Orchestrating Production Computer Algebra Components into Portable Parallel Programs

Abdallah Al Zain[1], Jost Berthold[3], Kevin Hammond[2], and Phil Trinder[1]

[1] School of Mathematics and Computer Sci., Heriot-Watt University, Edinburgh, UK
email: {ceeatia,trinder}@macs.hw.ac.uk

[2] School of Computer Science, University of St Andrews, St Andrews, UK.
email: kh@cs.st-and.ac.uk

[3] Fachbereich Mathematik und Informatik, Philipps-Universität Marburg, Germany
email: berthold@Mathematik.Uni-Marburg.de

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. For now, we illustrate our approach with a single, *unmodified* production computational algebra system, GAP, running on two common commodity architectures — a homogeneous cluster and an eight-core system.
Computational algebra applications are large, specialised, and symbolic, rather than the more commonly studied numerical applications. They also exhibit high levels of irregularity, and multiple levels of irregularity. We demonstrate that for three small but representative algebraic computations, good parallel speedup is possible relative to a sequential GAP system running on a single processor/core. We compare the performance of the orchestrated system with that of parGAP, an established parallel implementation of GAP, demonstrating

Keywords: Parallel Coordination, Orchestration, Computational Algebra, GAP, Functional Programming, Haskell.

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 [9], Mathematica [1] and MuPAD [25]; while widely-used free examples include Kant [12] and GAP [15]. 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 [10, 23], the underlying computational algebra system 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 computational algebra systems.

This paper is structured as follows. We first briefly introduce the GAP computational algebra system 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 (Sections 5–7) and a multicore machine (Section 8). Finally, we describe related work (Section 9) and conclude (Section 10).

This paper 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 (Sections 5–7);
2. *effectiveness* – we show good parallel performance for each program, both absolute performance and in comparison to an established parallel language, GpH [27], and compare the performance of alternative parallel algorithms to solve the same algebraic problem (Section 6); and
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, 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

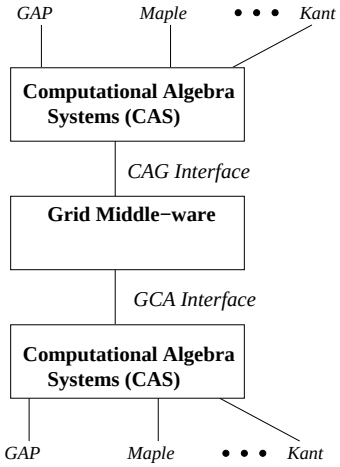

Fig. 1. SymGrid-Par Design

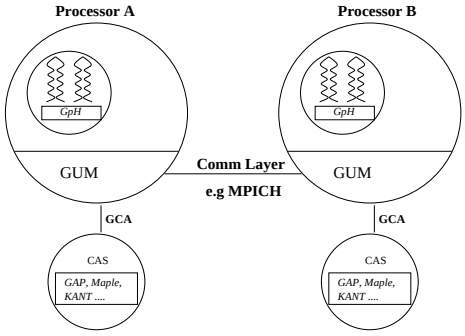

Fig. 2. The GCA Interface

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 rebalancing and task migration GAP [15] 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 **OpenMath** [2], 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 [16]. 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 [28], the runtime implementation of Glasgow Parallel Haskell (GPH). GPH is a a well-established *semi-implicit* [17] parallel extension to the standard non-strict purely functional language Haskell. GUM provides 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, 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 [3, 4] to Globus-based computational Grids using the Grid-enabled MPICH-G2 implementation of the standard MPI communication library.

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 computational algebra systems (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 consists of an API implementing a set of common patterns of symbolic computation, which are potentially amenable to parallel execution. The CAG interface supports these patterns as a set of dynamic *algorithmic skeletons* which may be called directly from within the computational steering interface, and which will consequently be used to orchestrate sequential GAP components into parallel computations.

3.2 The GCA Interface

The GCA interface (Figure 2) interfaces GUM 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:

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

Here, *gapEval* and *gapEvalN* allow GPH programs to invoke GAP functions by giving a function name plus a list of parameters as *gapObject*s; *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.

	Phys node	CPU MHz	CPU Cache	Archit
bwlf	28x1	3008	1024KB	Intel PentiumIV
ardbeg	1x8	2660	64KB/ 4096KB	Intel Xeon5355

Table 1. Experimental Architectures

Program	App Area	Parad	Reg	Source Lines of Code GpH	Source Lines of Code 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

Table 2. Program Characteristics

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): i) a 28-node Beowulf cluster, located at Heriot-Watt university (*bwlf*); and ii) 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. Nodes on the Beowulf cluster are connected using 100Mb/s Ethernet. Each node has a 533MHz front-side bus, and 512MB 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 16GB of fully-buffered 667MHz 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.

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 [22].

Figure 3 shows the speedup and run-time 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

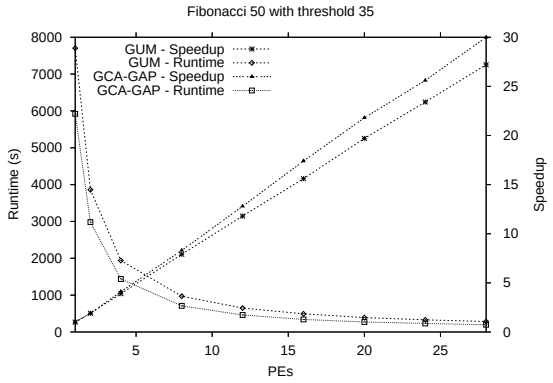

Fig. 3. Runtime and speedup curves for *parFib* 50, threshold 35

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 1 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 1. 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.

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.

```
sumTotient :: Int-> Int-> Int-> Int
sumTotient :lower upper c =
   sum (parMap (euler)
     (splitAtN 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
```

Fig. 4. *sum-Euler*: GpH

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

eulerGAP :: Int -> Int
eulerGAP n = gapObject2Int(gapEval ‘‘euler’’
             [int2GAPObject n])
```

Fig. 5. *sum-Euler*: GCA-GAP

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 2-3 times faster than the direct GAP program, and a factor of 8-17 times faster than the recursive GAP version.

Table 4 shows the performance of *sum-Euler* for arguments ranging between 1 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 1 and 80,000. In this case, no results could be recorded for the recursive implementation and these figures are therefore omitted.

The Recursive GCA-GAP Algorithm Table 4 shows that the recursive GCA-GAP algorithm delivers near-linear speedup, yielding a maximum speedup

```
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;;
```

Fig. 6. Recursive *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;;
```

Fig. 7. Direct *euler* in GAP

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

Table 3. Sequential

PE	GCA-GAP				GUM	
	Dir rt	Spd	Rcr rt	Spd	rt	Spd
1	3,006s	1	500s	1	164s	1.0
2	1,139s	2.6	320s	1.5	121s	1.3
4	542s	5.5	154s	3.2	69s	2.3
6	365s	8.2	107s	4.6	45s	3.6
8	267s	11.2	84s	5.9	38s	4.3
12	174s	17.2	59s	8.4	39s	4.2
16	141s	21.3	51s	9.8	35s	4.6
20	115s	26.1	45s	11.1	30s	5.4
28	95s	31.6	40s	12.5	23s	7.1

Table 4. 32,000

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

Table 5. 80,000

Table 6. *sum-Euler* Runtime and Speedup

of 31.6 on 28 PEs. Despite this, it is still slower than GUM by a factor of 18.3 on 1 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 [20, 11]. 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 [21]. The modest super-linearity in the GCA-GAP program can again be attributed to reduced memory management costs.

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 6 on 2 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 3 on 1 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 GCA-GAP 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

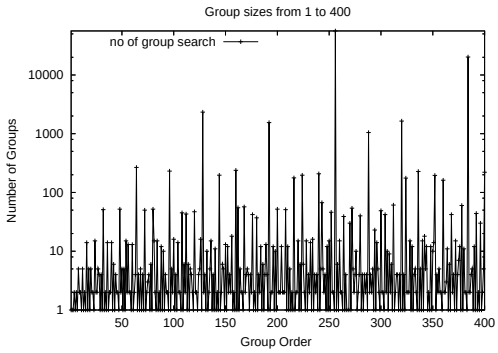

Fig. 8. Number of groups of a given order from 1 to 400

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 Figures 8, the number of groups of a given order varies enormously, i.e. by *5 orders of magnitude*; and
- secondly, there are variations in the cost of computing the prime power of each group.

The kernel of the *smallGroup* program is shown in Figures 9 and 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 [8] to generate a set of hierarchical master worker tasks to calculate `IntAvgOrder` in GAP.

```
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])
```

Fig. 9. GCA: *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;
```

Fig. 10. GAP: *smallGroup* search code

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 7 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 913s for the sequential GAP execution.

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

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

Table 7. [256..256]

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

Table 8. [1 ... 400]

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

Table 9. [600 ... 1000]

Table 10. *smallGroup*

then generates worker tasks for each element of the sequence. To demonstrate repeatability, we consider two sets of inputs, for orders in the ranges 1 to 400 and 600 to 1000, respectively.

Results for *smallGroup* [1 ... 400] Table 8 shows the results of computing *smallGroup* for orders between 1 and 400, where a total of 87,927 candidate small groups are considered. For these inputs, the sequential GAP program takes $1,658s$. 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.

Results for *smallGroup* [600 ... 1000] Table 9 similarly shows the results of computing *smallGroup* for orders between 600 and 1000, where a total of 1,163,006 candidate small groups are considered. For these inputs, the sequential GAP program takes $273,874s$. 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.

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 sixteen-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 5). We consider only the performance of the *smallGroup* program, since this represents the most serious of the applications we have previous 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 1000 on an eight-core machine. A total of 42,473 candidate

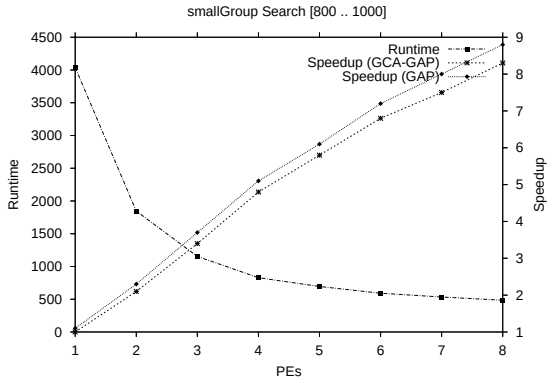

Fig. 11. *smallGroup* [800 ... 1000], on an **eight-core** Machine

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 ultra-light-weight threads that are available in GpH make it more suitable for use in multicore architecture [29, 18]. 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.

9 Related Work

Parallel Symbolic Computation. Work on parallel symbolic computation dates back to at least the early 1990s – Roch and Villard [26] 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. [6]. A number of one-off parallel programs have also been developed for specific algebraic computations, mainly in representation theory [24]. However, while several symbolic computation systems include some form of operator to introduce parallelism parallel Maple [7], or parallel GAP [10]), 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.

Parallel Functional Languages and Computer Algebra Systems include, for example, the GHC-Maple interface and the Eden-Maple system [23]. 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.

Orchestrating Services Over the last 20 years there has been a great interest in orchestrating services, for example *Grid Services*, such as job submission, data transfer and data portal services. A number of environments have been developed, both commercailly e.g. FlowMark [19] BPEL [13], and for research, e.g. DAGMan [14], and GridAnt [5]. **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 computational algebra systems 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 computational algebra system, 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 5 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 [27]. 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 computational algebra systems. 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 computational algebra systems, 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 Konovalov 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

1. *The Mathematica*. Wolfram Media, Inc., Champaign, IL, 1999.
2. The OpenMath Format, `http://www.openmath.org/`, 2007.
3. A. Al Zain, P. Trinder, H.-W. Loidl, and G. Michaelson. Managing Heterogeneity in a Grid Parallel Haskell. *J. Scalable Comp.: Practice and Experience*, 7(3):9–25, 2006.
4. A. Al Zain, P. Trinder, H-W. Loild, and G. Michaelson. Evaluating a High-Level Parallel Language (GpH) for Computational Grids. *IEEE TPDS*, 2007.
5. K. Amin, G. v. Laszewski, M. Hategan, N. J. Zaluzec, S. Hampton, and A. Rossi. GridAnt: A Client-Controllable Grid Workflow System. In *HICSS '04*, Washington, DC, USA, 2004. IEEE Computer Society.
6. 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.
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 *TFP'07*, Draft Proceedings, New York, USA, 2007.
9. B. W. Char et al. *Maple V Language Reference Manual*. Maple Publishing, Waterloo Canada, 1991.
10. G. Cooperman. GAP/MPI: Facilitating parallelism. In *Proc. DIMACS Workshop on Groups and Computation II*, volume 28, pages 69–84. AMS, 1997.
11. G. Cooperman. Parallel GAP: Mature interactive parallel. *Groups and computation, III (Columbus, OH,1999)*, 2001. de Gruyter, Berlin.
12. M. Daberkow, C. Fieker, J. Klüners, M. Pohst, K. Roegner, M. Schörnig, and K. Wildanger. Kant v4. *J. Symb. Comput.*, 24(3/4):267–283, 1997.
13. W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3(3-4):283–304, 2005.
14. J. Frey. Condor dagman: Handling inter-job dependencies. Technical report, University of Wisconsin, Department of Computer Science, 2002.
15. The GAP Group. GAP – Groups, Algorithms, and Programming, 2007. `http://www.gap-system.org/gap`.
16. K. Hammond, A. Al Zain, G. Cooperman, D. Petcu, and P. Trinder. SymGrid: a Framework for Symbolic Computation on the Grid. In *Proc. EuroPar'07 — European Conference on Parallel Processing*, LNCS, Rennes, France, 2007. Springer, to appear.

17. K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*, chapter Introduction. Springer-Verlag, 1999.
18. 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.
19. F. Leymann and D. Roller. Business process management with flowmark. In *Compcon Spring'94*, pages 230–234. IEEE Computer Society, 1994.
20. S.A. Linton and A. Konovalov. Gap memory management and recursive implementation. Technical Discussion, St Andrews, UK. http://www.gap-system.org/gap, February 2007. Attendee: Trinder, P.W. and Hammond, K.
21. H-W. Loidl and P. W. Trinder. Engineering Large Parallel Functional Programs. In *Implementation of Functional Languages, 1997*, LNCS, St. Andrews, Scotland, September 1997. Springer.
22. H-W. Loidl, P. W. Trinder, K. Hammond, S. B. Junaidu, R. G. Morgan, and S. L. Peyton Jones. Engineering Parallel Symbolic Programs in GPH. *Concurrency — Practice and Experience*, 11:701–752, 1999.
23. R. Martínez and R. Pena. Building an Interface Between Eden and Maple. In *Proc. IFL 2003, Springer-Verlag LNCS 3145*, pages 135–151, 2004.
24. G. O. Michler. *High performance computations in group representation theory.* Preprint, Institut fur Experimentelle Mathematik, Univerisitat GH Essen,, 1998.
25. K. Morisse and A. Kemper. The Computer Algebra System MuPAD. *Euromath Bulletin*, 1(2):95–102, 1994.
26. L. Roch and G. Villard. Parallel computer algebra. In *Proc. ISSAC '97: International Symposium on Symbolic and Algebraic Computation.* Preprint IMAG Grenoble France, 1997.
27. 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.
28. P.W. Trinder, K. Hammond, J.S. Mattson Jr., A.S Partridge, and S.L. Peyton Jones. GUM: a Portable Parallel Implementation of Haskell. In *Proc. PLDI'96*, pages 79–88, Philadelphia, PA, USA, May 1996.
29. P.W. Trinder, H-W. Loidl, E. Barry Jr., K. Hammond, U. Klusik, S.L. Peyton Jones, and Á.J. Rebón Portillo. The Multi-Architecture Performance of the Parallel Functional Language GPH. In *Euro-Par 2000 — Parallel Processing*, LNCS, pages 739–743, Munich, Germany, 2000. Springer-Verlag.