Low-Pain, High-Gain Multicore Programming in Haskell

Coordinating Irregular Symbolic Computations on MultiCore Architectures

A. Al Zain
P. Trinder M. Aswad
G. Michaelson
Computer Science Department, MACS,
Heriot-Watt University, Edinburgh, UK
{ceeatia,trinder,mka19,greg}@macs.hw.ac.uk

K. Hammond
School of Computer Science,
University of St Andrews,
St Andrews, UK
kh@cs.st-and.ac.uk

J. Berthold
FB Mathematik und Informatik,
Philipps-Universität, Marburg,
D-35032 Marburg, Germany
berthold@mathematik.uni-marburg.de

Abstract

With the emergence of commodity multicore architectures, exploiting tightly-coupled parallelism has become increasingly important. Functional programming languages, such as Haskell, are, in principle, well placed to take advantage of this trend, offering the ability to easily identify large amounts of fine-grained parallelism. Unfortunately, obtaining real performance benefits has often proved hard to realise in practice.

This paper reports on a new approach using middleware that has been constructed using the Eden parallel dialect of Haskell. Our approach is "low pain" in the sense that the programmer constructs a parallel program by inserting a small number of higher-order *algorithmic skeletons* at key points in the program. It is "high gain" in the sense that we are able to get good parallel speedups. Our approach is unusual in that we do not attempt to use shared memory directly, but rather coordinate parallel computations using a message-passing implementation. This approach has a number of advantages. Firstly, coordination, i.e. locking and communication, is both confined to limited shared memory areas, essentially the communication buffers, and is also isolated within well-understood libraries. Secondly, the coarse thread granularity that we obtain reduces coordination overheads, so locks are normally needed only on (relatively large) messages, and not on individual data items, as is often the case for simple shared-memory implementations. Finally, cache coherency requirements are reduced since individual tasks do not share caches, and can garbage collect independently.

We report results for two representative computational algebra problems. Computational algebra is a challenging application area that has not been widely studied in the general parallelism community. Computational algebra applications have high computational demands, and are, in principle, often suitable for parallel execution, but usually display a high degree of irregularity in terms of both task and data structure. This makes it difficult to construct parallel applications that perform well in practice. Using our system, we are able to obtain both extremely good processor utilisation (97%) and very good absolute speedups (up to 7.7) on an eight-core machine.

DAMP'09, January 20, 2009, Savannah, Georgia, USA.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: [Parallel processors]; D.1.3 [*Parallel Programming*]; D.3.2 [*Language Classification*]: Concurrent, distributed, and parallel languages; G.4 [*Mathematical Software*]; I.1 [*Symbolic and Algebraic Manipulation*]

General Terms Experimentation, Measurement, Performance.

Keywords Haskell, Eden, Multicore Parallelism, Algorithmic Skeletons, Computational Algebra, GAP.

1. Introduction

Multicore architectures are becoming increasingly common, and are projected to become still more pervasive. However, effectively utilising the tightly-coupled parallelism provided by these architectures is proving to be a significant challenge (34). This paper reports preliminary results from a new approach to multicore programming using Haskell-based parallel algorithmic skeletons, that exploits an underlying message-passing implementation to map independent, communicating threads onto multiple cores. We demonstrate the effectiveness of the approach by showing how it can be used to effectively coordinate a number of sequential computational components, written in the GAP computational algebra system (30), on a multicore architecture.

Unlike much other work on multicore parallelism, we concentrate exclusively on coordinating computational components using a high-level declarative parallel programming model (algorithmic skeletons) that exposes ultra-lightweight threads. Our approach exploits sophisticated parallel middleware that dynamically creates and controls processes and threads, that automatically manages communication and synchronisation, and that independently performs garbage collection of data within each parallel process. In contrast to the lightweight communication and synchronisation used by most parallel languages targeting shared-memory architectures[1], Eden uses relatively heavyweight message-passing. The specific research contributions of this paper are:

i) we advocate the use of a semi-explicit parallel functional programming model, using high-level parallel skeletons, for multicore architectures;

ii) we show how such a programming model, realised in the Eden parallel dialect of Haskell (38), can coordinate sequential computations written in GAP on a multicore architecture;

[1] Erlang (6; 23) is the notable exception.

iii) we demonstrate the first multicore performance results for such a system, showing that it achieves processor utilization up to 97% and speedups of up to 7.7 on an eight-core machine;

iv) we investigate the minimum thread granularity where our relatively heavyweight parallel programming model can deliver good performance on multicore architectures; and

v) we demonstrate, by example, that using our high-level parallel programming model, minimal effort is required to gain acceptable parallelism on multicore architectures for suitably high-grained symbolic computations.

2. Overview

As mentioned above, one of the most interesting aspects of our approach is that we use parallel Haskell as a *coordination language* (31). As shown in Figure 1, rather than (re)-implementing the algorithms that we wish to parallelise using parallel Haskell, we use the SymGrid-Par middleware (written in parallel Haskell) to *coordinate* essentially-unchanged *sequential* procedures written in some computational language (here, GAP, but, in principle, any computational language, including C, C++ or Java), into a coherent parallel application. In this way, we obtain the key advantages of declarative parallelism, namely the simple specification and correct implementation of parallelism (47), while still supporting existing sequential application code and libraries. Existing end-users will thus be easily able to migrate to multicore architectures, without significant reimplementation or language design effort on the part of the sequential language implementor.

This is an attractive proposition since it, at a stroke, almost eliminates the barrier to entry to multicore programming for many users and implementors (it is necessary to support data marshalling/unmarshalling, and remote procedure call over some simple intra-core communication mechanism, but this can be as simple as a function call, or pipe). The primary cost lies in potentially high (and, on a multicore, apparently unnecessary) data marshalling and unmarshalling costs, plus overheads associated with calling message-passing libraries. Somewhat surprisingly, the results presented in this paper, and elsewhere (9), suggest that these overheads can be negligible, even for quite fine-grained parallel applications, and that multicore systems provide effective latency-hiding mechanisms for message construction. In fact, we have shown elsewhere (9), that our implementation is highly competitive with, and,

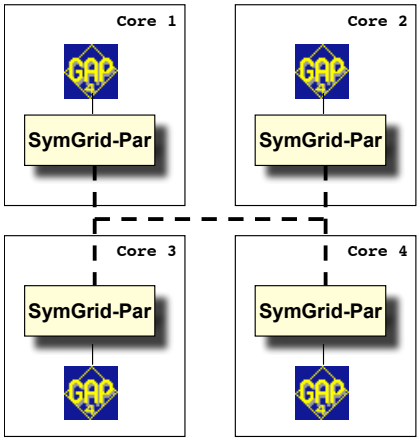

Figure 1. Using Haskell as a Coordination Language on a Multicore Platform

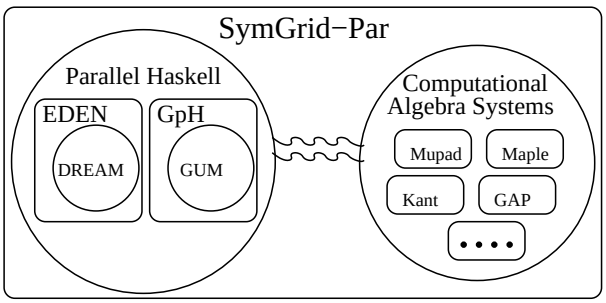

Figure 2. Structure of the `SymGrid-Par` Middleware

in some cases, superior to, an implementation of a very similar parallel Haskell using a shared heap. Although we are in the process of producing a shared-heap implementation of the Eden language constructs we describe here, we do not yet, however, have directly comparable results that we can report. We therefore *conjecture*, based on our other results using a shared-heap implementation, that there will usually be little performance difference between the two technologies on small numbers of cores, that the message-passing implementation will show significant advantages for most applications beyond 4-8 cores, and that there may be significant advantages for some communication-intensive applications even on 2-4 cores.

2.1 The GAP Computational Algebra System

Computational algebra has played an important role in a number of notable mathematical developments, for example in the classification of finite simple groups. It is essential in several areas of mathematics which apply to computer science, such as formal languages, coding theory, or cryptography. Computational algebra applications are typically characterised by complex and expensive computations that would benefit from parallel computation, but which may exhibit a high degree of irregularity in terms of both data- and computational-structures. Application developers are typically mathematicians or other domain experts, who may not possess parallel expertise or have the time/inclination to learn complicated parallel systems interfaces. Our work aims to support this application irregularity in a seamless and transparent fashion, by providing *easy-to-use* coordination middleware (`SymGrid-Par`) that supports dynamic task allocation, load re-balancing and task migration GAP (30) is a free-to-use, open source system for computational discrete algebra, which focuses on computational group theory. It provides a high-level domain-specific programming language, a library of algebraic functions, and libraries of common algebraic objects. GAP is used in research and teaching for studying groups and their representations, rings, vector spaces, algebras, and combinatorial structures.

2.2 The `SymGrid-Par` Middleware

We have built our implementation on the `SymGrid-Par` middleware (2), whose primary purpose is to orchestrate sequential computational algebra components into a (possibly parallel) Grid-enabled application. In the full `SymGrid-Par` design, components communicate using the **OpenMath** data-exchange protocol (41), an XML-based data description format, designed specifically to represent computational mathematical objects which may be distributed across a wide-area, heterogeneous computational Grid. We have adapted `SymGrid-Par` to our multicore test environment in order to explore whether we can obtain good performance for parallel symbolic computations on a multicore system. In this paper, we will not discuss the wider capabilities of `SymGrid-Par` on more general computational Grids.

SymGrid-Par (Figure 2) is built around parallel implementations of GUM (49; 3) (the runtime implementation of Glasgow Parallel Haskell (GPH) (50)) and DREAM (the runtime implementation of Eden (38)). Both GPH and Eden are well-established *semi-explicit* parallel extensions to Haskell. These two Haskell dialects provide various high-level parallelism services including support for ultra-light-weight threads, virtual shared-memory management, scheduling support, automatic thread placement, automatic datatype-specific marshalling/unmarshalling, explicit process control, implicit communication, load-based thread throttling, and thread migration. As will be illustrated later, SymGrid-Par thus provides a flexible, adaptive environment for managing parallelism at various degrees of granularity.

SymGrid-Par exploits the capabilities of the Eden and GPH systems by layering a simple API over their basic functionality, that can be exploited by various computational algebra systems. In this paper, we consider only the interfaces to/from the GAP system (30). By using SymGrid-Par, we achieve a clear separation of concerns: the parallel Haskell systems deal with issues of thread creation/coordination and orchestrate the GAP engines to work on the application as a whole; while each instance of the GAP engine deals solely with the execution of individual GAP computations.

2.3 Linking GAP and SymGrid-Par

We exploit a set of interfaces to link Haskell with a number of different computational algebra systems. Each *processing element* (PE) is an instance of a Haskell graph reduction engine interfaces to an instance of a computational algebra system engine, sending commands to the engine to execute symbolic computations, and receiving the results of running those computations, using a well-defined communication protocol based on the OpenMath standard for transmitting mathematical data. The main Haskell functions we provide to interface to GAP are:

```
gapEval        :: String -> [GAPObject] ->  GAPObject
gapEvalN       :: String -> [GAPObject] -> [GAPObject]
string2GAPExpr :: String    -> GAPObject
gapExpr2String :: GAPObject -> String
```

Here, gapEval and gapEvalN allow Haskell programs to invoke GAP functions by giving a function name (as a String) plus a list of function parameters of type GAPObject; gapEvalN is used to invoke GAP functions that return more than one object; and string2GAPExpr/gapExpr2String respectively convert GAP objects to/from Haskell Strings, from which they may be converted to/from the appropriate internal data formats using read/show. Similar operations are provided for each of the other computational algebra systems that we interface to.

2.4 Parallel Skeletons for Symbolic Computing

We have implemented a set of parallel skeletons for symbolic computing using Eden. These skeletons can be called directly from within the computational steering interface for our computational algebra target systems. The only difference seen by the user of the computational algebra system is in the use of such a skeleton: the SymGrid-Par implementation and automatic coordination is completely transparent to the end-user, except in providing a (hopefully good) parallelisation of their program. Many of the skeletons we have used so far are fairly standard (17; 18), and are based on commonly-used sequential higher-order functions.

```
parMap::       (a->b) ->                  [a] -> [b]
parZipWith::   (a->b->c) ->[a] ->         [b] -> [c]
parReduce::    (a->b->b) -> b ->          [a] ->  b
parMapReduce:: (c->[(d,a)]) -> (d->[a]->b) -> [c]->[(d,b)]
masterWorker:: (a->([a],b))-> [a] -> [b]
```

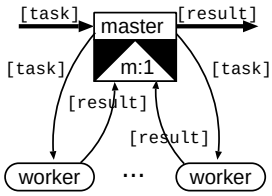

Figure 3. The masterWorker computation scheme.

So, for example, parMap is semantically equivalent to a conventional map, but differs operationally in applying its function argument to each element of the list argument *in parallel*. Similarly, parZipWith is a parallel version of the standard zipWith function; parReduce is a parallel variant of a standard fold(r) function; parMapReduce combines elements of parMap and parReduce, first generating a list of key-value pairs from every input item (in parallel), before reducing each set of values for one key across these intermediate results. The masterWorker skeleton generates a set of worker processes to apply the given function parameter in parallel to a set of input tasks (Figure 3). A master process distributes the tasks to worker processes on demand, including any new tasks that are generated by the workers when producing output. This implementation of the masterWorker skeleton provides dynamic load-balancing between workers, since newly created tasks will be automatically distributed to idle worker processes *on demand*. In addition to its obvious uses, the masterWorker skeleton can also be used to implement parallel backtracking and branch-and-bound algorithms by exploiting the ability of worker processes to dynamically produce and enqueue new tasks (8). Finally, our parMapReduce skeleton is similar to the Map-Reduce "programming model" that has recently been advocated by, *inter-alia*, Google (21; 36). In addition to these standard skeletons, we have also identified several computational patterns that are commonly found in symbolic computing systems. We intend to produce algorithmic skeletons to capture these patterns in the near future and to deploy these in suitable demonstrator applications taken from the domain of computational algebra.

3. Implementation using the Eden Parallel Haskell Dialect

The SymGrid-Par interfaces and skeletons described above have been implemented using Eden (10), a parallel Haskell extension for distributed memory systems that provides explicit control over process creation. Eden is implemented as an extension of the standard Glasgow Haskell compiler, GHC (26; 44). It comprises a few changes to the GHC front-end to handle parallel process creation and other Eden language extensions, plus major modifications to the GHC runtime environment. When run in parallel, Eden starts multiple instances of the sequential GHC runtime system, each acting as a PE. These instances communicate with each other both for synchronisation, and to exchange data. Typically, there will be one such instance per physical core in a multicore system. The implementation of the Eden communication system uses standard message-passing libraries – currently, either PVM or MPI because of their widespread availability and high portability.

3.1 The Eden Language and the SymGrid-Par Programming Model

Eden allows *process abstractions* to be defined using the process function. Processes created in this way can then be *instantiated* (i.e. executed) on remote processors using the (#) operator.

```
process :: (Trans a, Trans b) => (a -> b) -> Process a b
( # )   :: (Trans a, Trans b) => Process a b -> a -> b
```

Evaluating (**process** `f) # arg` for a given function `f` leads to the creation of a new process on a remote PE. This process evaluates the application of function `f` to its single argument `arg`. The argument is first evaluated to normal form on the original PE and then sent to the new process through a communication channel. Both the argument and the result are defined to be members of the Haskell class of transmissible data objects, `Trans`, a class that excludes function objects, and other data objects that may contain functions. Processes are encapsulated units of computation which communicate their inputs and results via *channels*. If the input or output of a process is a tuple, each component of the tuple will be evaluated and communicated by its own independent thread. Lists will be communicated element by element, values of other types will be communicated in single messages. All communication between processes is managed automatically by the system and is completely hidden from the programmer. This mechanism makes it possible to avoid sharing data through demands on an implicitly shared heap (as in done in GpH, for example): all the data that is needed by a process is either communicated through the explicit argument, or is evaluated by the process as part of its normal execution process.

Since Eden is a purely functional language, parallel skeletons can easily be expressed using higher-order functions. The programming model that is visible to the `SymGrid-Par` user is therefore not Eden, but is rather the user-level interface to the library of predefined skeletons. By using the skeleton programming model, we have simplified high-level parallel programming, eliminating the need to consider when messages are passed, how data is packed into buffers etc. For example, `parReduce` can be implemented as shown below.

```
parReduce f neutral list = foldr f neutral subRs
   where subRs = spawn (process (foldr f neutral)) subls
         subls = splitIntoN noPE list

spawn :: Process a b -> [a] -> [b] -- kicks off processes
spawn p inputs = ...               -- for whole input set
```

3.2 The Parallel Implementation Model

At first glance, it may seem somewhat curious to map a distributed memory programming model with a message-passing implementation onto a multicore machine, where data may be exchanged between cores simply using shared memory. We believe, however, that in addition to the expediency of exploiting an existing message-passing implementation, combining Eden with a library of predefined skeletons yields a good match both to the programmer requirements, and to the capabilities of current multicore machines.

As described above, Eden processes never share heap and communicate entirely by message-passing. This allows us to construct *completely independent sub-heaps* for each parallel process. The Eden processes that are being executed will contain linkages to other sub-heaps via implicit Eden communication channels. These channels have the property that all communicated data is always fully evaluated. In a shared-memory system, this has significant advantages for cache coherency, since no heap needs to be shared. This kind of heap separation also *completely eliminates the need for any locking at the thread level*: only message buffers need to be locked; this can be done on a point-to-point basis (so eliminating a global hotspot); and locks are only obtained when data is actually required (demand-driven locking). We regard this as an important contribution: in many parallel (multicore) systems, locking is a major cost and reducing locking is therefore potentially highly significant.

Our approach also allows completely independent garbage collection for each process. This eliminates a global pause during execution, both improving overall runtimes, and reducing visible delays for interactive/reactive systems. We anticipate that, using our system, application performance will scale well on future systems, which are likely to comprise considerably more cores per chip, and which may use hierarchical cache architectures. When scaling up, shared-memory models typically run into major performance problems on such architectures due to locking and cache coherence issues (48).

4. Parallel Symbolic Computation Exemplars

In this paper, we consider two representative computational algebra problems, *liouville* and *smallGroup*. These problems have been chosen since they are typical of a wide variety of real computational algebra problem, possessing interesting computational structures that may be amenable to parallelisation, while also processing large amounts of structured data. Moreover, both problems can be expressed in a simple and easily understood way. Generally, we expect the Haskell components to be provided by the systems programmer (usually in the form of skeletons that are called from the GAP code, or from the computational steering interface), and only the GAP code to be written by the applications programmer or end-user. The details of the implementation that we have given above, and the Haskell code that we show below, should not be given to or written by the typical applications programmer.

4.1 The Summatory Liouville function, *liouville*

For an integer n, the *Liouville* function $\lambda(n)$ is equal to $(-1)^{r(n)}$, where $r(n)$ is the number of prime factors of n, counted according to their multiplicity, with $r(1) = 0$. The *summatory Liouville's function*, $L(x)$, is the sum of values of *Liouville*(n) for all n from $[1..x]$. In Haskell, the coordination part of the kernel of the parallel *liouville* function is[2]:

```
l :: Integer -> Integer -> Int -> [(Integer,Integer)]
l lower upper c = sumL (myMakeList c lower upper)

sumL :: [(Integer,Integer)] -> Int -> [(Integer,Integer)]
sumL mylist c = mySum ((masterWorker liouville) mylist)

liouville :: (Integer,Integer) ->
   ((Integer, Integer),(Integer,Integer))
liouville  (lower,upper) =
  let
   l = map gapObject2Integer (gapEvalN "gapLiouville"
       [integer2GapObject lower,integer2GapObject upper])
  in ((head l, last l), (lower,upper))
```

and the GAP computation part (defined by a GAP end-user) is:

```
LiouvilleFunction:=function( n )
  if n=1 then return 1;
  elif Length(FactorsInt(n)) mod 2=0 then return 1;
  else return -1;
  fi;
end;

gapLiouville := function( n, x )
    local total, max, s, list;
    s    := LiouvilleFunction(n);
    total:= s;  max:= s; n:= n+1;
    while n <= x do
        s := LiouvilleFunction(n); total := total + s;
        if (max < total) then max:= total; fi;
        n := n+1;
    od;
```

[2] The full definition may be found at `http://www.macs.hw.ac.uk/~ceeatia/liouville.hs`

```
    list := [total, max]; return list;
end;
```

The implementation splits the whole sum into partial sums to be computed in parallel. `myMakeList` generates a special list which contains the boundaries of each partial sum. `sumL` relies on the `masterWorker` skeleton to apply the `liouville` function to each interval in parallel. The Haskell `liouville` functions calls the GAP function `gapLiouville` to calculate the partial sum for the given intervals. This function returns the interval boundary, the partial sum and the highest sum in the interval. `mySum` calculates the total sum for all partial intervals. For each interval, it also adds the total sum of the prior intervals to the highest sum to determine whether there is a positive sum from this interval. Finally, `mySum` returns the total sum plus indications for those intervals that have been determined to have a positive sum.

4.2 Small Finite Group Search, *smallGroup*

The *smallGroup* program searches for finite *groups* that have some given property, in this case, that the average order of the elements is an integer. The order of the groups must be no greater than a given constant, n, The kernel of the *smallGroup* program is divided into a coordination part in Haskell[3]:

```
smGrpSearch :: Int -> Int -> [(Int,Int)]
smGrpSearch lo hi = concat(map(ifmatch) (predSmGrp [lo..hi]))

predSmGrp :: (Int,Int) ->(Int,Int,Bool)
predSmGrp (i,n) = (i,n,(gapObject2String (gapEval
   "IntAvgOrder" [int2GapObject n, int2GapObject i])) == ''true'')

ifmatch :: ((Int,Int) -> (Int,Int,Bool)) -> Int -> [(Int, Int)]
ifmatch predSmGrp n=
   [(i,n) |   (i,n,b) <-(masterWorker predSmGrp
   [(i,n) | i<- [1 .. nrSmGrps n]]),b]

nrSmGrps :: Int -> Int
nrSmGrps n=gapObject2Int(gapEval "NrSmallGroups"[int2GapObject n])
```

plus a computational part written in GAP:

```
IntAvgOrder := function(n,i)
   local cc, sum, c, g;
  sum:=0; g:=SmallGroup(n,i); cc:= ConjugacyClasses(g);
  for c in cc do
    sum:=sum + Size(c)*Order(Representative(c));
  od;
  return(sum mod Size(g)) = 0;
end;

smallGroupsSearch := function(N, IntAvgOrder)
   local hits, n, i,g;
  hits:=[];
  for n in [1..N] do
    for i in [1..NrSmallGroups(n)] do
      if IntAvgOrder(n,i) then Add(hits,[n,i]);
      fi;
    od;
  od;
  return hits;
end;
```

There are two obvious places to introduce data parallelism: i) the `smGrpSearch` function generates a list of integers between a low value (`lo`) and a high value (`hi`), applying `predSmGrp` to each integer; and ii) the `ifmatch` function relies on the `masterWorker` skeleton to generate a set of hierarchical master worker tasks to calculate `IntAvgOrder` in GAP.

[3] The full definition may be found at `http://www.macs.hw.ac.uk/~ceeatia/smallGroup.hs`

There are also two levels of irregularity in the parallel structure: firstly, the number of groups of a given order varies enormously; and secondly there are significant variations in the cost of computing the conjugacy classes of each group.

5. MultiCore Performance Results

Our measurements are performed on an eight-core Dell PowerEdge 2950 machine located at the University of St Andrews (*ardbeg*). This machine is constructed from two quad-core Intel Xeon 5355 processors running at 2.66GHz. *ardbeg* has a 1333MHz front-side bus, and 16GB of fully-buffered 667MHz DIMMs. It runs CentOS Linux 4.5 (kernel version 2.6.9-55), and uses the current development version of Eden based on GHC-6.8.2.

5.1 *liouville* Performance

No PEs	Rtime	Spdup	CPU Utilis.
1	526s	1	92.8%
2	264s	1.9	89.6%
3	178s	2.9	93.1%
4	132s	3.9	92.0%
5	106s	4.9	90.7%
6	89s	5.9	90.7%
7	76s	6.9	89.4%
8	68s	7.7	88.9%

Table 1. Parallel performance of *liouville* $[1 \ldots 25{\times}10^6)]$

PE	Proc	Thr	Mesages	
			Sent	Recv
#1	4	36	270476	270449
#2	3	3	31988	31985
#3	3	3	33359	33356
#4	3	3	31542	31539
#5	3	3	33309	33306
#6	3	3	32709	32706
#7	3	3	33517	33514
#8	3	3	33244	33241

Table 2. Per-PE performance of *liouville* $[1 \ldots 25{\times}10^6]$

Table 1 shows the performance of the summatory *liouville* function for arguments ranging between 1 and $25{\times}10^6$. The first column shows the number of PEs that are involved in the computation; the second and third columns show the runtime and speedup; the final column shows overall CPU utilisation, as a percentage of the available processors. In order to reduce the impact of operating system and other effects, all runtimes shown in this paper are the mean of five measured times. We can clearly see that we obtain good performance, with speedups *over the sequential GAP system* of up to 7.7 on eight cores, and CPU utilisation of up to 93.1%. The maximum utilisation occurs when three cores are used, after which there is a slight degradation to 88.9% utilisation on eight cores. There is essentially no parallel overhead on one core: the `SymGrid-Par` middleware simply needs to start the *liouville* computation in GAP and then to record the final result that is communicated to it (an integer).

Table 2 shows the individual performance of each of the eight PEs for the final case, when all eight cores are active on the *liouville* program. The first column identifies the PE of interest; the second and third columns show the number of processes and threads that are generated by the PE; and the fourth and last columns show the number of messages sent/received by each PE, respectively. Besides the main process on PE 1, one Eden *process* on every PE is involved in coordinating precisely one GAP evaluation engine, and may give rise to several runtime threads. Two more – very short – processes per PE are responsible for setting up and shutting down the GAP-Haskell interface. Overall, we can see that there is an even distribution of processes and threads on each PE apart from #1 (the master PE), the one where the master process of `masterWorker` resides. The messages show a typical master-worker behaviour, where a similar number of messages are sent and received by each worker process. Overall, all but the master PE sends and receives around 32,000 messages (approx. 3MB in total, or 42KB/s). The master process on PE #1 sends and receives all tasks for the worker processes, resulting in around 270,000 messages (approx. 27MB in

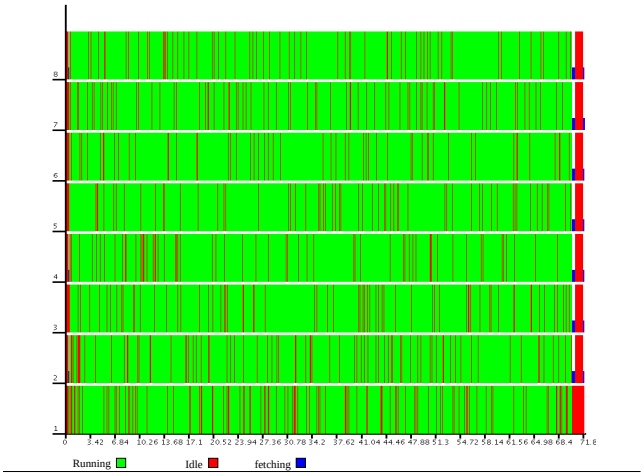

Figure 4. per-PE Activity Profile for *liouville*

total, or 350KB/s). In our setup of eight PEs, this is acceptable. If the master-worker system was scaled up to larger numbers of PEs, a hierarchical implementation might be needed, however.

Figure 4 shows the per-PE activity profile for *liouville* on eight PEs (each of which is mapped to a different core), plotting the behaviour of each of the PEs (y-axis) against execution time (x-axis). Each PE is visualised as a horizontal line. The colour of these lines changes depending on the state of the PE: a mid shade of gray (green in a colour profile) indicating a running PE; gaps or darker shade in the horizontal lines (red areas in the colour profile) indicating blocked processes; and black (blue in a colour profile) indicating PEs that do not run any process. As already indicated by the high CPU-usage, the performance is good overall, with active processes on all PEs throughout most of the computation.

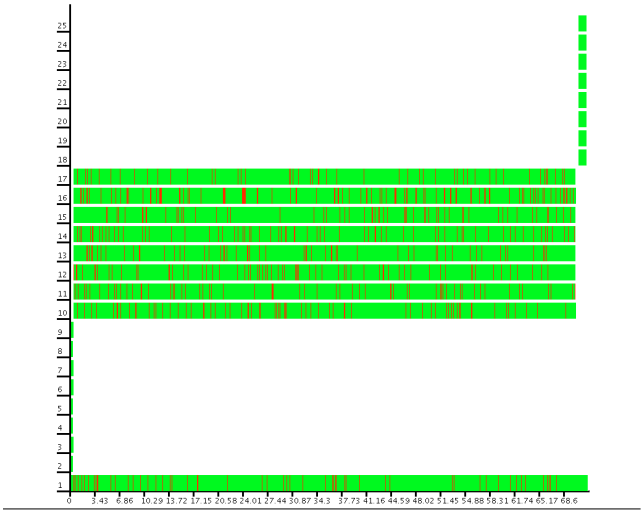

Figure 5. Per-Process Coordination Profile for *liouville*

The processes and their exact role in the parallel computation become clear in Figure 5, which visualizes the behaviour of each process in the program (y-axis) over execution time (x-axis). As before, the behaviour of each process is visualised as a horizontal line: a mid shade of gray (green in a colour profile) indicating that the process is involved in coordinating a GAP task; and a darker shade (red areas in the colour profile) indicating that the process is running. During coordination, the Haskell process is not active, but is suspended waiting for GAP to complete the required computation.

We can see from this diagram that the main task (process #1) is active throughout the execution, taking the master role in the skeleton. One process is created on each PE at the start of the program (#2–#9) to set up the GAP-Haskell interface. The computation itself is carried out by processes #10–#17, again one per PE. Each of these processes receives tasks, forward them to the GAP instance running on the same node and communicates results back to the master. As the predominantly green plot shows, these processes spend most of their time waiting for the external GAP calls to complete, and thus do not impose significant execution overheads. Finally, we see a final set of eight processes, #18-#25 which perform a controlled GAP shutdown before the Haskell program terminates. It is obvious from Figure 5 that the Eden processes do not incur significant overhead over the GAP tasks, since they are only actively executing for a small part of the total execution time (shown in red). It is also clear that coordination activities are spread evenly across the execution of the program.

No PEs	Rtime	Spdup	CPU Utilis.
1	480s	1	96.0%
2	246s	1.9	96.0%
3	165s	2.9	98.6%
4	125s	3.8	98.0%
5	104s	4.6	99.2%
6	91s	5.2	98,7%
7	82s	5.8	98.3%
8	76s	6.3	97.0%

Table 3. Parallel performance of *smallGroup* [1. . .350]

PE	Proc	Thr	Mesages	
			Sent	Recv
#1	4	5969	83796	80628
#2	3	352	9912	9560
#3	3	352	9729	9377
#4	3	352	9521	9369
#5	3	352	9483	9131
#6	3	352	9626	9274
#7	3	352	9262	8910
#8	3	352	9355	8903

Table 4. Per-PE Performance of *smallGroup* [1. . .350]

5.2 *smallGroup* Performance

Table 3 shows the performance of the *smallGroup* program on a sequence of group orders between 1 and 350. As shown by Figure 6, the number of groups of the given order varies enormously (*by 5 orders of magnitude*), and this, in turn, directly impacts the granularity of the tasks that are created, leading to highly irregular parallelism. As before, the first column shows the number of cores that are involved in the computation; the second and third columns show corresponding runtimes and speedups; and the final column shows overall CPU utilisation, as a percentage of the available processors. Our results show that we can achieve real speedups over sequential GAP of up to 6.3 on eight cores, with a maximum utilisation of 99.2% at five cores. As with *liouville*, utilisation tails off with an increasing number of cores, falling to 97% at eight cores.

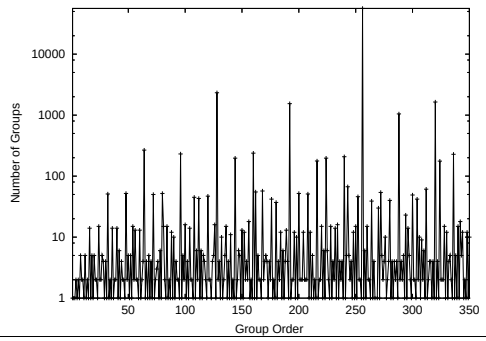

Figure 6. Number of Groups of Orders in the range [1. . .350]

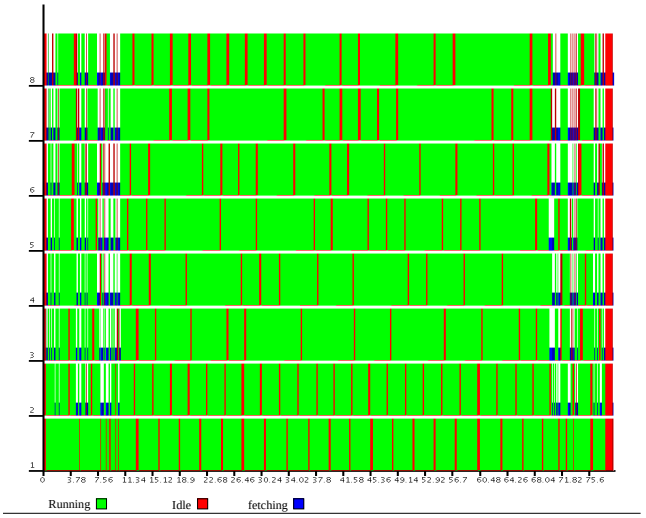

Figure 7. Per-PE (Per-Core) Activity Profile for *smallGroup*

The higher utilisation, but lower absolute speedup than for *liouville* suggests that some processor performance is devoted to coordination overhead.

Table 4 shows the individual performance of each of the eight PEs for the *smallGroup* program. As with *liouville*, there is an even distribution of processes and threads on each PE apart from #1 (the master PE). Also as with *liouville*, a similar number of messages are sent and received by each PE apart from the master. Overall, each PE apart from the master sends and receives around 9000 messages. The master PE sends and receives around 80000 messages. This is consistent with the *liouville* example, where the master PE also sends/receives almost nine times as many messages as each of the other PEs. Finally, Figure 7 shows the per-PE (or per-core) activity profile for *smallGroup* on eight PEs. As for *liouville*, we observe good overall performance. However, there are easily discernible periods towards the start and end of execution where all PEs are involved in transmitting data, and therefore are not actively searching for groups.

5.3 Discussion of Overall Performance Results

Tables 1 and 3 show that `SymGrid-Par` delivers good performance for both programs on varying numbers of cores. For both programs, we obtain a speedup of 1.9 on two PEs; and we obtain speedups of 7.7 and 6.3 on eight PEs for the summatory *liouville* function and *smallGroup* search, respectively. Moreover, we also achieve good overall CPU utilisation.

The good performance that we obtain for `SymGrid-Par` on multicore architecture is mainly due to the way that it deals with lightweight threads. More precisely, while the `SymGrid-Par` skeletons produce relatively fine-grained programs, which could then potentially swamp the system (43), the skeletons we have used increase granularity by combining smaller threads into larger computational units, so avoiding swamping and improving overall performance in multicore architecture using message-passing implementation. As shown in Tables 2 and 4, we generate 63 parallel threads from summatory *liouville* function and 8433 parallel threads from *smallGroup* search. `SymGrid-Par` generates different number of threads according to the problem's irregularity and potential parallelism. As discussed in Section 4.2, the *smallGroup* program is far more irregular than the *liouville* program and our system therefore generates more threads in order to maximise the parallelism in this case. *smallGroup* and *liouville* also have significant amounts of communication: on eight PEs (cores) *smallGroup* transmits 3892 message per second, and *liouville* transmits 14709 messages per second. This is handled efficiently by our message-passing implementation in multicore architecture.

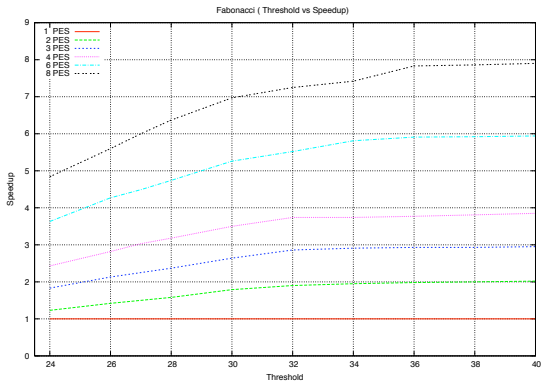

Figure 8. Fibonacci(51), speedup versus threshold

Cores	No. of Threads	Thread Granularity (ms)	Thread Granularity (Mcycles)
1	2097152	0.32	0.53
2	1048576	0.24	0.56
3	699051	0.24	0.57
4	524288	0.25	0.57
6	349525	0.25	0.57
8	262144	0.25	0.57

Table 5. Performance analysis of `Fibonacci` with threshold 30

Garbage collection and memory management designs play a key role in obtaining good multicore performance. In contrast with many designs for shared-memory systems, implementing `SymGrid-Par` on top of Eden processes allows each PE (core) to garbage collect its own heap independently, i.e. without synchronising with other PEs. This reduces memory contention between the cores during garbage collection, which can be a major problem with multicore implementations. For our examples, the garbage collector and memory management system is well-exercised, with *smallGroup* allocating $2.2GB$ of heap memory; and *liouville* allocating $308MB$ of heap, on our eight core machine.

As shown in Tables 2 and 4, all PEs apart from the main one generate the same number of processes and threads, and send/receive very similar numbers of messages. The exception for PE #1 is due to the way that workload management is handled in `SymGrid-Par`, and especially in the `masterWorker` skeleton that was used in our experiments. After the initial setup phase, PE #1 calls the skeleton and executes the master process, which distributes the entire task-set to the respective worker processes. It thus participates in virtually every communication. As explained above, more sophisticated implementations of the master-worker skeleton could circumvent the potential communication bottleneck when scaling up to a larger number of cores (8). Finally, Figures 4 and 7 show that all PEs are uniformly loaded, finish at the same time and include few idle periods. The idle periods shown here are due to PEs waiting for data to be evaluated, without having any other threads to be executed. It is obvious from the figures that all eight PEs are fully used throughout the computation.

6. MultiCore Thread Granularity

The preceding section shows that the Eden-based `SymGrid-Par` framework can effectively parallelise relatively coarse-grained computations for multicore architecture. A question arises, however, over precisely how coarse-grained each thread needs to be in

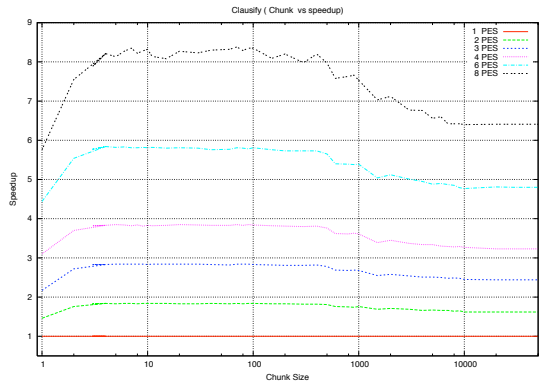

Figure 9. Clausify(50000), speedup versus chunk size

Cores	No. of Threads	Thread Granularity (ms)	Thread Granularity (Mcycles)
1	12500	27.92	65.07
2	6250	25.39	59.17
3	4167	24.75	57.7
4	3125	24.44	56.94
6	2084	24.04	56.02
8	1563	22.78	53.08

Table 6. Performance analysis of `Clausify` with chunk-size 4

order for us to obtain acceptable speedups. The answer to this question entails balancing two dynamic properties of a parallel program, namely the thread granularity and amount of communication.

We have addressed this question by undertaking a limit study that uses a naïve recursive `Fibonacci` function. This function has a divide-and-conquer recursive definition. In the parallel implementation, we generate parallel threads down to a certain threshold. Each leaf thread then runs sequentially to produce the Fibonacci function of some argument, before the results are combined to give an overall solution. This solution requires minimal communication: each sequential thread takes a single `Int` argument and generates a single `Int` result (the Fibonacci function of its argument). Figure 8 plots the speedup for 1-8 cores, given different threshold settings. It shows how speedup varies against thread granularity, as determined by the threshold parameter. All speedup curves show a steady increase in speedup as the threshold is increased up to 30, and are stable thereafter. For instance, on eight cores, when the threshold was changed from 24 to 40, speedup improved from 4.84 to 7.90 and on two cores the speedup improved from 1.23 to 2.00. Table 5 analyse the performance of `Fibonacci` given an input of 51 and a threshold 30. The first column shows the number of cores that are involved in the computation; the second column reports the number of threads that are generated; and the third and fourth columns, respectively, show the execution time for each thread in milliseconds and Mcycles (Millions of clock cycles). We conclude that a minimum thread granularity of the order of 1ms (1 Mcycles) is required to gain good parallel performance from a message-passing semi-explicit functional language like Eden on a multicore similar to *ardbeg*.

However, typical programs will communicate more data per thread than our rather ideal `Fibonacci` example, and coarser granularities may then be needed to offset the communication time. As an example involving more communication per thread, we consider the `Clausify` program from the `nofib` benchmark suite (42). `Clausify` parses logical propositions and puts them in clausal form. From a parallel perspective, this is a classical data-parallel program. We use a parameter (chunk size to control the granularity of the threads that are created by varying the number of items that evaluated in each data parallel task. Figure 9 shows speedup for this program plotted against various chunk sizes. It is clear that while the choice of chunk size (and hence granularity) has some impact on performance, there is not the linear relationship that we observed with the `Fibonacci` function. On eight PEs, we observe a variation in speedup ranging from 4.77 to 8.38; and on two PEs we observe variation between 1.00 and 1.84, with excessively large or small chunk sizes both having a negative impact on performance. The knee of the graph occurs with a chunk size of 4. Table 6 shows that different number of threads are generated according to the number of PEs that are involved in the computation. However, the granularity of the threads that are generated is similar regardless of the number of cores (varying between $23ms$ on one core and 28 ms on eight cores). This suggests that we should aim for a thread granularity of around 30 ms or 70 Mcycles if we wish to obtain effective parallelism for `Clausify`. These results are consistent with others that we have obtained for another data-parallel program: the `Rewrite` program (also from the `nofib` suite) requires a minimum thread granularity of 28 ms or ≅66 Mcycles. The code for our examples can be found at `http://www.macs.hw.ac.uk/~ceeatia/\{fibonacci,clausify\}.hs`.

7. Related Work

7.1 Parallelism in Multicore Architecture

The recent trend towards multicore architectures has sparked a significant amount of new work that is aimed at exploring novel programming models and runtime systems for such architectures (e.g. (35; 14; 12; 25; 27; 34; 5; 15)). This work has exploited a number of different approaches.

- Parallel libraries, such as *Pthreads* (37) and Phoenix (45), provide the programmer with the ability to express parallelism directly. In fact, Phoenix is an implementation of Google MapReduce for shared-memory systems. The Phoenix runtime automatically manages thread creation, dynamic task scheduling, data partitioning. The Phoenix runtime system is implemented on top of the Pthreads library.

  More advanced systems, such as Cilk (25) or OpenMP (16), provide higher level parallelisation primitives, including a mechanism to automatically schedule parallelism for performance, and support for nested parallelism (22). However, according to Bridges (11), this approach provides little support to help achieve correct or effective parallelism. For instance, the Cilk `inlet` directive is similar to the `Commutative` directive, in ensuring correct execution of code in a non-deterministic fashion. However, `inlet` is meant to serially update state upon return from a spawned function, while `Commutative` is meant to facilitate parallelism by removing serialisation. Clearly, it would be easy to use the wrong construct.

- One system that has taken a similar message-passing approach to that we have described for our implementation is Erlang (6; 23). We have been able to find very few published performance results for Erlang multicore implementations, and those we have found seem unreliable (for example, a speedup of 18 on eight cores suggests a fundamental problem with the underlying sequential implementation!). We have also been unable to find much implementation detail: we surmise that Erlang threads are probably mapped directly to operating system threads, created Unlike our approach, we do know that all communication/thread synchronisation in Erlang is explicit, using send and receive primitives. Using the approach we have de-

scribed, many of the low-level and complex details that must be dealt with in Erlang or other explicitly parallel notations (marshalling/unmarshalling complex data structures, thread allocation, communication, thread synchronisation, deadlock avoidance etc., are managed automatically in our system). As with the parallel libraries approach, Erlang provides little or no support for achieving correct or effective parallelism.

While there are well-known and obvious differences between Haskell and Erlang — Haskell is statically-typed, but non-strict, where Erlang is dynamically typed, but strict — we are not convinced that these will have a major impact on the programming model we have used, except perhaps in sequential performance. Because it is dynamically typed, for example, the Erlang implementation lacks some optimisations that are possible in Haskell. More significant is our general approach of treating Haskell as a coordination language with algorithmic skeletons used to introduce parallelism, that is then mapped automatically to multicore threads executing sequential program components. We find it especially significant that, using our approach, we have been able to obtain real speedups for sequential code fragments on multicore, and that this parallelism has been expressed in an essentially declarative way (using parallel algorithmic skeletons).

- An alternative technique is to use either implicit or explicit memory transactions (14; 34; 1). Some such approaches require an explicit step to make locations or objects part of a transaction, while other approaches make the memory operation behaviour implicit. Implicit transactions require either compiler or hardware support. Both of those techniques have been proposed to help the programmer express parallelism in an easier manner.

  For example, Harris and Fraser (32) added a *conditional critical region* (CCR) statement of the form `atomic (E) {S}` to Java. This automatically executes `S` when the expression `E` evaluates to true. Harris, Marlow and Peyton Jones subsequently (33) applied a similar *software transactional memory* approach to Concurrent Haskell, adding a transaction monad to Concurrent Haskell and introducing composable memory transactions with operations to block and compose nested transactions

  Grossman *et al* (29) observe some problems with "weak atomicity" that can arise with some software transactional memory approaches. They note that conditional critical regions provide a way to write multiple statements so that they appear to occur *atomically* to the entire system: either all of the CCR is guaranteed to have executed or none of it will have. In addition to these correctness issues, memory transaction techniques can also suffer from poor performance. For example, Zilles and Flint (51) clearly indicate the difficulties and challenge of improving performance using memory transactions alone, and Harris, Marlow and Peyton Jones (34) report poor and highly variable parallel performance, using memory transaction techniques on shared-memory machines.

- In contrast to the task-parallel approach we have described in this paper, where parallelism is exposed from the program's control flow by applying a series of skeletons, data-parallel approaches expose parallelism by evaluating elements of bulk data structures in parallel. Where a program has a regular, data-driven structure, this can be a useful technique. Indeed, the `parMap` skeleton described above follows essentially such a pattern.

  Data-parallel implementations are beginning to be applied in a multicore setting. For example, Data-Parallel Haskell (15) provides parallel arrays and special parallel operations to handle them. Static rewrites are used to eliminate unnecessary conversion steps. The optimising compilation follows sound mathematical rules, and is in the process of being implemented in GHC. Good results are reported for typical data-parallel problems (such as sparse matrix multiplication), with relative speedup of up to 3.8 on a quad-core Intel Xeon (similar to the figures that we obtain), 6.8 on an eight-processor AMD Opteron NUMA machine (using a HyperTransport bus), or up to 12.9 on a sixteen-processor Sun UltraSparc IV+ shared-memory machine. A similar approach is taken by Fluet *et al.* (24) who embed nested data-parallel constructs into an explicitly parallel Concurrent ML setting.

  Data-parallelism is, however, primarily suited for regular, data-driven parallelism, and clearly cannot capture all forms of parallel programming, such as the irregular task parallelism we have shown for *smallGroup*. Interestingly, when the AMD is reconfigured as a sixteen-core shared-memory machine (8 dual-core processors connected via the HyperTransport bus), performance on the sparse matrix example drops to less than a factor of seven speedup on sixteen cores, and only a factor of 4.8 on eight cores. This is apparently because the data parallel approach taken by Chakravarty et al, is not a good match to the non-uniform memory architecture used in this machine – the non-uniformity is lessened when the machine is configured to use only one of the two cores that are available on each processor. We speculate that since it is designed for such an environment, our approach would be better able to cope with the peculiarities of such an architecture.

In contrast to most of the approaches described above, we focus on exploiting message-passing and virtual shared-memory using multicore architecture to give an effective implementation of declarative parallelism. The closest parallel to our work is Harris and Singh's (35) attempt to parallelise the `nofib` benchmark suite (42) using SMP-GHC. Harris and Singh use a "lock-free" mechanism (34) and profiling and recompiling techniques to gain parallelism for a multicore architecture, whereas we use message-passing with message-level locking. Using their approach, Harris and Singh report a maximum parallelism of 1.8 using a quad-core architecture for the `hidden` program. This is, in fact, the best performance result that has been obtained from either attempt to obtain multicore parallelism using SMP-GHC, and other performance results are highly variable. In contrast, we present a consistent speedup of a factor of 7.7 on an eight-core architecture for real computational mathematical algebra applications.

7.2 Other Parallel Implementations of Symbolic Computation Systems

Work on parallel symbolic computation dates back to at least the early 1990s – Roch and Villard (46) provide a good general survey of early research. Within this general area, significant research has been undertaken to parallelise specific computational algebra algorithms, notably term re-writing and Gröbner basis completion (e.g. (4; 13)). A number of one-off parallel programs have also been developed for specific algebraic computations, mainly in representation theory (40). We are not aware of any implementations of these systems that specifically target multicore architectures, however. While several symbolic computation systems include some form of operator to introduce parallelism (e.g. parallel GCL, which supports Maxima (19), parallel Maple (7), or parallel GAP (20)), very few *production* parallel algorithms have been produced. This is partly due to the complexities involved in programming such algorithms using explicit parallelism, which are accidental for many programmers in this domain. It is also partly due to the lack of generalised support for communication, distribution etc, in these systems. By abstracting over such issues, and providing system-

independent coordination of parallel programs, our approach considerably simplifies the process of constructing parallel computational algebra systems.

7.3 Other Systems Linking Parallel Functional Languages and Computer Algebra Systems

There have been a few previous attempts that have aimed to link parallel functional programming languages with computer algebra systems. For example, the GHC-Maple interface (28) and the Eden-Maple system (39) both link parallel implementations of Haskell with the widely-used Maple system. None of these systems is in widespread use at present, none supports the broad range of computational algebra systems we are targeting, has the support of the developers of those systems, and none has yet achieved similar multicore results to those reported here.

8. Conclusions

We have introduced a high-level parallel programming model using algorithmic skeletons with a Haskell-based message-passing implementation to coordinate sequential computational algebra components on a multicore architecture. Unlike many other approaches to parallel programming, the approach is "low pain" in the sense that the programmer constructs a parallel program by inserting a small number of algorithmic skeletons at a few key points in the program. In contrast with some other approaches, notably those based on software transactional memory, our approach is also "high gain". In fact, despite being much more general, it delivers speedups that are as good as, or better than a data-parallel Haskell implementation on a multicore machine. It is perhaps surprising that an implementation using relatively heavyweight message-passing, more commonly associated with distributed memory architectures, can achieve such good performance on a multicore architecture. Our results, however, show that our approach delivers achieve extremely good utilisation on a typical eight-core machine, with excellent speedups over the sequential version of the same program. For two real symbolic applications running under the widely-used GAP computational algebra system, we have shown that we can achieve processor utilisation up to 97% and speedup of up to 7.7 on our eight-core testbed machine. This contrasts with typical shared-memory approaches, which can struggle to achieve 50% utilisation on a quad-core machine (34) and represents highly desirable performance gains for GAP users, especially at minimal user effort.

We have investigated the the minimum thread granularity where our relatively heavyweight parallel programming model can deliver good performance on multicore architectures. From a limit study we conclude that a minimum thread granularity of the order of 1 Mcycle is required, corresponding to 1ms on our target architecture. Most parallel programs communicate far more data than such ideal programs, and greater thread granularity is required to offset the communication time. Measurement of programs that undertake more communication suggests that a thread granularity of approximately 70 Mcycles, or 30ms on our target architecture, is required to achieve good parallel performance.

The key reasons for the unusually good parallel performance of our implementation is that all coordination activities (mainly locking and communication) are both coarse-grained and confined to well-understood libraries. This significantly reduces the need for fine-grained speculative locking that is found in many multicore implementations – it is only necessary to lock items that are *actually shared*, and, since we are communicating substantial subgraphs rather than individual nodes, a single lock suffices for many data items.

In addition to good performance, our approach carries major advantages for developers of parallel applications. In particular, no changes need to be made to the highly-optimised sequential engine that is used to execute the application. Rather we coordinate multiple instances of each engine to execute our parallel application. The middleware then maps threads to the available engines, dynamically rebalancing the workload, and throttling the creation of threads, as required by the application. This gives a powerful, yet highly general approach: we can use a single parallel system to coordinate computations that are written in a variety of computational algebra systems. Moreover, our work is not restricted to symbolic computations, but can, in principle, be applied to a wide variety of application domains.

It might be argued that any parallel program that delivers good performance on a distributed-memory parallel architecture could be expected to deliver even better performance on a shared-memory architecture as communication costs are so much lower. However such a simple argument ignores crucial memory, synchronisation and threading aspects of the parallel architecture. A multicore parallel language implementation must manage memory effectively: if the maximum residency required by all cores exceeds the physical memory then performance will be destroyed by virtual memory costs. Both memory allocation and distributed garbage collection must be efficient, some key issues are that garbage collection on one core must not require substantial synchronisation with other cores, and unused references between cores must eventually recovered. Eden addresses these issues by applying a distributed-memory model *ab initio*, so that all cores have disjoint heaps. In a multicore parallel language implementation, the synchronisation required to pass messages must not induce contention in the underlying architecture. Eden devolves this issue to the PVM communications library. The performance results show that we do achieve good contention. Finally, given the high latency induced by message-passing, a multicore parallel language implementation must provide lightweight multi-threading to utilise the cores effectively during communication.

9. Future Work

There are a number of obvious avenues for us to explore. Firstly, we have shown that our results scale well to the largest multicore system that we were able to access (an eight-core Intel Xeon server), giving utilisation up to 97% on eight cores. Now that 16-core machines are becoming available, it would be interesting to determine whether our results extend to these larger machines, or whether the slight tailing off in utilisation that we observe with the eight-core machine will be accentuated if additional cores are added.

Secondly, we have only considered a single computational algebra host system, GAP. It would be interesting to see whether our results also apply to other systems such as Maple or Mathematica. We are in the process of constructing a generic interface to computational algebra systems as part of the SCIEnce project, and hope to report such results in due course.

Thirdly, the programs we have orchestrated are relatively coarse-grained, and a message-passing architecture will only deliver good performance if the task and communication granularity is sufficiently large to offset the overheads of communicating and synchronising on the messages. We are currently exploring this design space to identify the minimum task and communication granularity for common multicore architectures. While we have gained good performance by using a *master-worker* skeleton to map irregular fine-grained parallel tasks to larger computational units (PEs), it would be interesting to see whether our results will also hold for programs that directly manage finer-grained threads. It would also be interesting to compare our results directly with those that have been obtained for software transactional memory and data-parallel approaches.

Acknowledgments

This research is generously supported by the European Union Framework 6 grant RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe; and by the UK Engineering and Physical Sciences Research Council (EPSRC) grant EP/F030592/1 (Islay).

References

[1] A-R Adl-Tabatabai, B.T. Lewis, V. Menon, B.R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[2] A. Al Zain, P. Trinder, K. Hammond, and S. Linton. Orchestrating Computational Algebra Components into a High-Performance Parallel System. In *Proc. 2008 IEEE International Symposium on Parallel and Distributed Computing with Applications (ISPA '08)*, 2008.

[3] A. Al Zain, P. Trinder, H-W. Loidl, and G. Michaelson. Evaluating a High-Level Parallel Language (GpH) for Computational GRIDs. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):219–233, 2008.

[4] B. Amrheim, O. Gloor, and W. Küchlin. A Case Study of Multithreaded Gröbner Basis Completion. In *Proc. ISSAC '96: International Symposium on Symbolic and Algebraic Computation*, pages 95–102. ACM Press, 1996.

[5] C.K. Anand and W. Kahl. A Domain-Specific Language for the Generation of Optimized SIMD-Parallel Assembly Code. SQRL Report 43, Software Quality Research Laboratory, McMaster University, May 2007. available from http://sqrl.mcmaster.ca/sqrl_reports.html.

[6] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in* ERLANG. Prentice Hall, 2nd edition, 1996.

[7] L. Bernardin. Maple on a Massively Parallel, Distributed Memory Machine. In *Proc. PASCO '97: Intl. Symp. on Parallel Symbolic Computation*, pages 217–222. ACM Press, 1997.

[8] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical Master-Worker Skeletons. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2008.

[9] J. Berthold, S. Marlow, A. Al Zain, and K. Hammond. Comparing and Optimising Parallel Haskell Implementation. In Sven-Bodo Scholz, editor, *IFL'08 — Implementation and Application of Functional Languages 20th International Symposium*, Draft Proceedings, pages 223–240, Hetfield, Hertfordshire, UK, September 2008. Technical Report No. 474.

[10] S. Breitinger, R. Loogen, Y. Ortega Malln, and R. Pea Marí. Eden — The Paradise of Functional Concurrent Programming. In *EuroPar'96 — European Conf. on Parallel Processing*, LNCS 1123, pages 710–713, Lyon, France, 1996. Springer.

[11] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August. Revisiting the Sequential Programming Model for Multi-Core. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Broadcom Corp. BCM1250 Multiprocessor. Technical report, Broadcom Corporation, April 2002.

[13] R. Bündgen, M. Göbel, and W. Küchlin. Multi-Threaded AC Term Re-writing. In *Proc. PASCO'94: Intl. Symp. on Parallel Symbolic Computation*, volume 5, pages 84–93. World Scientific, 1994.

[14] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Cao Minh, C. Kozyrakis, and K. Olukotun. The atomos transactional programming language. In *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. Jun 2006.

[15] M.M.T. Chakravarty, R. Leshchinskiy, S.L. Peyton Jones, G. Keller, and S. Marlow. Data Parallel Haskell: a Status Report. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming)*, Nice, France, 2007. ACM Press.

[16] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[17] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, MA, 1989.

[18] M.I. Cole. Algorithmic Skeletons. In K. Hammond and G. Michaelson, editors, *Research Directions in Parallel Functional Programming*, chapter 13, pages 289–304. Springer-Verlag, 1999.

[19] G. Cooperman. STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra Systems. In *Proc. ISSAC '95: International Symposium on Symbolic and Algebraic Computation*, volume 249 of *Lecture Notes in Control and Information Sciences*, pages 126–132. ACM Press, 1995.

[20] G. Cooperman. GAP/MPI: Facilitating Parallelism. In *Proc. DIMACS Workshop on Groups and Computation II*, volume 28 of *DIMACS Series in Discrete Maths. and Theoretical Comp. Sci.*, pages 69–84. AMS, 1997.

[21] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[22] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic Thread Distribution for Nested Parallelism in OpenMP. In *19th Annual International Conference on Supercomputing (ICS '05)*, pages 121–130, New York, NY, USA, 2005. ACM.

[23] Ericsson Utvecklings AB. *Erlang Home Page*.

[24] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A Heterogeneous Parallel Language. In *DAMP'07: Workshop on Declarative Aspects of Multicore Programming*, Nice, France, 2007.

[25] M. Frigo, C.E. Leiserson, and K.H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI'98 — Conf. on Programming Language, Design and Implementation*, volume 33 of *ACM SIGPLAN Notices*, pages 212–223. ACM Press, 1998.

[26] GHC. `http://www.haskell.org/ghc/`.

[27] M.I. Gordon, W. Thies, M. Karczmarek, J. Lin, A.S. Meli, A.A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS-X: 10th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[28] The GHC-Maple Interface, `http://www.risc .uni-linz.ac.at/software/ghc-maple/`.

[29] D. Grossman, J. Manson, and W. Pugh. What do High-Level Memory Models mean for Transactions? In *MSPC '06: 2006 workshop on Memory System Performance and Correctness*, pages 62–69, New York, NY, USA, 2006. ACM.

[30] The GAP Group. GAP – Groups, Algorithms, and Programming, 2007. `http://www.gap-system.org`.

[31] K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*, chapter Introduction. Springer-Verlag, 1999.

[32] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

[33] T. Harris, S. Marlow, and S. Peyton Jones. Composable Memory Transactions. In *PPoPP 2005: Principles and Practice of Parallel Programming*.

[34] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a Shared-Memory Multiprocessor. In *Proc. Haskell '05: 2005 ACM SIGPLAN Workshop on Haskell*, pages 49–61. ACM Press, September 2005.

[35] T. Harris and S. Singh. Feedback Directed Implicit Parallelism. In *ICFP '07: 2007 ACM SIGPLAN International Conference on Functional Programming*, pages 251–264, New York, NY, USA, 2007. ACM.

[36] R. Lämmel. Google's MapReduce Programming Model – Revisited. *Sci. Comput. Program*, 70(1):1–30, 2008.

[37] B. Lewis and D.J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[38] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[39] R. Martínez and R. Pena. Building an Interface Between Eden and Maple. In *Proc. IFL 2003, Springer-Verlag LNCS 3145*, pages 135–151, 2004.

[40] G. O. Michler. *High Performance Computations in Group Representation Theory*. Preprint, Institut für Experimentelle Mathematik, Univerisität GH Essen,, 1998.

[41] The OpenMath Standard, Version 2.0, `http://www.openmath.org/`.

[42] W. Partain. The `nofib` Benchmark Suite of Haskell Programs. In *Proc. 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag.

[43] S.L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP — a High-Performance Architecture for Parallel Graph Reduction. In *Intl. Conf. on Functional Programming Languages and Computer Architecture (FPCA '87)*, LNCS 274, pages 98–112, Portland, Oregon, September 1987. Springer-Verlag.

[44] S.L. Peyton Jones, C.V. Hall, K. Hammond, W.D. Partain, and P.L. Wadler. The Glasgow Haskell Compiler: a Technical Overview. In *Proc. JFIT (Joint Framework for Information Technology) Technical Conference*, pages 249–257, Keele, UK, March 1993.

[45] C Ranger, R. Raghuraman, A. Penmetsa, G.R. Bradski, and C. Kozyrakis. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *HPCA '07: 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[46] L. Roch and G. Villard. Parallel computer algebra. In *Proc. ISSAC '97: International Symposium on Symbolic and Algebraic Computation*. Preprint IMAG Grenoble France, 1997.

[47] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[48] Tian Tian and Chiu-Pi Shih. Software Techniques for Shared-Cache Multi-Core Systems, 2007. Online article in Intel developer community.

[49] P. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Proc. PLDI'96*, pages 79–88, Philadelphia, PA, USA, May 1996.

[50] P.W. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Functional Programming*, 8(1):23–60, January 1998.

[51] C. Zilles and D. Flint. Challenges to Providing Performance Isolation in Transactional Memories. In *Workshop on Duplicating, Deconstructing, and Debunking*. 2005.