
Orchestrating computational algebra components into a high-performance parallel system

A.D. Al Zain* and P.W. Trinder

Computer Science Department,
School of Mathematical and Computer Sciences,
Heriot-Watt University,
Edinburgh, UK
E-mail: a.d.alzain@hw.ac.uk
E-mail: p.w.trinder@hw.ac.uk
*Corresponding author

Comment [t1]: Author: Please provide the full mailing address of A.D. Al Zain and P.W. Trinder.

K. Hammond

School of Computer Science,
University of St Andrews,
St. Andrews, UK
E-mail: kh@cs.st-and.ac.uk

Comment [t2]: Author: Please provide the full mailing address of K. Hammond.

Abstract: This paper demonstrates that it is possible to obtain good, scalable parallel performance by coordinating multiple instances of *unaltered* sequential computational algebra systems in order to deliver a single parallel system. The paper presents the first substantial parallel performance results for *SymGrid-Par*, a system that orchestrates computational algebra components into a high-performance parallel application. We show that *SymGrid-Par* is capable of exploiting different parallel/multicore architectures without any change to the computational algebra component. Ultimately, our intention is to extend our system so that it is capable of orchestrating heterogeneous computations across a high-performance computational grid.

Keywords: parallel coordination; orchestration; computational algebra; cluster; multicore; grid computing; Liouville function; functional programming; Haskell; GAP; Maple.

Reference to this paper should be made as follows: Al Zain, A.D., Trinder, P.W. and Hammond, K. (xxxx) 'Orchestrating computational algebra components into a high-performance parallel system', *Int. J. High Performance Computing and Networking*, Vol. X, No. Y, pp.000–000.

Biographical notes: Abdallah D. Al Zain received his BSc in Computer Science from the Applied Science University, Jordan, in 1998 and PhD degree from the Heriot-Watt University, Edinburgh, UK in April 2006. He has a solid background in parallel and distributed systems and languages, with a strong focus on functional approaches to grid systems and multicore architectures. In the last seven years, he has published more than 20 conference and journal papers and a book chapter.

Phil Trinder received his BSc from Rhodes University, South Africa, and DPhil from Oxford University. He is currently a Full Professor of Computer Science at the Heriot-Watt University. He has a strong record of research in the design, implementation and evaluation of high-level parallel and distributed programming languages.

Kevin Hammond received his BSc and PhD from the University of East Anglia, UK. He has published extensively in the general field of programming language design and implementation, producing over 75 research publications in total. His work has been supported by more than 20 national and international research grants. He is presently a Professor in Computer Science at the University of St. Andrews.

1 Introduction

We describe the design and implementation of a new system for orchestrating sequential computational algebra components into a coherent parallel program.

Computational algebra applications are typically constructed using domain-specific programming notations, executed using specialist runtime engines that have rarely been designed with parallelism in mind. Common commercial examples include Maple (Char et al., 1991),

Mathematica (1999) and MuPAD (Morisse and Kemper, 1994); while widely-used free examples include Kant (Daberkow et al., 1997) and GAP (The GAP Group, 2007). While many computational algebra applications are computationally intensive, and could, in principle, make good use of the array of modern parallel architectures including multicore and cluster machines, relatively few parallel implementations are available. Those that are available can be unreliable and difficult to use. Indeed, in at least one case of which we are aware (Cooperman, 1997; Martínez and Pena, 2004), the underlying CAS has been explicitly optimised to be single-threading, rendering parallelisation a major and daunting task. By providing an external mechanism that is capable of orchestrating individual sequential components into a coherent parallel program, we aim to facilitate the parallelisation of a variety of CASs.

The key advantages of our approach are:

- 1 by using external middleware, it is not necessary to change the sequential CAS in any way
- 2 we can exploit the same middleware for multiple CASs, so gaining benefits of scale
- 3 we can support heterogeneous applications, orchestrating components from more than one system
- 4 we can benefit from future advances in parallelisation, without needing to change the end-user application, and with minimal consideration in the application of how that parallelism is achieved.

The main disadvantages that we have identified are:

- 1 there is a potential loss of performance compared with hand-coded parallel applications
- 2 the parallel application depends on our external middleware.

We believe that the advantages of painless parallelism for the average user of a computational algebra system (CAS) more than outweigh these disadvantages. Moreover, as we shall see, because we are able to concentrate purely on parallelism aspects, the resulting performance can be highly competitive with even specialised parallel implementations.

This paper is structured as follows. We first briefly introduce the GAP CAS that we will use to develop our experimental applications (Section 2), and discuss the *SymGrid-Par* middleware for parallelising computational algebra systems, (Section 3). We then describe our experimental setup (Section 4) and consider results for three simple applications running on networked clusters (Section 5–Section 7) and a multicore machine (Section 8). Finally, we describe related work (Section 9) and conclude (Section 10).

This paper extends our ISPA08 paper (Al Zain et al., 2008b) and presents the first substantial parallel performance results for the *SymGrid-Par* GCA component. In particular, our results demonstrate:

- 1 *flexibility* – we parallelise three test problems that capture typical features of real computational algebra problems, including problems with varying levels of irregular parallelism (Section 5–Section 7)
- 2 *effectiveness* – we show good parallel performance for each program, both absolute performance and in comparison to an established parallel language, GpH (Trinder et al., 1998), and compare the performance of alternative parallel algorithms to solve the same algebraic problem (Section 6)
- 3 *portability* – we show that *SymGrid-Par* GCA can deliver good parallel performance on both parallel clusters and multicores (Section 8).

2 Computational algebra and GAP

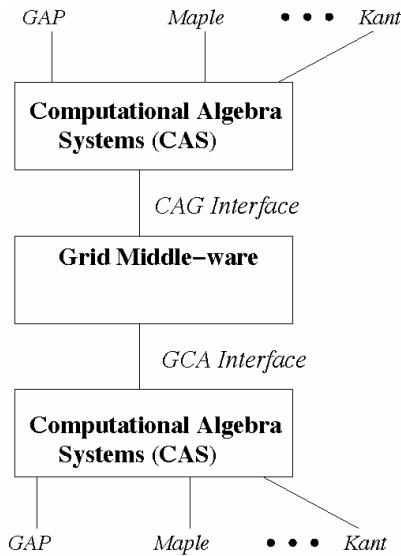
Computational algebra has played an important role in a number of notable mathematical developments, e.g., in the classification of finite simple groups. It is essential in several areas of mathematics which apply to computer science, such as formal languages, coding theory, or cryptography. Computational 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 that supports dynamic task allocation, load re-balancing and task migration GAP (The GAP Group, 2007) is a free 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.

3 The SymGrid-Par parallel middleware

The *SymGrid-Par* middleware orchestrates computational algebra components into a parallel application. *SymGrid-Par* components communicate using *Open-Math* (The OpenMath Format, 2007), an XML-based data description format, designed specifically to represent computational mathematical objects. A high performance computer algebra grid service is provided by an integration of *SymGrid-Par* within the *SymGrid* framework (Hammond et al., 2007). In this paper, we restrict our attention to parallel coordination on a single cluster or a multicore machine.

SymGrid-Par (Figure 1) is built around GUM (Trinder et al., 1996), the runtime implementation of Glasgow parallel Haskell (GpH). GpH is a well-established *semi-implicit* (Hammond and Michaelson, 1999) parallel extension to the standard non-strict purely functional language Haskell. GUM provides various high level parallelism services including support for ultralight-weight threads, virtual shared-memory management, scheduling support, automatic thread placement, automatic datatype-specific marshalling/unmarshalling, implicit communication, load-based thread throttling, and thread migration. It thus provides a flexible, adaptive environment for managing parallelism at various degrees of granularity. It has been ported to a variety of shared memory and distributed-memory parallel machines, and more recently (Al Zain et al., 2006, 2008a) to Globus-based computational grids using the grid-enabled MPICH-G2 implementation of the standard MPI communication library.

Figure 1 *SymGrid-Par* design



SymGrid-Par exploits the capabilities of the GUM system by layering a simple API over basic GUM functionality. It comprises two generic interfaces (described below): the CAG interface links CASs to GUM; and the GCA interface conversely links GUM to these systems. In this paper, we consider only the interfaces to/from GAP. Interfacing to other systems follows essentially the same pattern, however. The CAG interface is used by GAP to interact with GUM. GUM then uses the GCA interface to invoke remote GAP functions, to communicate with the GAP system etc. In this way, we achieve a clear separation of concerns: GUM deals with issues of thread creation/coordination and orchestrates the GAP engines to work on the application as a whole; while each instance of the GAP engine deals solely with execution of individual algebraic computations.

3.1 The CAG interface

The CAG interface comprises an API for each symbolic system that provides access to a set of common (and potentially parallel) patterns of symbolic computation. These patterns form a set of dynamic *algorithmic skeletons* (Cole, 1999), which may be called directly from within the CAS, and which may be used to orchestrate a set of sequential components into a parallel computation. In general (and unlike most skeleton approaches), these patterns will be nested and can be dynamically composed to form the required parallel computation. Also, in general, they may mix components taken from several different CASs.

We can identify two classes of pattern: standard patterns that apply both to computational algebra problems and to other problem types; and domain-specific patterns that arise particularly when dealing with computational algebra problems.

3.1.1 Standard parallel patterns

The standard patterns we have identified are listed below. The patterns are based on commonly-used sequential higher-order functions that can be found in functional languages such as Haskell. Similar patterns are often defined as algorithmic skeletons. Here, each argument to the pattern is separated by an arrow (\rightarrow), and may operate over lists of values ($[\dots]$), or pairs of values ((\dots)). All of the patterns are *polymorphic*: i.e., a , b etc. stand for (possibly different) concrete types. The first argument in each case is a function of either one or two arguments that is to be applied in parallel.

```

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]

```

So, e.g., `parMap` is a pattern taking two arguments and returning one result. Its first argument (of type $a \rightarrow b$) is a function from some type a to some other type b , and its second argument (of type $[a]$) is a list of values of type a . It returns a list of values each of type b . Operationally, `parMap` applies a function argument to each element of a list, in parallel, returning the list of results, e.g.,

```

parMap double [1, 4, 9, 16] == [2, 8, 18, 32]
where double x = x + x

```

It thus implements a parallel version of the commonly found `map` function, which applies a function to each element of list. Such patterns are commonly found in parallel applications, and we will therefore make extensive use of the `parMap` pattern to orchestrate the examples presented in this paper.

The `parZipWith` pattern similarly applies a function, but in this case to two arguments, one taken from each of its list arguments. Each application is performed in parallel. For example:

```
parZipWith add [1, 4, 9, 16] [3, 5, 7, 9] ==
[4, 9, 16, 25]
where ad x y = x + y
```

Again, this implements a parallel version of the `zipWith` function that is found in Haskell and some other functional languages. Finally, `parReduce` reduces its third argument (a list) by applying a function between pairs of elements, ending with the value supplied as its second argument, `parMapReduce` combines features of both `parMap` and `parReduce`, and `masterSlaves` is used to introduce a set of tasks, each of which is under the control of a coordinating master task. The `parReduce` and `parMapReduce` patterns are often used to construct parallel pipelines, where the elements of the list will themselves be lists, perhaps constructed using other parallel patterns. In this way, we can achieve nested parallelism.

3.1.2 Domain-specific parallel patterns

We have identified a number of new domain-specific patterns that may arise in a variety of computational algebraic problems, and incorporated these into the design of *SymGrid-Par*. These patterns include:

- 1 *Orbit calculation*: Generate unprocessed neighbouring states from a task queue of states to process.
- 2 *Duplicate elimination*: Given two lists of values, merge them while eliminating duplicates.
- 3 *Completion algorithm*: Given a set of objects to process, generate new objects from any pair of objects, and then reduce these new objects against the existing objects. Continue until no new objects can be found.

Each of these patterns produces highly parallel computations, with complex irregular task sizes, and mixes of task and data-parallelism.

3.2 The GCA interface

The GCA interface (Figure 2) interfaces with GAP, connecting to a small interpreter that allows the invocation of arbitrary CAS functions, marshalling and unmarshalling data as required. The interface comprises both C and Haskell components. The C component is mainly used to invoke operating system services that are needed to initiate the GAP process, to establish communication channels, and to send and receive commands/results from the GAP process. It also provides support for static memory that can be used to maintain state between calls. The Haskell component provides interface functions to the user program and implements the communication protocol with the GAP process. The main GpH functions are:

```
casEval      :: String -> [casObject]
              -> casObject

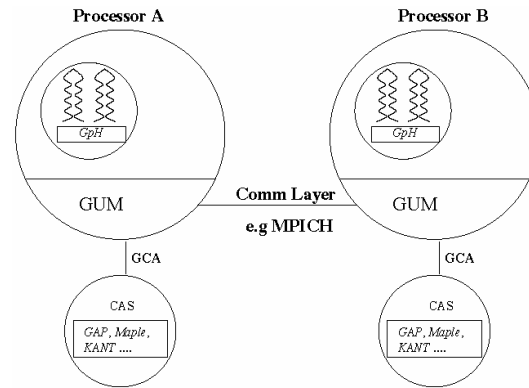
casEvalN     :: String -> [casObject]
              -> casObject

string2CASExpr :: String -> casObject

casExpr2String :: casObject -> String
```

Here, `casEval` and `casEvalN` allow GpH programs to invoke GAP functions by giving a function name plus a list of parameters as `gapObjects`; `gapEvalN` is used to invoke GAP functions that return more than one object; while `string2GAPExpr` and `gapExpr2String` convert GAP objects to/from internal GpH data formats.

Figure 2 The GCA interface



4 Experimental setup

We have implemented *SymGrid-Par* as described above for Intel/AMD machines running MPI under Linux. In this paper, we measure the performance of our implementation on two different systems (Table 1):

- 1 a 28-node Beowulf cluster, located at Heriot-Watt university (*bwlf*)
- 2 a new eight-core Dell Poweredge 2950 machine located at the University of St Andrews (*ardbeg*), constructed from two quad-core Intel Xeon 5355 processors.

Table 1 Experimental architectures

	Phys node	CPU		Archit
		MHz	Cache	
bwlf	28 × 1	3,008	1,024 Kb	Intel Pentium IV
ardbeg	1 × 8	2,660	64 KB/4,096 KB	Intel Xeon5355

Nodes on the Beowulf cluster are connected using 100 Mb/s Ethernet. Each node has a 533 MHz front-side bus, and 512 MB of standard DIMMs. All nodes run the same version of Fedora Linux (kernel version 2.6.10-1). The Dell Poweredge has a 1333MHz front-side bus, and 16 GB of

fully-buffered 667 MHz DIMMs. It runs CentOS Linux 4.5 (kernel version 2.6.9–55).

We measure three testbed parallel programs (Table 2). Fibonacci is a simple benchmark that computes Fibonacci numbers (Section 5). The *sum-Euler* program is a more realistic example that computes the sum of applying the Euler totient function to an integer list (Section 6). Finally, the *smallGroup* program is a real problem that determines whether the average order of the elements of a mathematical group is an integer (Section 7). In order to help reduce the impact of operating system and other system effects, all runtimes given below are taken as the mean of three wall-clock times.

Table 2 Program characteristics

Program	App area	Parad	Reg	Source lines of code	
				GpH	Gap
parFib	Numeric	Div-Conq.	Regul	22	14
sum-Euler	Numerical analysis	Data Par.	Irreg.	31	20
smallGroup	Symbolic algebra	Data Par.	v. high irreg.	28	24

5 Ideal example: Fibonacci

The first step in validating the GCA-GAP design is to demonstrate that it can efficiently and effectively parallelise simple programs with good parallel behaviour. That is, we show that GAP and GUM can interact correctly and with limited overhead, and that it is possible to orchestrate sequential GAP components into a coherent and efficient parallel program. The parallel Fibonacci function has previously been shown to deliver excellent parallel performance under both GpH and other languages (Loidl et al., 1999).

Figure 3 Runtime and speedup curves for *parFib* 50, threshold 35

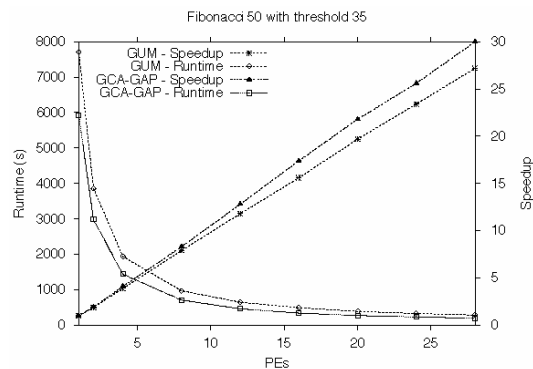


Figure 3 shows the speedup and runtime curves for GCA-GAP and GpH Fibonacci implementations. It shows that GCA-GAP delivers marginally super-linear speedup for this ideal parallel application, giving a maximum speedup of 30 on 28 PEs. As the sequential *parFib* 35 is faster in GAP than GpH, it follows also GCA-GAP is faster than GpH on a single PE: 5,921 s vs. 7,704 s. Moreover, the performance advantage scales well on the number of processors we have tried here. The modest super-linearity in the GCA-GAP speedups can be attributed to reduced memory management costs when the problem is split into several smaller parts.

6 A non-trivial example: sum-Euler

6.1 Problem outline and implementations

The *sum-Euler* program computes the sum of applying the Euler totient function to the elements of an integer list. Figure 4 shows the GpH code that calculates the sum of the Euler totients for some range of numbers. The main function, `sumTotient`, generates a list of integers between the specified lower and upper limits. This is split into chunks of size `c` using the `splitAtN` function. The `euler` function is mapped in parallel across each chunk using `parMap`, then the result summed sequentially for each chunk. Finally, the sum can be determined for all chunks. This gives a data parallel implementation, with a fairly cheap combination phase involving only a small amount of communication. Figure 5 shows the corresponding GCA-GAP implementation, which calls the GAP function `euler`. This function generates a list from one to n , selecting only those elements of the list that are prime relative to n . It returns a count of the number of these relatively prime elements. It uses the auxiliary `relprime` function, which returns true if its arguments are relatively prime, i.e., the highest common factor of the two arguments is one. Despite its use of data parallelism, *sum-Euler* gives rise to highly irregular parallelism during execution, since task granularity can vary significantly depending on input parameter values.

Figure 4 sum-Euler: GpH

```

sumTotient :: Int -> Int -> Int -> Int
sumTotient :: lower upper c = sum (parMap
  (euler) (plitAtN c [lower .. upper]))

euler :: Int -> Int
euler n = length (filter (relprime n)
  [1 .. n-1])

relprime :: Int -> Int Bool
relprime x y = hcf x y == 1

```

Figure 5 *sum-Euler*: GCA-GAP

```

sumTotientGAP :: Int-> Int-> Int-> Int
sumTotientGAP lower upper c = sum(parMap
(eulerGAP) (splitAt n c [lower .. upper]))

eulerGAP :: Int-> Int
eulerGAP n = gapObject2Int(gapEval "euler"
[int2GAPObject n])

```

We have defined two versions of the *euler* function in GAP. Figure 6, shows a naïve recursive implementation, and Figure 7, shows the more usual direct implementation. In the recursive implementation, which is a direct translation of the GpH code, the *relprime* function calls the *hcf* function to calculate the highest common factor of *x* and *y* recursively. In the direct implementation, *relprime* instead uses the highly optimised GAP function *GcdInt* to identify arguments which are relatively prime.

Figure 6 Recursive *euler* in GAP

```

hcf := function(x, y)
  local m;
  if y=0 then return x;
  else m:= x mod y; return hcf(y, m); if;
end;;

relprime := function(x, y)
  local x;
  m := hcf(x, y); return m=1;
end;;

euler := function (n)
  local x;
  x := Number(Filtered([1 .. n],
x->relprime(x, n)));
  return x;
end;;

```

Figure 7 Direct *euler* in GAP

```

relprime := function(x, y)
  local m;
  m := GcdInt(x, y); return m=1;
end;;

euler := function(n)
  local x;
  x := Number(Filtered([1..n])),
x-> relprime(x, n));
  return x;
end;;

```

6.2 Results for *sum-Euler*

Table 3 shows sequential results for *sum-Euler*. For this example, we can see that the GpH/GUM implementation is significantly more efficient than either of the GAP implementations, and that the direct GAP solution is significantly faster than the recursive implementation. Overall, the GpH/GUM program is a factor of two to three times faster than the direct GAP program, and a factor of eight to 17 times faster than the recursive GAP version.

Table 3 Sequential

	<i>sum-Euler</i>	
	200	32,000
GpH/GUM	0.006 s	164 s
GAP direct	0.011 s	500 s
GAP recursive	0.05 s	2,928 s

Table 4 *sum-Euler* runtime and speedup 32,000

PE	GCA-GAP			GUM	
	<i>Dir rt</i>	<i>Spd</i>	<i>Rcr rt</i>	<i>rt</i>	<i>Spd</i>
1	3,006 s	1	500 s	164 s	1.0
2	1,139 s	2.6	320 s	121 s	1.3
4	542 s	5.5	154 s	69 s	2.3
6	365 s	8.2	107 s	45 s	3.6
8	267 s	11.2	84 s	38 s	4.3
12	174 s	17.2	59 s	39 s	4.2
16	141 s	21.3	51 s	35 s	4.6
20	115 s	26.1	45 s	30 s	5.4
28	95 s	31.6	40 s	23 s	7.1

Table 5 *sum-Euler* runtime and speedup: 80,000

PE	GCA-GAP		GUM	
	<i>Dir rt</i>	<i>Spd</i>	<i>rt</i>	<i>Spd</i>
1	3,265 s	1	1,088 s	1
2	1,677 s	1.9	633 s	1.7
4	818 s	3.9	330 s	3.3
8	406 s	8.0	165 s	6.6
12	277 s	11.7	126 s	8.6
16	207 s	15.7	95 s	11.4
20	183 s	17.8	78 s	13.9
24	159 s	20.5	76 s	14.3
28	139 s	23.4	76 s	14.3

Table 4 shows the performance of *sum-Euler* for arguments ranging between one and 32,000 on the *bwlf* cluster in Table 1. The first column shows the number of PEs; the second and third columns show runtime and speedup for GCA-GAP using the direct implementation of *euler*; the fourth and fifth columns show runtimes and speedups for GCA-GAP using the recursive implementation of *euler*, and the last two columns show runtimes and speedups for GUM. Table 5 shows the corresponding performance for

arguments ranging between one and 80,000. In this case, no results could be recorded for the recursive implementation and these figures are therefore omitted.

6.2.1 The recursive GCA-GAP algorithm

Table 4 shows that the recursive GCA-GAP algorithm delivers near-linear speedup, yielding a maximum speedup of 31.6 on 28 PEs. Despite this, it is still slower than GUM by a factor of 18.3 on one PE and a factor of 4.1 on 28 PEs. This difference in performance can be attributed to the lack of memory management optimisations for recursive programming in GAP (Linton and Kononov, 2007; Cooperman, 2001). Moreover, the `mod` operator, which is used in the `hcf` function, is very memory-intensive in GAP. In contrast, the GUM memory management and garbage collectors are better optimised for recursion (Loidl and Trinder, 1997). The modest super-linearity in the GCA-GAP program can again be attributed to reduced memory management costs.

6.2.2 The direct GCA-GAP algorithm

From Table 4, we can see that the direct algorithm yields some, but not exceptional, parallel performance. We observe a speedup of 1.5 on 2 PEs and 12.5 on 28 PEs. Although the direct algorithm displays worse speedup than its recursive counterpart, it is faster by a factor of six on two PEs and by a factor of 2.3 on 28 PEs. It is clear that, for this example, the direct algorithm is implemented more efficiently by the GAP kernel. Although, the direct algorithm delivers better *speedup* than the GUM algorithm (Table 4), GUM is still faster by a factor of three on one PE and by a factor of 1.7 on 28 PEs. This is mainly a consequence of the highly optimised sequential Haskell implementation that has been exploited by GpH, though marshalling/unmarshalling adds some overhead to the GCAGAP performance. The poor speedup observed on 28 PEs with an input of 32,000 for both GUM and the direct GCA-GAP implementation is largely a consequence of the problem size. Increasing the input size to 80,000 (Table 5) improves the speedup by almost a factor of two in each case, to 14.3 and 23.4 on 28 PEs, respectively.

7 Multi-level irregular example: *smallGroup*

We now apply GCA-GAP to a real computational algebra problem that exhibits two levels of irregularity, together with the potential for nested parallelism, *smallGroup*.

7.1 Problem outline and implementations

The *smallGroup* program searches for mathematical *groups* whose order is not greater than a given constant, n , that have some specific property. In the case of the problem we have chosen to study, the property is that the average order of

their elements is an integer. This example provides two levels of irregularity:

- firstly, as shown by Figure 8, the number of groups of a given order varies enormously, i.e., by *five orders of magnitude*
- secondly, there are variations in the cost of computing the prime power of each group.

Figure 8 Number of groups of a given order from one to 400

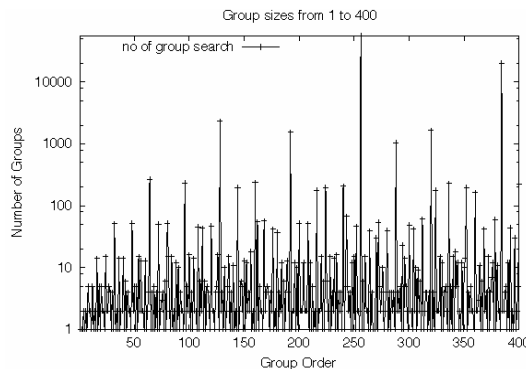


Figure 9 GCA: *smallGroup* search code

```

smallGroupSearch :: Int -> Int [(Int,Int)]
smallGroupSearch lo hi =
  concat (map (ifmatch) (predSmallGroup
    [lo..hi]))

predSmallGroup :: ((Int,Int) ->
  (Int,Int,Bool)) -> Int -> [(Int,Int)]
predSmallGroup (i,n) =
  (i,n,(gapObject2String
    (gapEval ``IntAvgOrder`` [int2GapObject
      n,
      int2GapObject i])) == ``true``)

ifmatch :: ((Int,Int) -> (Int,Int,Bool)) ->
  Int -> [(Int, Int)]
ifmatch predSmallGroup n = [(i,n) |
  (i,n,b) <-
    (masterSlaves predSmallGroup [(i,n) |
      i<-
        [1 nrSmallGroups n]),b]

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

```

The kernel of the *smallGroup* program is shown in Figure 9 and Figure 10. There are two obvious places to introduce data parallelism:

- the `smallGroupSearch` function generates a list of integers between a low value (`lo`) and a high value (`hi`), and sequentially applies `predSmallGroup` to each integer.
- the `ifmatch` function relies on the `masterSlaves` skeleton (Berthold et al., 2007) to generate a set of hierarchical master worker tasks to calculate `IntAvgOrder` in GAP.

Figure 10 GAP: *smallGroup* search code

```

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]);
      if;
    od;
  od;
  return hits
end;

```

7.2 Single level irregularity: one group order

Our first experiment studies GCA-GAP performance with a single level of irregularity, i.e., by computing the property for a single group order, i.e., introduces tasks for each of the 56,092 groups generated for $n = 256$. Table 6 shows the results of evaluating *smallGroup 256* in parallel. The first column shows the number of PEs; the second and third columns show runtimes and speedups for GCA-GAP and the final column shows the speedup over the sequential GAP implementation. We observe good parallel performance, with a relative speedup of 26.7 on 28 PEs (95% efficiency), and even better absolute speedup over sequential GAP. As with the *sum-Euler* example, by assigning memory management and coordination aspects to

GpH, we are able to outperform sequential GAP even on a single processor, requiring only 829 s for the single-PE GpH execution, versus 913 s for the sequential GAP execution.

Table 6 *smallGroup* [256 .. 256]

PE	GCA-GPA	Spd	Spd (GAP)
1	829 s	1	1.1
2	416 s	1.9	2.1
4	206 s	4.0	4.4
8	104 s	7.9	8.7
12	70 s	11.8	13.0
16	53 s	15.6	17.2
20	42 s	19.7	21.7
24	36 s	23.0	25.3
28	31 s	26.7	29.4

7.3 Multi-level irregularity: ranges of group orders

Our second experiment investigates multi-level irregularity, where the outer level applies the *predSmallGroup* to a sequence of group orders, and the inner level then generates worker tasks for each element of the sequence. To demonstrate repeatability, we consider two sets of inputs, for orders in the ranges one to 400 and 600 to 1,000, respectively.

7.3.1 Results for *smallGroup* [1 ... 400]

Table 7 shows the results of computing *smallGroup* for orders between one and 400, where a total of 87,927 candidate small groups are considered. For these inputs, the sequential GAP program takes 1,658 s. We observe good parallel performance: GCA-GAP shows a relative speedup of 1.9 on 2 PEs (representing 95% efficiency), and 18.8 on 28 PEs (67% efficiency), and absolute speedup over sequential GAP of up to 22.7 on 28 PEs. The loss of efficiency can be attributed to the nature of the *smallGroup* program, which is a challenging parallel application with two levels of parallelism and highly irregular task granularity.

Table 7 *smallGroup* [1 ... 400]

PE	GCA-GPA	Spd	Spd (GAP)
1	1,377 s	1	1.2
2	698 s	1.9	2.3
4	360 s	3.8	4.6
6	243 s	5.6	6.8
8	186 s	7.4	8.9
12	132 s	10.4	12.5
16	105 s	13.1	15.7
20	89 s	15.4	18.6
28	73 s	18.8	22.7

7.3.2 Results for *smallGroup* [600 ... 1000]

Table 8 similarly shows the results of computing *smallGroup* for orders between 600 and 1,000, where a total of 1,163,006 candidate small groups are considered. For these inputs, the sequential GAP program takes 273,874 s. The results shown here confirm those in the previous section: GCA-GAP continues to provide significant parallel performance on the larger input sizes considered here. Despite the irregularity of the *smallGroup* application, we observe super-linear speedup of a factor of 29.2 on 28 PEs. As with the *sum-Euler* example of Section 6, this is a consequence of the memory management/garbage collection system used in the sequential implementation.

Table 8 *smallGroup* [600 ... 1000]

PE	GCA-GPA	Spd	Spd (GAP)
1	239,121 s	1	1.1
4	63,296 s	3.7	4.3
8	30,929 s	7.7	8.8
16	15,049 s	15.8	18.1
28	8,179 s	29.2	33.4

8 Multicore results for *smallGroup*

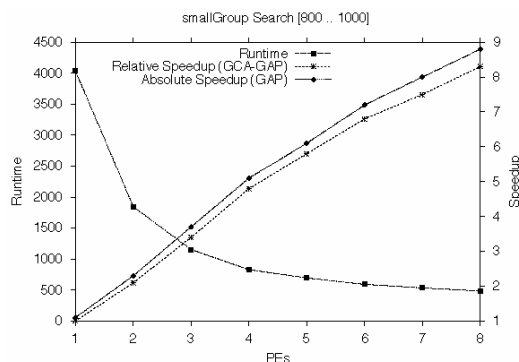
Multicore architectures are becoming increasingly common: quad-core Intel processor sets based on Pentium IV cores are now available, and it is possible to purchase off-the-shelf systems containing eight Intel Pentium IV cores, with 16-core machines recently announced. Clearly, such systems are likely to replace not only commodity shared-memory machines (which may indeed now comprise several multicore processors), but also traditional desktop processors. They thus represent an important emerging target architecture for users of computation algebra (and indeed, other) systems. Effectively managing parallelism for such systems has proved to be an interesting challenge, however. This section investigates the performance of GCA-GAP on an eight-core Dell Poweredge system, built from two quad-core Intel Xeon 5355 processors (*ardbeg*, see Table 1, page 4). We consider only the performance of the *smallGroup* program, since this represents the most serious of the applications we have previously studied in this paper.

8.1 Results for *smallGroup* on an eight-core machine

Figure 11 shows the runtime and speedup curves for computing *smallGroup* between 800 and 1,000 on an eight-core machine. A total of 42,473 candidate small groups are considered. The results show that a slightly super-linear speedup is possible on eight cores, with a real absolute speedup of a factor of 8.8 over sequential GAP. As anticipated, our results show better scalability for GCA-GAP on the multicore system than on the Beowulf cluster. This is mainly due to the low communication costs for the multicore architecture compared with the distributed

machine. Moreover, the ultralight-weight threads that are available in GpH make it more suitable for use in multicore architecture (Trinder et al., 2000; Harris et al., 2005). We conclude that our approach is capable of producing good performance results on multicore architectures as well as clusters, and that this performance is, in general, likely to be superior for a multicore machine of a given size.

Figure 11 *smallGroup* [800 ... 1000], on an eight-core machine



9 Related work

9.1 Parallel symbolic computation

Work on parallel symbolic computation dates back to at least the early 1990s – Roch and Villard (1997) provide a good general survey of early research. Within this general area, significant research has been undertaken for parallelise specific computational algebra algorithms, notably term re-writing and Gröbner basis completion (e.g., Amrhein et al., 1996). A number of one-off parallel programs have also been developed for specific algebraic computations, mainly in representation theory (Michler, 1998).

However, while several symbolic computation systems include some form of operator to introduce parallelism parallel Maple (Bernardin, 1997), or parallel GAP (Cooperman, 1997), very few production parallel algorithms have been produced. This is partly due to the complexities involved in programming such algorithms using explicit parallelism and partly due to the lack of generalised support for communication, distribution etc, in these systems. By abstracting over such issues, by providing system-independent orchestration of parallel programs, we anticipate that *SymGrid-Par* will considerably simplify the construction of parallel computational algebra computations.

9.2 Parallel functional languages and computer algebra systems

Include, e.g., the GHC-Maple interface and the Eden-Maple system (Martínez and Pena, 2004). None of these systems is in widespread use at present, none supports the broad range of computational algebra applications we are targeting, nor

has the support of the developers of those systems, none has such an ambitious goal in terms of orchestrating legacy sequential components, and none has achieved the results on both multicore and cluster systems reported here.

9.3 Orchestrating services

Over the last 20 years there has been a great interest in orchestrating services, e.g., *grid services*, such as job submission, data transfer and data portal services. A number of environments have been developed, both commercially e.g., FlowMark (Leymann and Roller, 1994) BPEL (Emmerich et al., 2005), and for research, e.g., DAGMan (Frey, 2002), and GridAnt (Amin et al., 2004). *SymGrid-Par* orchestrates heterogeneous computations across high-performance computational Grid environments, rather than services. Moreover, *SymGrid-Par* targets both commercial applications (Maple, Mathematica, MuPAD) and research/academic applications (Kant, GAP).

10 Conclusions and future work

We have outlined *SymGrid-Par*, a system that orchestrates legacy sequential computational algebra components into a high-performance parallel application. The CASs we coordinate are large and complex, utilising specialised data structures and algorithms. Moreover, they are symbolic in nature rather than the more commonly studied numerical applications.

The primary research contribution of the paper is to demonstrate that it is possible to obtain good, scalable parallel performance by coordinating *unaltered* computer algebra system instances. We present the first substantial parallel performance results for the *SymGrid-Par* GCA component, restricted to a single CAS, GAP. We show the *flexibility* of *SymGrid-Par* by parallelising three typical algebraic computations, including the *smallGroup* problem with multiple levels of extremely irregular parallelism where problem sizes may vary by five orders of magnitude. We show the *effectiveness* of the architecture by demonstrating good parallel performance for each program, achieving relative speedups of between 12.5 and 31.6 on a 28-node cluster, and up to 8.3 on the eight-core machine. We further compare GCA parallel performance with an established parallel language, GpH (Trinder et al., 1998). We further compare the performance of alternative parallel algorithms to solve the same algebraic problem. We show the *portability* of *SymGrid-Par* by showing that GCA can deliver good parallel performance on two common commodity architectures – an homogeneous cluster and an eight-core system.

We must now extend our results to cover a wider variety of CASs. The implementers of the Maple, Kant and MuPAD systems are partners in the SCIENCE project, and we will shortly integrate them into *SymGrid-Par*. We anticipate delivering similar parallel performance to the users of these other CASs, again *without needing to alter the stable, reliable and widely-used sequential kernels of*

these systems. The benefits of lightweight orchestration through GpH are clear: it is possible to achieve good parallel performance without needing major system rewrites. This is a major gain for developers of complex legacy systems wishing to take rapid and straightforward advantage of the upcoming availability of cheap commodity parallel hardware.

Acknowledgements

We would like to thank Hans-Wolfgang Loidl for his constructive comments on a previous draft of this paper, Steve Linton and Alexander Kononov for computational algebra expertise.

This research is partially supported by European Union Framework 6 grant RII3-CT-2005-026133 SCIENCE: Symbolic Computing Infrastructure in Europe.

References

- Al Zain, A., Trinder, P., Loidl, H-W. and Michaelson, G. (2006) 'Managing heterogeneity in a grid parallel Haskell', *J. Scalable Comp.: Practice and Experience*, Vol. 7, No. 3, pp.9–25.
- Al Zain, A., Trinder, P., Loidl, H-W. and Michaelson, G. (2008a) 'Evaluating a high-level parallel language (GpH) for computational GRIDS', *IEEE Transactions on Parallel and Distributed Systems*, Vol. 19, No. 2, pp.219–233.
- Al Zain, A.D., Trinder, P.W., Hammond, K., Kononov, A. and Linton, S. (2008b) 'Parallelism without pain: orchestrating computational algebra components into a high performance parallel system', in *IEEE International Symposium on Parallel and Distributed Processing with Application (ISPA08)*, IEEE Computer Society, December, No. P3471, pp.99–112, Sydney, Australia.
- Amin, K., Laszewski, G.V., Hategan, M., Zaluzec, N.J., Hampton, S. and Rossi, A. (2004) 'GridAnt: a client-controllable grid workflow system', in *HICSS '04*, IEEE Computer Society, Washington, DC, USA.
- Amrhein, B., Gloor, O. and Küchlin, W. (1996) 'A case study of multithreaded Gröbner basis completion', in *Proc. ISSAC '96: International Symposium on Symbolic and Algebraic Computation*, ACM Press, pp.95–102.
- Bernardin, L. (1997) 'Maple on a massively parallel, distributed memory machine', in *Proc. PASC0 '97: Intl. Symp. on Parallel Symbolic Computation*, ACM Press, pp.217–222.
- Berthold, J., Dieterle, M., Loogen, R. and Priebe, S. (2007) 'Hierarchical master-worker skeletons', in *TFP'07, Draft Proceedings*, New York, USA.
- Char, B.W., et al. (1991) *Maple V Language Reference Manual*, Maple Publishing, Waterloo Canada.
- Cole, M.I. (1999) 'Algorithmic skeletons', in Hammond, K. and Michaelson, G. (Eds.): *Research Directions in Parallel Functional Programming*, Springer-Verlag, Chapter 13, pp.289–304.
- Cooperman, G. (1997) 'GAP/MPI: facilitating parallelism', in *Proc. DIMACS Workshop on Groups and Computation II*, AMS, Vol. 28, pp.69–84.

- Cooperman, G. (2001) 'Parallel GAP: Mature interactive parallel', *Groups and Computation, III (Columbus, OH, 1999)*, de Gruyter, Berlin.
- Daberkow, M., Fieker, C., Klüners, J., Pohst, M., Roegner, K., Schömgig, M. and Wildanger, K. (1997) 'Kant v4', *J. Symb. Comput.*, Vol. 24, Nos. 3/4, pp.267–283.
- Emmerich, W., Butchart, B., Chen, L., Wassermann, B. and Price, S.L. (2005) 'Grid service orchestration using the business process execution language (bpel)', *Journal of Grid Computing*, Vol. 3, Nos. 3–4, pp.283–304.
- Frey, J. (2002) 'Condor Dagman: handling inter-job dependencies', Technical report, University of Wisconsin, Department of Computer Science.
- Hammond, K. and Michaelson, G. (1999) *Research Directions in Parallel Functional Programming*, Chapter Introduction, Springer-Verlag.
- Hammond, K., Al Zain, A., Cooperman, G., Petcu, D. and Trinder, P. (2007) 'SymGrid: a framework for symbolic computation on the grid', in *Proc. EuroPar'07 – European Conference on Parallel Processing*, LNCS, Rennes, France, Springer, to appear.
- Harris, T., Marlow, S. and Peyton Jones, S. (2005) 'Haskell on a shared-memory multiprocessor', in *Proc. Haskell '05: 2005 ACM SIGPLAN Workshop on Haskell*, ACM Press, September, pp.49–61.
- Leymann, F. and Roller, D. (1994) 'Business process management with flowmark', in *Comcon Spring'94*, IEEE Computer Society, pp.230–234.
- Linton, S.A. and Konovalov, A. (2007) 'Gap memory management and recursive implementation', Technical discussion, February, Attendee: Trinder, P.W. and Hammond, K., St Andrews, UK, available at <http://www.gap-system.org/gap>.
- Loidl, H-W. and Trinder, P.W. (1997) 'Engineering large parallel functional programs', in *Implementation of Functional Languages*, September, LNCS, St. Andrews, Scotland, Springer.
- Loidl, H-W., Trinder, P.W., Hammond, K., Junaidu, S.B., Morgan, R.G. and Peyton Jones, S.L. (1999) 'Engineering parallel symbolic programs in GPH', *Concurrency – Practice and Experience*, Vol. 11, pp.701–752.
- Martínez, R. and Pena, R. (2004) 'Building an interface between Eden and Maple', in *Proc. IFL 2003*, Springer-Verlag LNCS 3145, pp.135–151.
- Michler, G.O. (1998) *High Performance Computations in Group Representation Theory*, Preprint, Institut für Experimentelle Mathematik, Universität GH Essen.
- Morisse, K. and Kemper, A. (1994) 'The computer algebra system MuPAD', *Euromath Bulletin*, Vol. 1, No. 2, pp.95–102.
- Roch, L. and Villard, G. (1997) 'Parallel computer algebra', in *Proc. ISSAC '97: International Symposium on Symbolic and Algebraic Computation*, Preprint IMAG Grenoble France.
- The GAP Group (2007) 'GAP – groups, algorithms, and programming', available at <http://www.gap-system.org/gap>.
- The Mathematica (1999) Wolfram Media, Inc., Champaign, IL.
- The OpenMath Format (2007) available at <http://www.openmath.org/>.
- Trinder, P.W., Hammond, K., Loidl, H-W. and Peyton Jones, S.L. (1998) 'Algorithm + strategy = parallelism', *J. Functional Programming*, January, Vol. 8, No. 1, pp.23–60.
- Trinder, P.W., Hammond, K., Mattson, J.S. Jr., Partridge, A.S. and Peyton Jones, S.L. (1996) 'GUM: a portable parallel implementation of Haskell', in *Proc. PLDI'96*, May, pp.79–88, Philadelphia, PA, USA.
- Trinder, P.W., Loidl, H-W., Barry, E. Jr., Hammond, K., Klusik, U., Peyton Jones, S.L. and Rebón Portillo, Á.J. (2000) 'The multi-architecture performance of the parallel functional language GPH', in *Euro-Par 2000 – Parallel Processing*, LNCS, pp.739–743, Springer-Verlag, Munich, Germany.