...\ \text{p}_x,\ ...], & \text{S}_i\ )_i\ \} &
\end{array}$$

where $i = [\text{t}_1,\ ...\ \text{t}_n]$

A.3 Modifications for fully speculative evaulation

fs-app$_1$ (spawn)

$$\begin{array}{lllrl}
 & (\ \mathrm{ExpVar}\ \mathrm{e}_1\ \mathrm{x}, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{S}\ )_{t} & \mathrm{H}\ [\mathrm{p}_x \rightarrow (\mathrm{e}_2, \mathrm{E}_2)\ ] \\
\Rightarrow & (\ \mathrm{e}_1, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{p}_x\mathrm{:}\ \mathrm{S}\ )_{t} & \mathrm{H}\ [\mathrm{p}_x \rightarrow \bullet_{[]}\ ] \\
 & (\ \mathrm{e}_2, & \mathrm{E}_2, & [\ \#\mathrm{p}_x]\ )_{t'} &
\end{array}$$

where t' is fresh

fs-app$_3$

$$\begin{array}{lllrl}
 & (\ \mathrm{ExpVar}\ \mathrm{e}\ \mathrm{x}, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{S}\ ) & \mathrm{H}\ [\mathrm{p}_x \rightarrow \bullet_{ts}] \\
\Rightarrow & (\ \mathrm{e}, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{p}_x\mathrm{:}\ \mathrm{S}\ ) & \mathrm{H}
\end{array}$$

A.4 Modifications for `par-seq` Evaluation

par-app$_1$ (spawn)

$$\begin{array}{lllrl}
 & (\ \mathrm{Par}\ \mathrm{x}\ \mathrm{e}_1, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{S}\ )_{t} & \mathrm{H}\ [\mathrm{p}_x \rightarrow (\mathrm{e}_2, \mathrm{E}_2)\ ] \\
\Rightarrow & (\ \mathrm{e}_1, & [\ldots, \mathrm{p}_x, \ldots], & \mathrm{p}_x\mathrm{:}\ \mathrm{S}\ )_{t} & \mathrm{H}\ [\mathrm{p}_x \rightarrow \bullet_{[]}\ ] \\
 & (\ \mathrm{e}_2, & \mathrm{E}_2, & [\ \#\mathrm{p}_x]\ )_{t'} &
\end{array}$$

where t' is fresh

par-app$_2$

$$\begin{array}{lllrl}
 & (\ \mathrm{Par}\ \mathrm{x}\ \mathrm{e}_1, & \mathrm{E}, & \mathrm{S}\ ) & \mathrm{H}\ [\mathrm{p}_x \rightarrow \bullet_{[]}] \\
\Rightarrow & (\ \mathrm{e}_1, & \mathrm{E}, & \mathrm{S}\ ) & \mathrm{H}
\end{array}$$