CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **28**:3606–3636
Published online 15 January 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.3746

HPC-GAP: engineering a 21st-century high-performance computer algebra system‡

Reimer Behrends[1], Kevin Hammond[2], Vladimir Janjic[2], Alexander Konovalov[2], Steve Linton[2], Hans-Wolfgang Loidl[3], Patrick Maier[4] and Phil Trinder[4,*,†]

[1]*Department of Mathematics, University of Kaiserslautern, Kaiserslautern, Germany*
[2]*School of Computer Science, University of St Andrews, Fife, UK*
[3]*School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, UK*
[4]*School of Computing Science, University of Glasgow, Glasgow, UK*

SUMMARY

Symbolic computation has underpinned a number of key advances in Mathematics and Computer Science. Applications are typically large and potentially highly parallel, making them good candidates for parallel execution at a variety of scales from multi-core to high-performance computing systems. However, much existing work on parallel computing is based around numeric rather than symbolic computations. In particular, symbolic computing presents particular problems in terms of varying granularity and irregular task sizes that do not match conventional approaches to parallelisation. It also presents problems in terms of the structure of the algorithms and data. This paper describes a new implementation of the free open-source GAP computational algebra system that places parallelism at the heart of the design, dealing with the key scalability and cross-platform portability problems. We provide three system layers that deal with the three most important classes of hardware: individual shared memory multi-core nodes, mid-scale distributed clusters of (multi-core) nodes and full-blown high-performance computing systems, comprising large-scale tightly connected networks of multi-core nodes. This requires us to develop new cross-layer programming abstractions in the form of new domain-specific skeletons that allow us to seamlessly target different hardware levels. Our results show that, using our approach, we can achieve good scalability and speedups for two realistic exemplars, on high-performance systems comprising up to 32 000 cores, as well as on ubiquitous multi-core systems and distributed clusters. The work reported here paves the way towards full-scale exploitation of symbolic computation by high-performance computing systems, and we demonstrate the potential with two major case studies. © 2016 The Authors. *Concurrency and Computation: Practice and Experience* Published by John Wiley & Sons Ltd.

Received 5 May 2015; Revised 27 October 2015; Accepted 2 November 2015

KEY WORDS: parallelism; multicore; high-performance computing; computational algebra

1. INTRODUCTION

This paper considers how parallelism can be provided in a production symbolic computation system, GAP (Groups, Algorithms, Programming [1]), to meet the demands of a variety of users. Symbolic computation has underpinned several key advances in Mathematics and Computer Science, for example, in number theory, cryptography and coding theory. Computational algebra is an important class of symbolic computation, where applications are typically characterised by complex and expensive computations.

*Correspondence to: Phil Trinder, School of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK.
†E-mail: Phil.Trinder@glasgow.ac.uk

‡The copyright line for this article was changed on 28 June 2016, after original online publication.

Many symbolic problems are extremely large, and the algorithms often exhibit a high degree of potential parallelism. This makes them good candidates for execution on both large and small-scale parallel systems, including high-performance computing (HPC) systems. With the widespread availability of multi-core systems, there is an increasing need to provide effective support for symbolic computation on small shared-memory computers. The size and complexity of the problems that are being tackled means, however, that there is also a significant demand for symbolic computation on clusters of distributed-memory systems and potentially for full-blown HPC systems. The HPC–GAP systems described in this paper aim to meet those diverse needs in a way that is idiomatic for the GAP programmer and easy for the domain specialist to exploit.

There are, however, numerous practical challenges that must be overcome if the parallelism potential is to be exploited. Firstly, symbolic algorithms tend to employ complex data and control structures that are not generally studied by the parallel programming community. Secondly, the parallelism that is present is often both dynamically generated and highly irregular, for example, the number and sizes of subtasks may vary by several orders of magnitude. These properties present significant challenges to conventional parallelisation techniques. Finally, the developers of symbolic computing applications are not parallelism experts: they are typically mathematicians or other domain experts, who usually do not possess a background in computer science and who have limited time and inclination to learn complex system interfaces. What is needed is a simple, but highly scalable, way to deal with massive parallelism in symbolic computing systems, which is consistent with existing symbolic programming approaches.

This paper studies how these challenges may be overcome in the context of the widely used GAP computational algebra system. Our work establishes symbolic computation as a new and exciting application domain for HPC. It also provides a *vade mecum* for the process of producing effective high-performance versions of large legacy systems. The primary research contributions of this paper are as follows:

1. The systematic description of HPC-GAP as an integrated suite of new language extensions and libraries for parallel symbolic computation. These are a thread-safe multi-core implementation, GAP5 (Section 2); an MPI binding to exploit clusters (Section 3) and the SymGridPar2 framework that provides symbolic computation at HPC scale (Section 4). Collectively, these allow us to address scalability at multiple levels of abstraction up to large-scale HPC systems (Section 5).
2. We provide two substantial case studies to demonstrate the effectiveness of the language mechanisms for irregular symbolic computations. Specifically, we consider Orbit calculations and a Hecke Algebra representation theory calculation on the HECToR HPC system. The datasets for these, and other experiments in the paper, are available at [2] (Section 6).
3. SymGridPar2 re-establishes symbolic computation as an application domain for HPC for the first time in 20 years (Sections 4, 5.3 and 6).

Novelty: We provide the first complete and systematic description of HPC-GAP. Many of the individual components of HPC-GAP and associated case studies have been previously reported piecemeal: with descriptions of GAP5 [3], SymGridPar2 [4], the MPI-GAP case study [5] and the SymGridPar2 case study [6]. A key part of the systematic presentation of the HPC-GAP components is to provide a common SumEuler benchmark to facilitate comparison of the programming models and performance at different scales. Moreover, some key aspects of the components are presented for the first time, for example, Section 4.2, describing the interface between GAP and SymGridPar2. Section 5 that uses SumEuler as a basis for a performance evaluation of the HPC-GAP components, and discusses their interworking, is almost entirely new.

1.1. Computational algebra and the GAP system

GAP ('Groups, Algorithms and Programming' [1]) is the leading open source system for computational discrete algebra. GAP supports very efficient linear algebra over small finite fields, multiple representations of groups, subgroups, *cosets* and different types of group elements, and backtrack search algorithms for permutation groups. The GAP system and extension packages now comprise 360K lines of C and 900K lines of GAP code that is distributed under the GNU Public License. To

DOI: 10.1002/cpe

date, GAP has been installed at over 2000 sites and is cited in more than 1000 publications. GAP has been used in such landmark computations as the 'Millennium Project' to classify all finite groups of order up to 2000 [7]. From its inception, GAP was designed to be natural to use for mathematicians, powerful and flexible for expert users and freely extensible so that it can encompass new areas of mathematics as needed. All three objectives have been met comprehensively. Our achievements were recently recognised by the award of the ACM/SIGSAM Richard D. Jenks Memorial Prize for Excellence in Software Engineering applied to Computer Algebra.

GAP fills an important need in the computer algebra community. Unlike GAP, general 'computer algebra systems' such as Maple [8], Mathematica [9] or MATLAB [10] simply do not address computation in abstract algebra in any serious way. Maple, for example, can compute with some finite permutation groups, but is many orders of magnitude slower than GAP and can handle only 'toy' examples. Only MAGMA [11] and Sage [12] have any significant overlap with GAP in its core areas of computational algebra and discrete mathematics. MAGMA has a strong focus on number theory and cryptography, and unlike GAP, is closed source, monolithic and difficult to extend. Sage [12] incorporates and derives much of its algebraic functionality from GAP, as permitted by the GNU Public License. Neither of these systems supports parallelism well: Sage has limited support for client-server style distributed computing, but does not support large-scale parallel computing; and MAGMA currently has no support for parallelism.

1.2. Parallelism and high-performance computing

With the widespread adoption of multi-core processors, parallelism has become mainstream. While most laptops, desktops and even mobile platforms comprise single multi-core systems, higher-performance systems will *cluster* several multi-cores into a single coherent system, connected either by a high-speed network or by a shared memory bus. HPC systems provide a further level of hierarchy, connecting multiple clusters into a coherent, highly parallel supercomputing platform. While the general problems of identifying and controlling parallelism are similar in each level, the increasing scale and complexity of larger systems means that more powerful programming models are needed to deal with them effectively. These models should abstract over low-level architectures, in order to provide scalable parallelism up to HPC-size architectures.

While low-level programming models are still dominant for HPC applications, this low level of abstraction is in stark contrast to the high level of abstraction sought for in the domain of symbolic computation. A typical user of a computational algebra system will naturally aim to develop theories and programs in a style close to mathematical notations and therefore refrain from (prematurely) constraining how the theories are executed. In fact, the bigger gap between model and implementation found in symbolic computation makes parallel realisation of these problems relatively rare, and some exceptions are discussed in [13].

While there are instances of explicit coordination infrastructures in this domain, and we discuss them in the following sections, the most natural combination of technology and domain is a mostly declarative programming style, and the main part of our survey here focuses on such systems.

1.2.1. Threading and message-passing approaches. The most commonly used technologies on today's HPC systems start from established, if not dated, sequential host languages, such as Fortran or C, and provide added functionality for parallelism in the form of libraries or compiler annotations. Different technologies are often combined, reflecting the underlying architecture, for example, using an explicit message passing library (e.g. MPI) between nodes combined with a dedicated shared memory model (e.g. OpenMP in the form of compiler annotations) within a multi-core node. This combination aims to use the most efficient technology on each level, but necessarily complicates software design. This development is aggravated by the heavy use of Graphic Processing Units (GPUs) and many-core co-processors, such as Xeon Phis, on the latest generation of supercomputers.

In contrast to the mixed paradigm approach common in HPC codes, we advocate a unified, high-level approach. Such an approach abstracts over low-level machine details, reducing the complexity of the parallel programming task, while still enabling the tuning of key parallel execution characteristics, such as the size of the parallel tasks and the distribution of the application data. The latter

DOI: 10.1002/cpe

is becoming increasingly important with modern applications often being extremely data intensive and the memory-per-core ratio falling. In this aspect, symbolic computation software technology is ahead of mainstream HPC, because symbolic applications often require enormous and complex intermediate data structures. Hence, bespoke libraries have been developed to deal with huge data structures efficiently have been developed, for example, the Roomy library discussed in Section 1.3. Hence, control of data, in addition to control of tasks, features prominently in the design of HPC-GAP.

1.2.2. Parallel patterns and skeletons. Parallel pattern that abstracts and re-uses common parallelism idioms is an increasingly important technology [14]. *Algorithmic skeletons* [15] provide specific implementations of patterns in the form of functions that have effective parallel implementations and that are instantiated with concrete parameters to specify functionality and behaviour. The practical relevance of such approaches is underlined by recent practitioner oriented textbooks on parallel patterns [16, 17] and by high levels of activity in projects such as Google's MapReduce [18] and Apache's Hadoop [19].

Well-known skeleton systems include P^3L [20], Skil [21] and the SCL coordination language [22], which are all based on a functional coordination language. Moreover, numerous skeleton libraries provide skeletons that can be used from a mainstream sequential host language, for example, SkePU [23], Müsli [24] and eSkel [25], which are all embedded in C/C++.

While general purpose skeleton libraries like those mentioned previously can be used across domains, some packages encode domain-specific patterns. One such example is the LinBox [26] exact linear algebra library, which provides a wide selection of linear algebra operations operating over arbitrary precision numbers. Another example is our work on the Orbit pattern [5], a commonly occurring pattern in symbolic computation applications used as a case study in Section 6.1.

An important development in the area of programming languages for HPC systems is the emergence of partitioned global address space (PGAS) languages as a new model of high-level parallel programming. The basic idea in these languages is to provide language support for distributing large data structures, typically flat arrays, over a collection of distributed memory nodes. Parallelism is then expressed in the form of data parallelism, with constructs to place a computation on the machine that holds the relevant segment of the data structure, and synchronisation and communication become implicit. While the focus is on data parallel constructs, general fork-and-join parallelism is also available, although less frequently used. First generation PGAS languages that defined these concepts have been developed by major HPC vendors: Chapel [27] by Cray, X10 [28] by IBM and Fortress [29] by Sun/Oracle. More recently, PGAS constructs have been integrated into mainstream languages: Unified Parallel C [30] and Coarray Fortran [31].

1.3. Parallel computational algebra

Several computer algebra systems offer dedicated support for parallelism and/or distribution ([32, Sec 2.18] and [33]). Early systems focused on shared-memory architectures, and some prominent examples are Aldor [34], PARSAC2 [35] and PACLIB [36]. Because our focus is on distributed memory systems, we do not discuss these further.

Distributed Maple [37] provides a portable Java-based communication layer to permit the interaction of Maple instances over a network. It uses future-based language constructs for synchronisation and communication and has been used to parallelise several computational geometry algorithms. Unlike our system, it does not provide specific multi-core or HPC support, however. The Sugarbush [38] system is another distributed memory extension of Maple, which uses Linda as a coordination language. The GAPMPI [39] package is a distributed-memory parallel extension to GAP, which provides access to the MPI interface from within GAP. Unlike our work, no higher-level abstractions are provided to the programmer.

The TOP-C system provides task-oriented parallelism on top of a distributed shared memory system [40], implementing several symbolic applications, including parallel computations over Hecke algebras [41] on networks of workstations.

The Roomy system [42] takes a data-centric view of parallel symbolic computation, recognising that many applications in the domain manipulate enormous data structures. Roomy provides a library for the distribution of data structures and transparent access to its components, implemented

DOI: 10.1002/cpe

on top of the MPI message passing library. Example applications implemented on top of Roomy include a determinisation algorithm for non-deterministic finite state automata.

Two decades ago, there were several projects that parallelised algebraic computations, but there has been little work in this area since. Sibert *et al.* [43] describe the implementation of basic arithmetic over finite fields on a Connection Machine. Roch *et al.* [44] discuss the implementation of the parallel computer algebra system PAC on the Floating Point System hypercube Tesseract 20 and study the performance of a parallel Gröbner Basis algorithm on up to 16 nodes. Another parallel Gröbner Basis algorithm is implemented on a Cray Y-MP by Neun and Melenek [45] and later on a Connection Machine by Loustaunau and Wang [46]. We are not aware of any other work within the last 20 years that targets HPC for computational algebra.

More recently mainstream computer algebra systems have developed interfaces for large-scale distribution, aiming to exploit Grid infrastructures [47]. The community effort of defining the Symbolic Computation Software Composability Protocol (SCSCP) for symbolic data exchange on such infrastructures supports distributed execution and interchange between different computer algebra systems [48]. In contrast to these Grid-based infrastructures, our SymGridPar2 framework targets massively parallel supercomputers.

2. PARALLELISM SUPPORT IN GAP5

The shared-memory component of HPC-GAP, which we refer to for this paper as GAP5 reimplements the latest version of GAP (GAP4) to include support for parallelism at a number of levels [3]. This has a number of implications for the language design and implementation. Distributed memory implementations of memory hungry symbolic computations can duplicate large data structures and may even lead to memory exhaustion. For example, the standard GAP library already requires several 100 MB of memory per process, much of which is data stored in read-only tables; a multi-threaded implementation can provide access to that data to multiple threads concurrently without replicating it for each thread. This is why it is crucial to use shared memory to achieve efficient fine-grained parallelisation of GAP applications. Secondly, we need to maintain consistency with the previous version of the GAP system, refactoring it to make it *thread-safe*. Achieving this has involved rewriting significant amounts of legacy code to eliminate common, but unsafe, practices such as using global variables to store parameters, having mutable data structures that appear to be immutable to the programmer, and so on.

The overall GAP5 system architecture aims to expose an interface to the programmer that is as close to the existing GAP4 implementation as possible in order to facilitate the porting of algorithms and libraries. The two primary new concepts that a GAP5 programmer may need to understand is that of *tasks* (an implementation of futures [49]) and *regions*, a GAP-level abstraction for areas of shared memory. Existing sequential code will run largely unchanged, and sequential code can also use parallelised libraries transparently. Only code that exploits GAP5 concurrency or kernel functionality needs to be adapted to the new programming model.

2.1. Task introduction and management

GAP5 adopts a future-based task model [49], identifying tasks using the `RunTask` primitive: `td := RunTask(f, x1, ... xn);` Here, f is a GAP function, and the x_i are the arguments to be passed to that function. A call to `RunTask` introduces a *future* into the runtime system, which will eventually yield a parallel task executing $f(x_1, \ldots, x_n)$. `RunTask` returns a task descriptor, `td`, which can be used to further control or observe the task behaviour. In particular, the result of a task can be obtained using a blocking call to the `TaskResult` primitive; `WaitAnyTask` can be used to block until one of a number of tasks has completed; and `ScheduleTask` is a variant of `RunTask` that allows the creation of a task at some future point when the specified condition becomes true.

```
result := TaskResult(td);
...
td1 := ScheduleTask(cond, f, x1, ..., xn);
...
tdx := WaitAnyTask(td1, td2, td3);
```


DOI: 10.1002/cpe

The return value of `TaskResult` is the value returned by the task. The return value of `ScheduleTask` is a task descriptor. `WaitAnyTask` returns a positive integer that denotes which of the tasks passed as arguments (via their descriptors) completed first, starting with 1 for the first argument. In the aforementioned example, `tdx` would be 1 if `td1` completes first, 2 if `td2` completes first or 3 if `td3` completes first.

2.2. SumEuler in GAP5

We use the SumEuler computation as a common example to demonstrate the GAP5, MPI-GAP and SymGridPar2 components of HPC-GAP. SumEuler is a symbolic computation that computes the sum of Euler's totient function over an integer interval. The computation is a fold (the sum) of a map (of the totient function). The computation is irregular as the time required to compute the totient of a large integer is greater than for a small integer. We use chunked data parallel versions here. That is, the time to compute the totient of a single integer is small, so tasks typically compute the totients for a 'chunk' or interval of integers.

Figure 1 illustrates how to express SumEuler in GAP5, the source cade for this and all other examples is available at [2]. `SumEulerSeq` is a simple sequential implementation, using the standard GAP implementation of the totient function, `Phi`. The GAP `List` function implements a functional *map* operation: taking a list as its first argument and a function or closure as its second, applying that function to each element in the list and returning the resulting list. `SumEulerPar` is a very simple parallelisation of the same code. It starts a task for each number in the range `n1` through `n2`, then collects the results of those tasks. While simple, this is also inefficient because most computations of `Phi(x)` will be very short and the overhead will therefore be high. `SumEulerPar2` is a more efficient implementation that reduces the overhead by aggregating the data into intervals of size 100.

2.3. Shared regions in GAP5

Safe mutable state in GAP5 is handled through the concept of *shared regions*. A region is a set of GAP objects; each object in GAP5 belongs to exactly one region, and each region has exactly one associated read-write lock. The GAP5 runtime ensures that an object is only accessed by a thread that has either a read or write lock on the region containing the object. Each thread has an

```
1  SumEulerSeq := function(n1, n2)
2    return Sum(List([n1..n2], Phi));
3  end;
4
5  SumEulerPar := function(n1, n2)
6    local tasks;
7    tasks := List([n1..n2], x → RunTask(Phi, x));
8    return Sum(List(tasks, TaskResult));
9  end;
10
11 SumEulerParAux := function(n1, n2, step)
12   local i, tasks;
13   tasks := [];
14   i := n1;
15   while i ⩽ n2 do
16     Add(tasks, RunTask(SumEulerSeq, i, Minimum(i + step - 1, n2)));
17     i := i + step;
18   od;
19   return Sum(List(tasks, TaskResult));
20 end;
21
22 SumEulerPar2 := function(n1, n2)
23   return SumEulerParAux(n1, n2, 100);
24 end;
```

Figure 1. *SumEuler* in GAP5.

DOI: 10.1002/cpe

associated *thread-local region*, on which it has a permanent write lock. By default, newly created GAP objects are created in the thread-local region of the current thread. The goal of thread-local regions (in particular, the thread-local region of the main thread) is to present programmers with a quasi-sequential environment. This environment is protected against unexpected interference from other threads, because only the thread associated with the thread-local region can ever access it, thus allowing programmers controlled use of concurrency features.

A programmer can create any number of *shared regions*. The `ShareObj(ob)` primitive takes a GAP object as an argument, creates a new shared region and *migrates* the object and all its sub-objects to the shared region. The `ShareSingleObj(ob)` variant migrates only `ob` and not any sub-objects. Objects in a shared region must be accessed using the GAP5 `atomic` statement, as shown as follows.

```
SharedList := ShareObj([]);

AddShared := function(x)
  atomic readwrite SharedList do
    Add(SharedList, x);
  od;
end;

LastShared := function()
  atomic readonly SharedList do
    return SharedList[Length(SharedList)];
  od;
end;
```

Note that we never operate on a region directly. The `atomic` statement operates on GAP objects, and the corresponding region is inferred by the runtime system. The `atomic` statement also allows the simultaneous locking of multiple regions by providing multiple arguments. These regions are then guaranteed to be acquired atomically and without causing deadlock. The following code fragment illustrates this usage of `atomic`.

```
atomic ob1, ob2 do
  DoSomethingWith(ob1, ob2);
od;
```

A special case of a shared region is the *read-only region*. Every thread holds a permanent read-lock for that region, and objects can be migrated to it using the `MakeReadOnly` primitive. This is primarily intended for the convenient storage of large, constant tables in order to access them without the overhead and inconvenience of an `atomic` statement.

Objects can be *migrated* between shared regions or between the current thread-local region and a shared region. Unlike copying, migration is generally a cheap operation, which only changes the region descriptor of the underlying GAP object (a single word). As already mentioned, `ShareObj(ob)` implicitly migrates its argument to the new region. More generally, `MigrateObj(ob1, ob2)` migrates `ob1` and any sub-objects to the region containing `ob2`. For this to succeed, the current thread must have a write lock on both regions. The `AdoptObj(ob)` primitive is a special case of `MigrateObj` that migrates `ob` and any sub-objects to the current thread-local region.

Figure 2 shows how this works in practice for the parallel multiplication of two square thread-local matrices. `ParMatrixMultiplyRow` is the usual GAP code to multiply the *i-th* row vector of `m1` with the matrix `m2`. It is identical to the sequential version except that it is declared to be *atomic*, and `m1` and `m2` are declared to be `readonly`. `ParMatrixMultiply` contains the actual parallel matrix multiplication code. This extends the sequential version slightly.

DOI: 10.1002/cpe

```
1  ParMatrixMultiplyRow :=
2    atomic function(readonly m1, readonly m2, i)
3      local result, j, k, n, s;
4      result := [];
5      n := Length(m1);
6      for j in [1..n] do
7        s := 0;
8        for k in [1..n] do
9        s := s + m1[i][k] * m2[k][j];
10       od;
11       result[j] := s;
12     od;
13     return result;
14   end;
15
16 ParMatrixMultiply := function(m1, m2)
17   local tasks, result;
18   ShareObj(m1);
19   ShareObj(m2);
20   atomic readonly m1, readonly m2 do
21     tasks :=
22       List([1..Length(m1)],
23         i → RunTask(ParMatrixMultiplyRow, m1, m2, i));
24     result := List(tasks, TaskResult);
25   od;
26   atomic m1, m2 do
27     AdoptObj(m1);
28     AdoptObj(m2);
29   od;
30   return result;
31 end;
```

Figure 2. Parallel matrix multiplication in GAP5.

Firstly, in lines 18–19, `m1` and `m2` are placed in shared regions. Then, in lines 21–23, each row vector of `m1` is multiplied with `m2` using its own GAP task. Line 24 collects the results. Finally, in lines 26–30, `m1` and `m2` are migrated to the current thread-local region using atomic calls to `AdoptObj`.

2.4. Comparison with other parallel computational algebra systems

Symbolic Computation Software Composability Protocol. The computational algebra community has defined a common protocol for combining computer algebra systems, the SCSCP [48]. Essentially, it is possible to make a remote procedure call from one system to another, and both protocol messages and data are encoded in the OpenMath format. SCSCP compliant components may be combined to solve scientific problems that cannot be solved within a single CAS, or may be organised into a system for distributed parallel computations. The SCSCP has become a *de facto* standard for mathematical software with implementations available for seven CAS and libraries for C/C++, Java Haskell, and so on. On a single multicore, GAP5 provides higher performance by allowing direct functions manipulating GAP objects, rather than an SCSCP RPC where the function and arguments are encoded as OpenMath XML objects. Moreover, while SCSCP's explicit RPC mechanism is suitable for small-scale parallel execution, SymGridPar2 scales onto HPCs.

GAP4. The current stable release of GAP4 provides two ways to run distributed memory parallel computations: (i) the ParGAP package that uses MPI and (ii) an SCSCP package. Both provide basic implementations of the `ParList` skeleton plus low-level tools to develop new parallel GAP code. Both approaches are prone to limitations of the distributed memory approach concerning serialisation, and so on. In particular, it may not be straightforward or may even be impossible to transmit complex GAP objects from one node to another, limiting performance [50].

Sage has some rudimentary facilities for distributed memory parallel calculations, which are still under development. The current design focuses on task parallelism and uses Python's

DOI: 10.1002/cpe

decorator concept to add parallel execution functionality to existing methods. While this adds concurrency to the programming model, it does not provide means of explicit synchronisation or communication. Another parallelism primitive is a parallel iterator. The implementation of parallelism is also seriously hampered by the central scheduler lock in the Python runtime-system.

Maple supports multithreading on shared memory architectures through a task programming model. While its kernel is thread-safe, work on a thread-safe library is still in progress. On distributed memory architectures parallelism is supported by the Maple Grid Toolbox [47]. It is implemented on top of MPI and can be configured in either MPI (cluster) or HPC mode. The latter allows for integration with batch job scheduling middleware, whereas the former is targeted at dynamic, self-assembling Grid infrastructures. In terms of language support for parallelism, the Maple Grid Toolbox mainly provides MPI-level primitives for synchronisation and communication. Support for higher-level primitives is rudimentary and at the moment restricted to a parallel map operation. The older Distributed Maple [37] system, with its higher-level coordination primitives, has been discussed in Section 1.3.

MAGMA is a single-threaded application that does not support parallelism [11]. Distributed memory computations could be emulated by, for example, interfacing multiple copies to Sage, or by using an SCSCP wrapper for MAGMA [48].

TRIP is dedicated to celestial mechanics, specialising in operations with polynomials [51]. Its single-core implementation is multithreaded, reporting good parallel performance on sparse polynomial operations [52], and SCSCP can be used for small-scale distributed memory parallelism.

MATLAB. The MATLAB parallel computing toolbox supports multicores, GPUs and clusters using a number of simple primitives to expose data parallelism and loop-parallelism, based on parallel for-loops [10]. Support for heterogeneous devices is provided through a Cuda interface.

Using the MATLAB Distributed Computing Server, parallelism can be orchestrated on large-scale, distributed memory configurations. The preferred programming mode is data parallelism over the pre-defined data structure of distributed arrays. Additionally, a general map-reduce structure is available. On a lower level, primitives for single-program multiple-data parallelism and explicit message passing are also available.

The current version of MATLAB's symbolic computation toolbox, MuPAD, does not support parallel computing. However, earlier versions of MuPAD provided master-work parallelism on distributed systems in the form of 'macro parallelism' [53].

Mathematica. Starting with Mathematica 7, the internal Wolfram Language also supports parallel computation. The focus of the language support is on data-parallelism, with pre-defined parallel map and combine operations. Parallel iterators are provided for common data structures. Speculative parallelism is also supported, enabling the programmer to try several approaches to solve a problem and pick up the first successful computation. More generally, fork-and-join parallelism can be defined, using lower-level thread creation and synchronisation primitives. However, the language design favours the usage of the higher-level primitives, in particular for defining map-reduce-style computations. Support for heterogeneous devices is provided through separate interfaces to Cuda and to OpenCL. In a distributed memory setting, the parallelism generated by the primitives is managed by the gridMathematica system, originally developed for large-scale Grid architectures [54]. This system automatically handles marshalling and communication of data structures, as well as retrieving the results from a remote execution. Through the usage of the higher-level parallelism primitives, the details of the architecture are mostly hidden, although aspects such as communication latency need to be taken into account for parallel performance tuning.

Reflection. Thus, we see that the majority of existing computational algebra system only provide support for (independent) distributed parallel calculations, or else have limited computational algebra functionality. The work described in this paper considers a wider class of parallel computation, including shared-memory multicores, clusters and HPC system, as well as distributed systems. Through the free GAP system, it makes the benefits of parallelism available to a much wider, open source community. Achieving this, while obtaining significant real speedups, represents

DOI: 10.1002/cpe

a major technical challenge: GAP is a large dynamically typed, interpreted system, with a complex implementation that manipulates large and complex data structures.

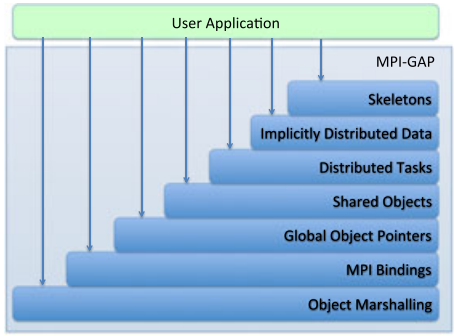
3. MPI-GAP DESIGN AND IMPLEMENTATION

MPI-GAP is an extension of GAP5, which targets distributed-memory systems by building on the *de facto* standard MPI message passing library. MPI-GAP provides both a high-level programming model that supports the same set of high-level parallel task constructs as the shared-memory version of GAP5, `CreateTask`, `RunTask`, `TaskResult`, and so on, and a low-level programming model that offers more control of distributed task and data placement when required. MPI-GAP is implemented in a layered manner (Figure 3), where each layer builds on the constructs provided by the layer below to offer a higher level of abstraction.

Object marshalling layer. This layer consists of operations that transform GAP objects into an equivalent binary representation that can be transmitted using MPI primitives. MPI-GAP currently supports two classes of marshalling operations: *native serialisation*, implemented as kernel operations, and *IO pickling*, implemented as GAP-level operations. Native serialisation is much faster, but it is architecture dependent and currently only allows certain GAP objects to be marshalled. IO pickling is significantly slower but is architecture independent and more customisable – more data structures are supported, and programmers can provide functions to marshal custom data structures. In general, if an application needs to transfer large amounts of 'simple' data (e.g. lists of simple objects and matrices) and the underlying data processing is sufficiently fine-grained, native serialisation is preferred. Conversely, when transferring smaller objects or when the underlying data processing is coarse-grained, IO pickling should be used.

MPI binding layer. This layer consists of GAP bindings to raw MPI functions to send and receive messages, based largely on ParGAP [55]. Currently, only a small subset of MPI functions are supported on the GAP side, including blocking and non-blocking MPI send and receive functions, probing, and so on. Object marshalling primitives must be used to marshal GAP objects so that they can be transmitted using these operations.

Global object pointer layer. This layer supports operations on GAP objects (handles) that represent pointers to other GAP objects that may live on remote computing nodes. Handles can be copied



Layer	Supported primitives/operations
Distributed Tasks	`CreateTask`, `RunTask`, `TaskResult`, `SendTask`...
Shared Objects	`RemoteCopyObj`, `RemoteCloneObj`, `RemotePushObj`, `RemotePullObj`
Global Object Pointers	`CreateHandleFromObj`, `SendHandle`, `SetHandleObj`, `GetHandleObj`
MPI Bindings	`MPI_Send`, `MPI_Isend`, `MPI_Recv`, `MPI_Probe`, `MPI_Iprobe`, ...
Object Marhsalling	`SerializeToNativeString`, `DeserializeNativeString`, `IO_Pickle`, `IO_Unpickle`

Figure 3. MPI-GAP layers and the associated primitives/operations.

DOI: 10.1002/cpe

between different GAP processes and can then be used to access the same physical GAP object from different nodes. Handles, and the underlying GAP objects, are managed in a semi-automatic way – handles are created, opened, closed and destroyed manually, but a reference-counting mechanism prevents unsafe operations, for example, destroying a handle that exists on multiple distributed nodes and, so, possibly garbage-collecting the underlying object.

Handles are created using the `CreateHandleFromObj` operation that takes a GAP object and returns a handle to it. Depending on the required level of automatic control, handles can be *read-only*, *read-write* or *volatile*. Objects referred to by *read-only* handles can be freely copied between nodes, but cannot be modified using `SendHandle`, `GetHandleObj` or `SetHandleObj`. Objects referred to by *read-write* handles can be freely modified, but the underlying objects cannot be copied between nodes. They can, however, be migrated. Finally, *volatile* handles are unrestricted, that is, a programmer has full control and is responsible for handling consistency of cached copies, and so on. Handles can be transferred between nodes using the `SendHandle` operation. They can be read and modified using the `GetHandleObj` and `SetHandleObj` operations.

Shared object layer. This layer supports operations on objects that are shared between multiple distributed nodes and accessed via handles from the global object pointer layer. Objects can be copied using the `RemoteCopyObj` or `RemoteCloneObj` operations and migrated using the `RemotePushObj` and `RemotePullObj` operations, where, in each case, the first version is used by the sender and the second by the recipient. Depending on the handle type, certain operations might be prohibited for safety reasons, for example, `RemoteCopyObj` is prohibited on read-write handles.

```
1  DeclareGlobalFunction(''Euler'');
2  InstallGlobalFunction(Euler,
3    function(handle)
4      Open(handle);
5      inputList := GetHandleObj(handle);
6      resList := List ( EulerPhi, inputList);
7      res := Sum(resList);
8      Close(handle);
9      return resList;
10   end
11 );
12
13 SumEuler := function (n)
14   if processId = 0 then
15     chunkSize := n / commSize;
16     ts := [];
17     # send work to other nodes
18     for nodeId in [1..commSize] do
19       h := CreateHandleFromObj ([n*chunkSize+1..(n+1)*chunkSize])
20       SendHandle (h, nodeId);
21       RemotePushObj(h);
22       ts[nodeId] := CreateTask (''Euler'', h);
23       SendTask (t, nodeId);
24     od;
25     # do local calculation
26     result = Sum(Euler ([1..chunkSize]));
27     # fetch the results
28     for nodeId in [1..commSize] do
29       remoteResult = TaskResult(ts[nodeId]);
30       result := result + remoteResult;
31     od;
32     return result;
33   else
34     return -1; # only master process should call this function
35   fi;
36 end;
```

Figure 4. SumEuler in MPI-GAP.

DOI: 10.1002/cpe

Distributed task layer. This layer supports the same task creation and management operations as shared-memory GAP5, for example, `CreateTask`, `RunTask` and `TaskResult`. In addition, it also supports explicit task placement using the `SendTask` operation. Task distribution between nodes in MPI-GAP uses a *random work stealing* work stealing approach, which has been shown to perform well in low-latency settings. Tasks that are not yet executed are kept in node-local *task pools*. Each node apart from the master starts in the *idle* mode and requests work from another randomly chosen node. If a node that has more than one task receives a message requesting work, it offloads one of its tasks. This distributes tasks among the nodes on an as-needed basis.

3.1. SumEuler in MPI-GAP

Figure 4 shows the implementation of SumEuler in MPI-GAP. Line 1 declares a global function to map the Euler totient function over a list of values and sum the results. In lines 2–11, we implement this function, passing it a handle to a list of positive integers. We first open the handle (line 4), in order to inform the reference-counting mechanism that the handle is in use by the current thread. We then read the list that the handle points to (line 5). After applying the EulerPhi function to all of the elements of the list, to produce a list of results resList (line 6), we close the handle (line 8) to inform the reference-counting mechanism that the current thread has finished using the handle.

The main function, SumEuler (lines 12–35), is only called by a master GAP process (i.e. the GAP instance that controls the execution). We first determine the sizes of the chunks of work that will be sent to each worker GAP process (line 15) and initialise a list of tasks (line 16). For each worker process, we create a handle to a list of positive integers (line 19), send the handle to the worker process (line 20), migrate the actual list (line 21), create a task that calls the Euler function on the handle (line 22) and finally send the task.

The program implements *eager* work distribution by explicitly assigning tasks to the processes. After sending work to each worker process, the master process does its own chunk of work (line 26), and then fetches and sums the results of the tasks (lines 28–31).

4. THE DESIGN AND IMPLEMENTATION OF SYMGRIDPAR2

SymGridPar2 (SGP2) is middleware for coordinating large-scale task-parallel computations distributed over many networked GAP instances. SymGridPar2 inherits its architecture, and specifically, its skeleton-based programming model, from SymGridPar [56]. The key new feature is the use of HdpH (Haskell distributed parallel Haskell), a novel *domain-specific language* (DSL) for task-parallel programming on large-scale networks, including HPC platforms. The SGP2 middleware comprises three layers:

1. the HdpH DSL for parallel coordination, embedded into Haskell (Section 4.1);
2. a Haskell binding for GAP (Section 4.2); and
3. a set of algorithmic skeletons, written in HdpH and calling into GAP (Section 4.3).

Figure 5 illustrates the SGP2 architecture and its interaction with GAP. In the figure, the outer dotted rectangles represent a host, possibly multi-core, and while two instances are shown, there may be an arbitrary number. The solid rectangles represent operating system processes, and the arrows represent communication.

The intention is that a GAP program invokes appropriate skeletons, essentially GAP functions with parallel semantics, for example, `parList([40..60], fibonacci)` may create 21 tasks to compute the Fibonacci function of the integers between 40 and 60. The GAP skeleton corresponds to an HdpH skeleton that distributes the subproblems to multiple hosts where they are passed to local GAP instances to be solved, and the solutions are returned to the caller.

As the GAP skeleton library has not yet been constructed, SGP2 applications are HdpH programs. While this section presents some key Haskell code, it is intended to be comprehensible without knowledge of Haskell. The skeletons generate tasks, which are distributed and executed by the HdpH runtime; HdpH provides sophisticated distributed work stealing built on top of an MPI

DOI: 10.1002/cpe

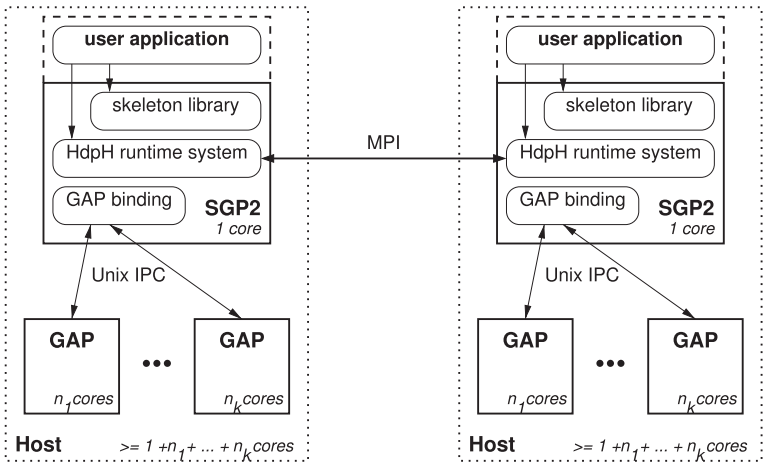


Figure 5. SymGridPar2 architecture.

```
-- Computations
data Par a                  -- data type of computations returning type 'a'
runParIO :: Par a → IO a    -- run parallel computation

-- Communication of results
data IVar a                 -- data type of futures of type 'a'
get :: IVar a → Par a       -- blocking read on a future

-- Locations
data Node                   -- data type representing hosts running SGP2
myNode   :: Par Node        -- current host
allNodes :: Par [Node]      -- list of all hosts
```

Figure 6. Type signatures of a minimal HdpH API.

communication layer. Tasks will typically call GAP functions through the GAP binding. Each SGP2 instance can transparently coordinate a pool of several GAP4 or GAP5 servers running on the same host. Communication between SGP2 and these GAP servers is via Unix IPC. SGP2 requires one dedicated core per host, the remaining cores can be used by GAP servers. On an n-core host, one SGP2 instance will typically coordinate $n - 1$ GAP4 instances or a single GAP5 instance running $n - 1$ threads. To reduce memory bus contention on large Non-Uniform Memory Access (NUMA) servers, it is also possible to coordinate several multi-threaded GAP5 instances, each bound to its own NUMA region. Section 5.4 contains some example deployments.

4.1. Coordination DSL

SGP2 exposes parallel patterns to the programmer as *algorithmic skeletons* [15], implemented in *HdpH*. HdpH is described in detail elsewhere [57, 58], and here, we discuss only the small part of the API which users of the SGP2 skeleton library are exposed to, as outlined in Figure 6. As is customary in functional programming languages, the HdpH DSL is embedded into the host language, Haskell, by wrapping DSL *computations* in a special computation type. In HdpH, this is performed by the `Par` type constructor, which forms a *monad*, the standard Haskell construct for effectful computations. Almost all HdpH and SGP2 primitives create or transform `Par` computations; ultimately, an SGP2 program is one large `Par` computation. The primitive `runParIO` takes such a computation and executes it on the HdpH runtime system.

In HdpH, parallel computations return results asynchronously through *futures*. These are represented by the type constructor `IVar`. Futures are created by a number of skeletons as well as by the GAP binding. A user may read the value stored in a future by calling the primitive `get`, which blocks the caller until the value is available.

DOI: 10.1002/cpe

As a distributed DSL, HdpH exposes *locations* (i.e. hosts) as values of type `Node`. SGP2 programs can query the current node and obtain a list of all nodes by calling the primitives `myNode` and `allNodes`, respectively. Crucially, the API in Figure 6 does not list any primitives for creating or distributing parallel *tasks*. This functionality is hidden in the skeletons discussed in Section 4.3.

4.2. GAP binding

HdpH is a DSL for coordinating task-parallel computations. Ordinarily, HdpH tasks evaluate function calls in Haskell. To coordinate computations in GAP, SGP2 has to provide a way to call from HdpH into GAP. This is performed by providing a *GAP binding*, that is, a library of Haskell functions to call GAP functions, including marshalling arguments from Haskell to GAP and results from GAP to Haskell.

Additionally, the GAP binding also provides functions for starting and stopping GAP servers. Each HdpH node can launch one or more GAP servers, running as independent operating system processes on the same host as the HdpH node and communicating via pipes. The API distinguishes between *stateful* GAP servers, each with a distinct identity, and *stateless* GAP servers, which are anonymous and interchangeable.

4.2.1. Controlling stateful GAP servers. Figure 7 shows the HdpH API for controlling GAP servers; primitives ending in `ST` are concerned with stateful servers. HdpH can start a GAP server by calling `startST`, which expects information on how to call GAP and a list of GAP function calls used to initialise the server. The primitive `startST` immediately returns a future that will eventually provide a handle to the GAP server, once the server is up and running. The primitive

```
-- Types
data GAP          -- Parameters to start GAP (path, flags, ..)
data GAPHandleST  -- Handle to stateful GAP server
data GAPObj       -- Encoded GAP object
data GAPError     -- Encoded GAP error
type GAPFunction  -- GAP function name
type GAPCall      -- GAP function call

-- Constructing GAP function calls
mkGAPCallFuncList :: GAPFunction → [GAPObj] → GAPCall

-- Calling individual stateful GAP server; blocking if not idle
callST :: GAPHandleST → GAPCall → Par (IVar (Either GAPError GAPObj))

-- Starting and stopping individual stateful GAP servers (on current node)
startST   :: GAP → [GAPCall] → Par (IVar GAPHandleST)  -- non-blocking
barrierST :: GAPHandleST → Par ()                      -- blocking
stopST    :: GAPHandleST → Par ()                      -- blocking

-- Querying the node of a stateful GAP server
atST :: GAPHandleST → Node

-- Calling any stateless GAP server; blocking if none idle
call :: GAPCall → Par (IVar (Either GAPError GAPObj))

-- Starting and stopping a pool of stateless GAP servers
start   :: GAP → [GAPCall] → Int → Par ()  -- non-blocking
barrier :: Par ()                          -- blocking
stop    :: Par ()                          -- blocking

-- Testers for GAP type of given GAP object
isGAPFail, isGAPBool, isGAPInt, isGAPRat,
isGAPList, isGAPNull, isGAPString, isGAPOpaque :: GAPObj → Bool

-- Class of Haskell types that can be marshalled to/from GAP.
class GAPEnc a where
  toGAP   :: a → GAPObj
  fromGAP :: GAPObj → a
```

Figure 7. HdpH Primitives for controlling GAP servers.

DOI: 10.1002/cpe

`barrierST` takes a server handle and blocks until the server is *idle*, that is, no longer busy executing a previous function call. Similarly, `stopST` blocks until the server is idle and then terminates the server (after which the handle is void). Finally, the primitive `callST` posts a function call to the given server. It blocks until the server is idle and then immediately returns a future that will eventually yield either an error or the result of the call. GAP function calls (type `GAPCall`) are simply strings. For convenience, they can be constructed by applying a GAP function name (type `GAPFunction`) to a list of encoded GAP objects using the function `mkGAPCallFuncList`; the encoding is discussed in Section 4.2.3.

Note that all primitives operate on stateful GAP servers running on the same host as the calling HdpH node; calls to distributed stateful GAP servers are provided by the higher-level skeletons discussed in Section 4.3. More precisely, any stateful GAP server can only be controlled by the HdpH node that started it, as identified by the primitive `atST`, and a GAP handle is only valid on the node indicated by `atST`.

4.2.2. Controlling stateless GAP servers. Many SGP2 applications interact with GAP in a stateless fashion, that is, GAP calls are proper function calls that do not manipulate hidden state on the server. For this case, HdpH offers primitives to coordinate a pool of stateless GAP servers.

The stateless primitives mirror the stateful ones (Figure 7). The `start` primitive takes the same information about calling and initialising GAP. Of course, the GAP calls used to initialise a stateless GAP server, like `startST`, change the system state, that is, are themselves stateful. The `start` function takes an additional pool size argument and returns nothing. The choice of pool size is critical for performance, for example, performance suffers if the pool size exceeds the number of physical cores. Because stateless servers are interchangeable, they can be anonymous, so there is no need to return a handle. The primitives `barrier` and `stop` behave like their stateful counterparts, except that they block until the whole pool of servers is idle, and terminate the whole pool.

The primitive `call` posts a function call to any stateless server. It blocks until a server is idle and then immediately returns an IVar that will eventually yield either an error or the result of the call. Note that the GAP function call is expected to be stateless in the sense that any side effects on the GAP server do not affect subsequent calls. Violating this restriction may result in unpredictable behaviour.

4.2.3. Marshalling data to and from GAP. Marshalling data between GAP and HdpH requires an encoding and a Haskell representation of encoded GAP objects; the latter is provided by the data type `GAPObj`. A `GAPObj` can represent the GAP constant `Fail`, GAP Booleans, GAP integers, GAP rationals, lists of GAP objects, the empty GAP list `[]`, GAP strings or *opaque* GAP objects; opaque objects are all those that are not explicitly listed in the aforementioned enumeration. The encoding assumes that GAP can print the object to be marshalled into a string that can be parsed back later; objects that do not support this need to be explicitly serialised to a GAP string prior to marshalling.

The bottom of Figure 7 displays the API for accessing `GAPObjs`. There are a number of functions testing which GAP type the encoded `GAPObj` represents. And there are two overloaded functions `toGAP` and `fromGAP`, defined on types instantiating class `GAPEnc`, for encoding to and decoding from `GAPObj`. The Haskell types, which are instances of `GAPEnc`, are the standard base types `()`, `Bool`, `Integer` and `Rational`, as well as `ByteString` (which encodes GAP strings). Standard list, pair and `Maybe` data types can also be encoded, provided their components are encodable.

Opaque GAP objects cannot be decoded and remain opaque as there is no instance of class `GAPEnc` which they can be decoded to. This is justified because, as a coordination layer, SGP2 mainly needs to decode containers (lists or tuples) and numbers; other GAP objects are simply passed as is from one GAP server to another.

DOI: 10.1002/cpe

4.3. The SymGridPar2 programming model

SGP2 exposes parallel patterns to the programmer as *algorithmic skeletons* [15] where the computations coordinated are GAP functions. Figure 8 lists some basic SGP2 skeletons for starting, stopping and calling stateful and stateless GAP servers. These skeletons use the coordination features of HdpH to extend the scope of the GAP binding in that they transparently control GAP servers beyond the current node.

The `startGAP` skeleton initiates starting a stateful GAP server on a remote node (using the primitive `startST`); it returns a future that will eventually, when the server is up and running, yield a handle. The `startGAPEverywhere` skeleton starts pools of stateless GAP servers on all nodes (using the primitive `start`). However, unlike `startGAP`, this skeleton blocks until all GAP servers are up and running, and then returns nothing. The started GAP servers can be terminated by `stopGAP` and `stopGAPEverywhere`; both skeletons block until the servers in question have actually quit.

The `pushGAPCall` skeleton is a generalisation of the primitive `callST` for calling a stateful GAP server on any node, transparently marshalling call and result across the network. The skeleton returns a future, which will eventually hold the result of the call. In contrast, the `parGAPCalls` skeleton is a generalisation of the `call` primitive for calling stateless GAP servers, handling both lists of calls as well as balancing load between all GAP servers in the system (relying on HdpH random work stealing). Unlike `pushGAPCall`, the `parGAPCalls` skeleton blocks and only returns the list of results after the last result has come in; the skeleton will return the empty list if any of the GAP calls failed.

4.3.1. SumEuler in SymGridPar2. Figure 9 shows the implementation of the SumEuler example in SGP2. The `parSumEuler` starts a number of GAP servers before computing the sum of Euler's totients in parallel. The computation divides the input interval into smaller chunks and then proceeds by using the skeleton `parGAPCalls`, converting its results (`GAPObjs` encoding integers) into Haskell integers, and summing up. Incidentally starting GAP servers may take several seconds and hence is usually performed at the start of an application rather than immediately before `parGAPCalls` as here.

The list of GAP calls passed to `parGAPCalls` consists of calls to the GAP function `SumEuler`, the definition of which is passed to every GAP server during the start up phase coordinated by `startGAPEverywhere`. In Haskell, this function definition is a literal string embedded into the Haskell code. Alternatively, the definition can be read from a file.

```
-- Start a stateful GAP server on given (remote) node; returns a future which
-- will yield a server handle once the server is up and running.
startGAP :: GAP → [GAPCall] → Node → Par (IVar GAPHandleST)

-- Start pools of stateless GAP servers on all nodes;
-- blocks until all servers are up and running.
startGAPEverywhere :: GAP → [GAPCall] → Int → Par ()

-- Stop given (remote) stateful GAP server; blocks until server has died.
stopGAP :: GAPHandleST → Par ()

-- Terminate all stateless GAP servers; blocks until all servers have died.
stopGAPEverywhere :: Par ()

-- Post GAP call to given (remote) stateful GAP server; returns a future
-- which will eventually yield the result.
pushGAPCall :: GAPHandleST → GAPCall → Par (IVar (Either GAPError GAPObj))

-- Process list of stateless GAP calls in parallel using work stealing;
-- returns [] if any GAP call caused an error.
parGAPCalls :: [GAPCall] → Par [GAPObj]
```

Figure 8. Example SGP2 skeletons and GAP interface.

DOI: 10.1002/cpe

```
-- Define SumEuler function in GAP.
defSumEuler :: GAPCall
defSumEuler = pack $ unlines [
  "SumEuler := function (lower, upper)",
  "  return Sum ([lower..upper], x -> Phi(x));",
  "end;"]

funSumEuler :: GAPFunction
funSumEuler = pack "SumEuler"

mkIntervals :: Integer -> Integer -> Integer -> [(Integer, Integer)]
mkIntervals lower upper chunksize = go lower
  where
    go lb | lb > upper = []
          | otherwise  = (lb, ub) : go lb'
                         where
                           lb' = lb + chunksize
                           ub  = min upper (lb' - 1)

-- Start 'n' stateless GAP servers everywhere, compute the sum of Euler's
-- totient function on the integral interval ('lower','upper'), evaluating
-- sub-interval of length 'chunksize' in parallel.
parSumEuler :: Int -> GAP -> Integer -> Integer -> Integer -> Par Integer
parSumEuler n gap lower upper chunksize = do
  startGAPEverywhere gap [defSumEuler] n
  let calls = [mkGAPCallFuncList funSumEuler [toGAP l, toGAP u] |
               (l,u) <- mkIntervals lower upper chunksize]
  gapObjs <- parGAPCalls calls
  let result = sum (map fromGAP gapObjs)
  stopGAPEverywhere
  return result
```

Figure 9. SumEuler in SGP2.

4.4. Advanced features

The capabilities of SymGridPar2 described so far match largely those of SymGridPar. SGP2 goes beyond SGP in its support for large architectures. Where SGP assumes a uniform communication latency between any two cores, SGP2's scheduling algorithm can take into account a non-uniform hierarchy of latencies; such latency hierarchies are common in large compute clusters.

Abstract model of hierarchical distances. The coordination DSL HdpH exposes node-to-node latencies abstractly to the programmer as distances in a metric space. More precisely, the distance between nodes is a real-valued binary function satisfying the axioms of *ultrametric spaces*. As a result, the programmer need not be concerned with actual latencies and can instead express task scheduling constraints in terms of relative distances like *here*, *close by* or *far away*. We refer the reader to [58] for more detail about ultrametric distances in HdpH.

Task scheduling HdpH offers two modes of work distribution: on-demand implicit task placement and eager explicit task placement.

On-demand implicit task placement is realised by a distributed work stealing algorithm. Upon creation, a task is stored in the creating node's local task pool. Stored with the task is its *radius*, as chosen by the programmer. The task radius is the maximum distance the task may travel away from its originating node. HdpH's work stealing algorithm is based on random stealing and has two properties relating to radii: the task radius is a strict bound on the distance between victim and thief; thieves prefer to steal tasks within small radii. The radius can be viewed as expressing a *size constraint* on a task, where larger radii mean bigger tasks. Adopting this view, HdpH's work stealing algorithm prefers to keep small tasks near their originating nodes, similar to [59], in the hope of minimising communication overheads related to work stealing.

Many basic SGP2 skeletons, for instance `parGAPCalls`, rely on HdpH work stealing, typically without imposing constraints on work stealing distances; we refer to [4] for examples of SGP2 skeletons that make use of radii for bounding work stealing.

DOI: 10.1002/cpe

On-demand random task placement performs well with irregular parallelism. However, it tends to under-utilise large scale architectures at the beginning of the computation. To combat this drawback, HdpH offers *eager explicit task placement* as a complementary placement mode. Explicit placement ships tasks to selected nodes, where they are executed immediately, taking priority over any implicitly placed tasks. Eager execution implies that these tasks are meant to perform coordination, e. g. create further tasks, rather than actual computation.

The downside of explicit placement is that the programmer has to select target nodes, which may jeopardise portability. Fortunately, HdpH's abstract ultrametric distances offer some help for the purpose of flooding large architectures with work. More precisely, the HdpH runtime identifies an *equidistant basis*, that is, a maximal set of nodes such that any two basis nodes are maximally far apart. At the beginning of a parallel computation, the programmer can explicitly place large tasks at the basis nodes, where each will generate smaller tasks that can be stolen quickly by nearby nodes; stealing may or may not be bounded, depending on the irregularity of the small tasks.

SGP2 offers a number of skeletons implementing such a *two-level task distribution*, for example, the `parMap2Level` and `parMap2LevelRelaxed` skeletons used in Section 5.3, and detailed in [4].

Reliability. The coordination DSL HdpH is designed for *transparent fault tolerance* and [60] presents an alternative work stealing scheduler that monitors nodes and pessimistically replicates tasks that may have been lost to node failure. The fault tolerant work stealing is shown to have low runtime overheads. The fault tolerance properties of HdpH are, however, not yet inherited by SGP2 because the current GAP binding is not fault tolerant.

5. PERFORMANCE EVALUATION

This section reports a performance and interworking evaluation of the HPC-GAP components. The primary comparators are the SumEuler benchmark implementations from the preceding sections, although we investigate some GAP5 concurrency overheads using microbenchmarks. We measure strong scaling, that is, speedups and runtimes, for GAP5 on a multicore (Section 5.1), and for MPI-GAP and SymGridPar2 (Sections 5.2 and 5.4) on Beowulf clusters. Only weak scaling is appropriate for large architectures, and we measure SymGridPar2 on up to 32K cores of the HECToR HPC (Section 5.3).

Section 5.2 demonstrates the interworking MPI-GAP and GAP4, and Section 5.4 demonstrates the interworking SymGridPar2 with both GAP4 and GAP5.

5.1. GAP5 evaluation

We consider both the overheads on sequential programs introduced by managing concurrency in GAP5 and demonstrate the performance gains that can be obtained through parallelisation.

5.1.1. Sources of concurrency overheads. GAP5 exposes essentially the same programming language and libraries as GAP4, augmented with the task and region features described in Section 2. However, supporting multithreading requires many, and often significant, changes even where the functionality is unchanged. While these changes should be transparent to GAP programmers, they affect the performance of purely sequential code. These performance differences can be attributed to one or more of the following factors.

1. Whereas GAP4 has a sequential, generational, compacting garbage collector, GAP5 has a concurrent, non-generational, non-compacting garbage collector. Depending on the workload, either GAP4 or GAP5 can perform better; GAP4 where the generational collection wins out, GAP5 where concurrent collection (with a sufficient number of processors) makes it faster. Generational collection generally wins out when a program allocates a large number of short-lived objects.

DOI: 10.1002/cpe

2. Memory allocation in GAP4 occurs through a very fast bump allocator made possible by the compacting garbage collector. Memory allocation in GAP5 is more complicated in that it has to support concurrent allocation, which incurs a small, but non-zero, overhead. Most allocations occur through the thread-local allocator, which is nearly as fast as the GAP4 bump allocator, but larger chunks of memory cannot be handled this way and require that their allocation be serialised.
3. To ensure memory safety, the GAP5 kernel code had to be instrumented with checks that no GAP object is being accessed without the corresponding lock being held. Like bound checks for arrays, this incurs an unavoidable overhead. While this instrumentation can be disabled if speed is essential, it is generally assumed that users want these checks to be enabled when running parallel code. The instrumentation is currently performed (almost) fully automatically by a separate tool as part of the build process that performs dataflow analysis and attempts to optimise the instrumentation (such as hoisting checks out of loops). This automated instrumentation can be manually overridden where the optimiser fails to correctly identify such possibilities, though so far, we make very little use of that.
4. The current GAP5 implementation does not yet fully support GAP-to-C compilation for all new features. As a result, some core GAP modules that are compiled in GAP4 are still being interpreted in GAP5.
5. A few internal data structures of the GAP kernel had to be redesigned in order to make them thread-safe. One example is the implementation of global variables in GAP, which required non-trivial changes to ensure that multiple threads can access them concurrently, even where they, and their contents, are immutable.

The overhead varies from program to program, and the combined effect of the concurrency overheads is that sequential code typically executes between 10% and 40% more slowly in GAP5 than in GAP4, based on our observations with various synthetic benchmarks and existing GAP code. Sections 5.1.3 and 5.3 report the overheads for specific programs, and it is reassuring to see that the overheads fall as the parallel system scales. We also expect to be able to eliminate the remaining avoidable overheads in the medium term. Nevertheless, our plans for GAP5 include a purely sequential build option for code that cannot effectively exploit parallelism.

5.1.2. Quantifying concurrency overheads. The first GAP 5 concurrency overhead we investigate is that of instrumenting the GAP5 kernel with guards against race conditions. We execute the

```
1  benchmark_base := function()
2    local i;
3    for i in [1..10^9] do
4    od;
5  end;
6
7  benchmark_inc := function()
8    local i, s;
9    s := 0;
10   for i in [1..10^9] do
11     s := s + 1;
12   od;
13 end;
14
15 benchmark_index := function()
16   local i, s;
17   s := [0];
18   for i in [1..10^9] do
19     s[1] := s[1] + 1;
20   od;
21 end;
```

Figure 10. Microbenchmarks to measure GAP5 instrumentation overheads.

DOI: 10.1002/cpe

Table I. GAP5 instrumentation runtime overheads.

	GAP5 (no checks)	GAP5 (checks)	Overhead
`benchmark_base`	3.4 s	3.4 s	0%
`benchmark_inc`	12.8 s	13.7 s	7%
`benchmark_index`	50.7 s	66.5 s	31%

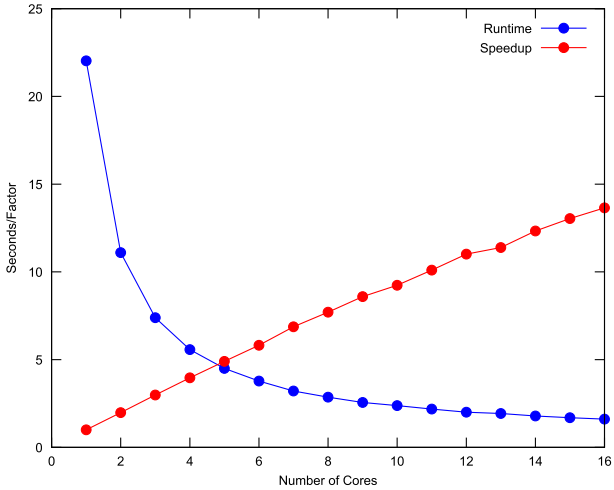


Figure 11. GAP5 SumEuler runtimes and speedups (multicore).

micro-benchmarks in Figure 10 with and without instrumentation, taking the median value over five runs. All of the micro-benchmarks allocate only a small and fixed amount of memory and do not access global variables, so neither garbage collection nor global variable access has a measurable effect on their performance. Measurements in this section are performed on a 64-core machine with AMD Opteron 6376 processors.

The results in Table I show that for `benchmark_base` there is almost no overhead for a basic `for` loop. The `benchmark_inc` result shows that there is a small amount of overhead for performing local variable increments. The `benchmark_index` result shows that there is around a 33% overhead (after subtracting the time for the for loop) for performing increments on an array element. In fact, there are no checks in the C code that performs local variable increments, so the performance differences for `benchmark_inc` are probably due to other effects, such as changes in memory locality as a side effect of instrumentation.

The overhead for `benchmark_index` is not surprising, because for each iteration, three checks are performed. This is suboptimal as our tool inserts two checks for the assignment (one, before checking that the length of the list is at least 1, and another, before performing the assignment) where one would suffice. So the overhead could be reduced by a third.

Garbage collection overhead can vary greatly by workload. Allocating many short-lived objects can result in an overhead of 30% or more; more typical for GAP applications is an overhead in the range of 10%–20%. In cases where allocations of long-lived objects dominate, the GAP5 garbage collector can also be faster than the GAP4 garbage collector, as it can parallelise its work, while the GAP4 garbage collector gains little from its generational features.

5.1.3. SumEuler performance. We investigate the performance of the GAP5 version of SumEuler outlined in Section 2.2 (page 3611). GAP's standard implementation of Euler's totient function relies heavily on cache-based implementation of factorisation routines. This can have unpredictable effects on performance, depending on how number ranges are being broken into chunks and how these chunks are assigned to threads. In order to obtain reproducible results, we therefore use a cache-less implementation of the totient function for this benchmark.

DOI: 10.1002/cpe

Figure 11 shows the runtime and relative speedup of the GAP5 SumEuler up to 16 cores, taking the median value of three successive runs. The sequential SumEuler runs in 16.5 s on GAP4, so GAP5 (22.0 s) is 33% slower.

5.2. *MPI-GAP evaluation*

We investigate the performance of the MPI-GAP version of SumEuler outlined in Section 3.1 (page 3617). The evaluation was conducted on a 32-node Beowulf cluster, with each node having 8-core Intel Xeon E5504 2 GHz processor with 16 GB RAM. One node was designated as a master node, and it distributes chunks of work to the slave nodes, who then execute them on a single-threaded GAP4 instance. To exercise a cluster, the input is larger than that for a single GAP5 instance in Section 5.1.3.

The runtime for a single-threaded GAP4 SumEuler computation is 211.7 s, and for the MPI-GAP version on one node is 217.3 s. So the overhead incurred by the MPI-GAP on a single node is approximately 3% for this benchmark. The MPI-GAP runtime on 32-nodes is 9.9 s. These runtimes are reflected in the speedup graph in Figure 12, which shows near-linear speedups up to a maximum of 22 on 32 cores.

5.3. *SGP2 evaluation*

We investigate the performance of the SymGridPar2 version of SumEuler outlined in Section 4.3.1 (page 16) on the HECToR supercomputer. At the time (2013), HECToR was the UK's largest public HPC with approximately 90 000 cores [61]. Figure 13 studies *weak scaling* of two variants of the SumEuler benchmark from 1 to 1024 HECToR nodes (i.e. 32 to 32K cores). The two variants differ only in their use of the underlying skeleton: One uses `parMap2Level`, and the other uses `parMap2LevelRelaxed` both outlined in Section 4.4.

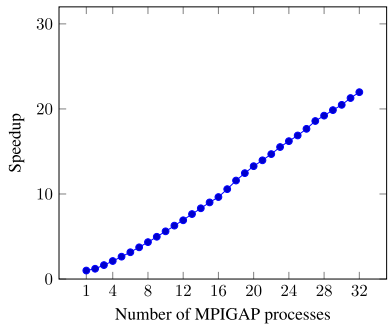


Figure 12. MPIGAP SumEuler speedups (Beowulf cluster).

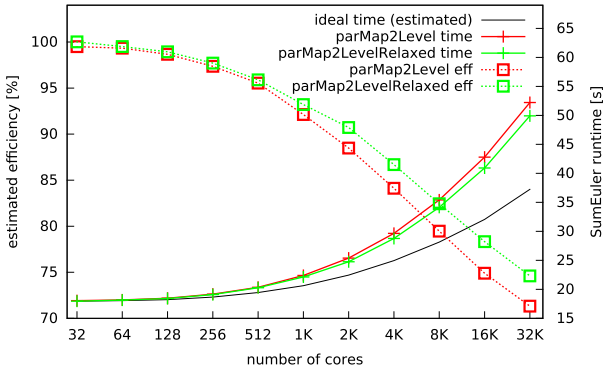


Figure 13. SymGridPar2 SumEuler runtimes and efficiency (weak scaling on HECToR HPC).

DOI: 10.1002/cpe

The benchmarks start with an input interval of 6.25 million integers on 32 cores, doubling the size of the interval as the number of cores doubles, so that on 32K cores, for example, the interval is 6.4 billion integers long. Doubling the size of the input interval increases the amount of work by more than a factor of 2; by sampling the sequential runtime of small subintervals, we estimate a runtime curve for ideal scaling.

The runtime graphs in Figure 13 show that the two benchmarks do not scale perfectly. However, even on 32K cores, their runtimes are still within 50% of the ideal. Efficiency is calculated with respect to the estimated ideal scaling, and the curves show that efficiency declines steadily, yet remains above 70% even on 32K cores. These graphs also show that the `parMap2LevelRelaxed` skeleton offers a small efficiency advantage (of 3–4%) over `parMap2Level`. This is probably because `parMap2LevelRelaxed` copes better with the irregularity of the benchmark in the final stages of the computation, because unlike `parMap2Level`, it can utilise nodes that have run out of work early.

5.4. HPC-GAP interworking

Section 5.2 demonstrated MPI-GAP coordinating single-threaded GAP4 servers, and this section focuses on the performance implications of SymGridPar2 coordinating GAP4 (version `4.7.6`) and multi-threaded GAP5 (version `alpha_2015_01_24`) servers. We investigate the speedup of SumEuler on the interval $[1, 10^8]$ on a Beowulf cluster with 256 cores, distributed over 16 nodes (Intel Xeon CPUs, 2 GHz, 64 GB RAM per node). The input interval is subdivided evenly into subintervals, and the SGP2 instances distribute the resulting SGP2 tasks by work stealing. When co-ordinating GAP4, each SGP2 instance further subdivides the current task interval and distributes calls to function `SumEulerSeq` (Figure 1) to 15 single-threaded GAP4 servers. When co-ordinating GAP5, each SGP2 instance calls a single GAP5 server (running on 15 cores) which in turn subdivides the interval and uses the GAP5 `RunTask` primitive to evaluate calls to `SumEulerSeq` on subintervals in parallel.

Table II records optimal values for the numbers of SGP2 tasks, GAP calls per SGP2 task and GAP tasks per GAP call. These optima are determined by exhaustive experiment. The table also reports sequential runtimes for GAP4 and GAP5, from which the average runtime of a GAP task, that is, mean time to compute `SumEulerSeq`, is derived. Given the discussion of GAP5 overheads in Section 5.1, it is unsurprising that the sequential GAP5 runtime is 24% greater than for GAP4.

Table II. SGP2 with GAP4/GAP5 optimal configurations and runtimes.

	Seq. runtime	SGP2 tasks	GAP calls per SGP2 task	GAP tasks per GAP call	Mean GAP task runtime
SGP2/GAP4	2160.0 s	320	15	1	450 ms
SGP2/GAP5	2836.1 s	400	1	300	24 ms

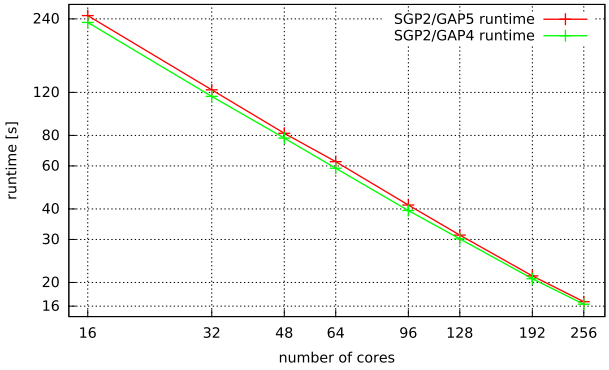


Figure 14. SGP2 with GAP4 and GAP5 SumEuler runtimes (Beowulf).

DOI: 10.1002/cpe

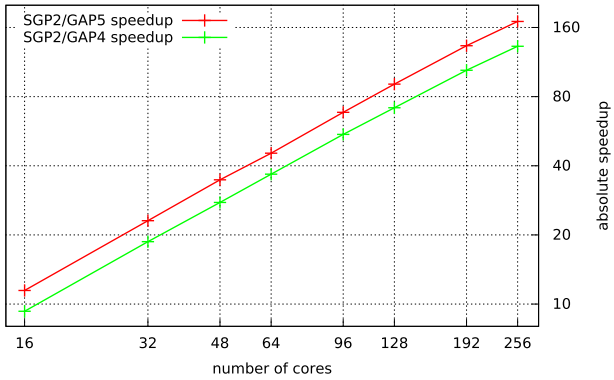


Figure 15. SGP2 with GAP4 and GAP5 SumEuler speedups (Beowulf).

Figures 14 and 15 show log/log plots of runtime and absolute speedup for SGP2/GAP4 and SGP2/GAP5. The numbers of SGP2 tasks, GAP calls and GAP tasks are fixed to the optimal values from Table II, and the runtime data is the median over 11 runs. We find that variance, measured as standard error, is consistently between 1% and 2%.

Figure 14 shows that while SGP2/GAP4 has consistently lower runtime, the advantage falls from about 24% on one core to 6% on one node to only about 2% on 16 nodes. The absolute speedups in Figure 15 are computed with respect to the sequential runtime of the respective GAP version. The curves show that both systems scale similarly, yet the speedup of SGP2/GAP5 is consistently about 25% greater than for SGP2/GAP4. On 1 node, SGP2/GAP5 achieves an absolute speedup of 11.5, corresponding to 76% efficiency: ideal speedup is 15 because only 15 cores are available to GAP while the 16th core is dedicated to SGP2. On 16 nodes, SGP2/GAP5 efficiency drops to 71%, corresponding to a speedup of 170.

Symbolic computations are memory hungry, and on large shared-memory architectures, SGP2/GAP5 can use a small number of GAP processes (possibly just one), where SGP2/GAP4 must use as many GAP4 processes as there are cores. Sampling memory residency for SumEuler (using Unix tools like `ps`) consistently shows a total per-node residency of SGP2/GAP4 (i.e. 15 GAP4 processes plus 1 SGP2 process) of about 2 GB. In contrast, the residency of SGP2/GAP5 (one GAP5 process plus one SGP2 process) is only 240 MB, or approximately 1/8th of the residency of SGP2/GAP4.

In conclusion, SGP2/GAP5 is able to almost entirely eliminate the GAP5 overheads by achieving better speedups even on mid-scale parallel architectures. While we have not been able to demonstrate SGP2/GAP 5 outperforming SGP2/GAP4, we anticipate this may happen on larger architectures. A second advantage of SGP2/GAP5 over SGP2/GAP4 is greatly reduced memory residency on large shared-memory architectures.

6. CASE STUDIES

We present two case studies to demonstrate the capability of the HPC-GAP components to deliver good parallel performance for typically irregular algebraic computations on both NUMA and HPC architectures.

6.1. Orbits in GAP5

Enumerating an orbit is a typical algebraic computation and can be described as follows.

Definition 1 (Orbit of an element)
Let X be a set of points, G a set of generators, $f : X \times G \to X$ a function (where $f(x, g)$ will be denoted by $x \cdot g$) and $x_0 \in X$ a point in X. The *orbit* of x_0, denoted by $\mathcal{O}_G(x_0)$, is the smallest set $T \subseteq X$ such that

DOI: 10.1002/cpe

```
   Data: x_0 ∈ X, g_1, g_2, ..., g_k : G
   Result: A hash table T containing the orbit of the element x_0
1  T := {x_0} (a hash table); U = [x_0] (a list);
2  while U is not empty do
3      x := head of U; Remove x from U;
4      for i ← 1 to k do
5          y := x · g_i;
6          if y ∉ T then
7              Add y to T;
8              Append y to the end of U;
```

Figure 16. Sequential Orbit pseudocode.

1. $x_0 \in T$
2. for all $y \in T$ and all $g \in G$ we have $y \cdot g \in T$.

Often the set X can be very large and is not given *a priori*.

The sequential orbit algorithm in Figure 16 maintains a hash table T of orbit points discovered so far and a list U of unprocessed points. The hash table facilitates a fast duplicate test (line 6). Initially both T and U contain only x_0, and thereafter, we iterate, in each step removing the first point x from U, applying all generators $g \in G$ to x and adding any new points to both T and U. The computation terminates when there are no unprocessed points (line 2).

As an example of how the algorithm works, let $X = \{1, 2, 3, 4, 5\}$, $x_0 = 2$, $G = \{g_1, g_2\}$ and f be given by the following table:

x	1	2	3	4	5
$x \cdot g_1$	1	3	2	4	5
$x \cdot g_2$	5	2	4	3	1

The following table shows the steps in evaluating the orbit T of x_0:

Step	1	2	3	4
T	{2}	{2, 3}	{2, 3, 4}	{2, 3, 4}
U	[2]	[3]	[4]	∅

6.1.1. The parallel orbit skeleton. At first glance, it may seem that the sequential Orbit algorithm can be parallelised by carrying out step 5 for all generators and all points in U independently. However, synchronisation is required to avoid duplicates (steps 6–8), and this requires communication among the tasks that apply generators to different elements of U.

Our Parallel Orbit skeleton uses two different kinds of threads:

- *Hash server* threads, which use hash tables to determine which points are duplicates, and
- *Worker* threads that obtain chunks of points from hash servers and apply generators to them, so producing new points.

This means that we have explicit two-way communication between each hash server and each worker, but no communication between different hash servers or different workers. Where there are multiple hash servers, we use a distribution hash function to decide which hash server is responsible for storing each point. Figures 17 and 18 show the pseudocode for a worker thread and a hash server thread, respectively. Initially, all hash servers and input queues are empty, and all workers are idle. We first feed a single point into one of the hash servers, which immediately creates some work. This hash server then sends this work to a worker which, in turn, produces more points that are then sent to the corresponding hash servers, bootstrapping the computation. The number of points in the system increases and, eventually, all input queues become completely full, and the system as a whole becomes busy. Towards the end of the computation, fewer new points are discovered. Eventually, all generators will have been applied to all points and the computation terminates. Now, all workers are idle again and the hash servers collectively know all the points in the orbit. In our GAP implementation, we maintain one channel for the input queue of each hash server. Any worker

DOI: 10.1002/cpe

```
Data: the set G, the action function X × G → X, the number h of hash servers and a
      distribution hash function f : X → {1, ..., h}
1 while true do
2 |   get a chunk C of points ;
3 |   R := a list of length h of empty lists;
4 |   for x ∈ C do
5 |   |   for g ∈ G do
6 |   |   |   y := x · g;
7 |   |   |   append y to R[f(y)];
8 |   for j ∈ {1, ..., h} do
9 |   |   schedule sending R[j] to hash server j;
```

Figure 17. Parallel Orbit worker pseudocode.

```
Data: A chunk size s
1 initialise a hash table T;
2 while true do
3 |   get a chunk C of points from our input queue;
4 |   for x ∈ C do
5 |   |   if x ∉ T then
6 |   |   |   add x to T;
7 |   |   |   if at least s points in T are unscheduled then
8 |   |   |   |   schedule a chunk of size s points for some worker;
9 |   if there are unscheduled points in T then
10|   |   schedule a chunk of size < s points for some worker;
```

Figure 18. Parallel Orbit hash server pseudocode.

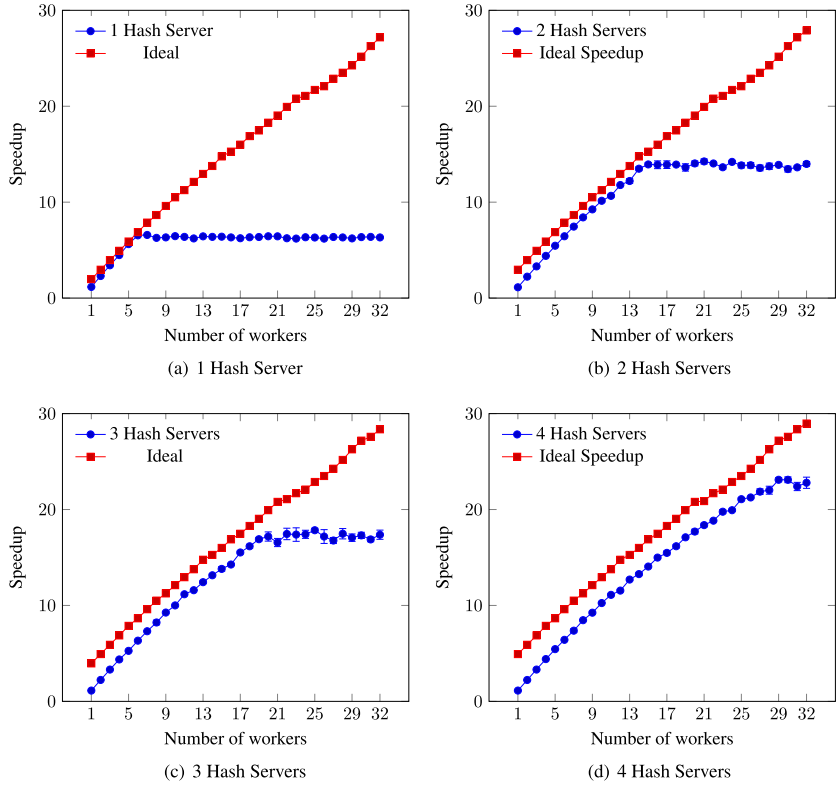
can write to any of these channels. We also maintain a single shared channel for the work queue. Because of the chunking mechanism, this is not a bottleneck.

6.1.2. Performance results. This section presents a preliminary evaluation of our Parallel Orbit skeleton on a shared-memory machine consisting of four AMD Opteron 6376 processors, each comprising 16 cores (giving us a 64-core system). Each core runs at 1.4 GHz and has 16 Kb of private L1 data cache. In addition, each processor has 8×64 Kb of L1 instruction caches and 8×2 Mb of L2 caches (shared between two cores) and 2×8 Mb of L3 caches (shared between eight cores). As an instance of an Orbit enumeration, we consider the sporadic finite simple Harada-Norton group in its 760-dimensional representation over the field with two elements, acting on vectors. The set X of all possible points consists of about 10^{228} elements. The size of the orbit is 1.4 million points, and we are using ten random generators. On each figure, we consider mean speedups over five runs of the same instance. We have also added error bars in the cases where error margin was notable.

The sequential runtime of this orbit calculation on our test machine was 250 s. Additional experiments with larger instances and a sequential runtime of 1514 s produced similar runtimes and speedups. Figure 19 shows the absolute speedups over the sequential Orbit algorithm that we have obtained with 1 to 4 hash server threads, and 1 to 32 worker threads. We can observe good speedups, up to 23 with four hash servers and 28 worker threads (so, using 32 cores in total). We can also observe that about seven workers produce enough points to fully load one hash server. Also, for a fixed number of hash servers h, the speedups grow almost linearly with the increase in the number of workers. Once the number of workers reaches $7h$, we do not observe further increase in speedups. Note further that, on our testbed, it is not possible to obtain linear speedups, because of frequency scaling of cores when multiple cores are used at the same time. Therefore, the figure also shows the ideal speedups that could be obtained. The ideal speedup on n cores was estimated by taking the runtime of the sequential Orbit algorithm in one thread[‡] and then dividing it by n. We obtain similar results on other shared-memory architectures [5].

[‡]With $n - 1$ parallel threads executing a dummy spin locking function that does not use any memory, to scale down the frequency.

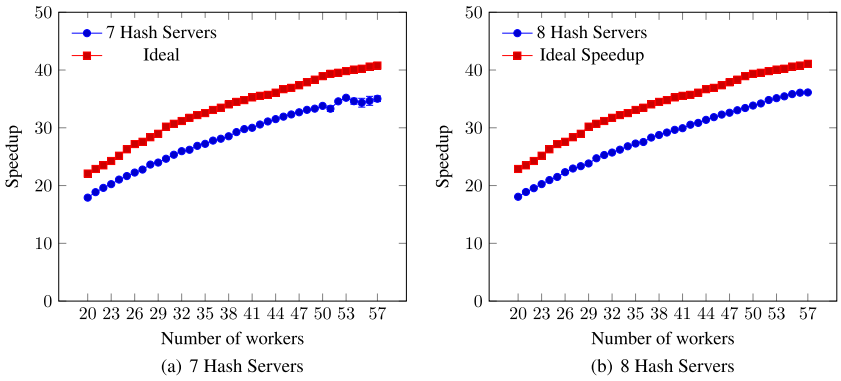
DOI: 10.1002/cpe



(a) 1 Hash Server (b) 2 Hash Servers

(c) 3 Hash Servers (d) 4 Hash Servers

Figure 19. GAP5 Harada-Norton Orbit speedups (NUMA).



(a) 7 Hash Servers (b) 8 Hash Servers

Figure 20. GAP5 Harada-Norton Orbit speedups with seven or eight hash servers (NUMA).

To test the scalability further, Figure 20 shows the speedups obtained on the same machine with seven and eight hash servers, and up to 57 workers, therefore using all 64 cores in the system. The speedups steadily increase in both cases from 18 to 35 (with seven hash servers) and 36 (with eight hash servers). The fact that the speedup obtained is suboptimal can be attributed both to the problem itself, because in these cases, the bootstrapping and finishing phases (where not enough points are available to utilise all workers) become dominant, and to the hardware, because there are only 16 memory channels, which limits the memory access when more hash servers and workers are used.

6.2. Hecke algebras in SGP2

SGP2 has been used to compute matrices of invariant bilinear forms in the representation theory of Hecke algebras. In the following, we will only sketch this case study and present the main per-

DOI: 10.1002/cpe

formance results; the reader is referred to [6] for a detailed report and to [62] for background on the problem.

6.2.1. Problem: finding invariant bilinear forms. We consider Hecke algebras $\mathcal{H}$ of exceptional type E_m ($m = 6, 7, 8$), generated by m generators $T_1, \ldots, T_m$.

An n-dimensional *representation* ρ of $\mathcal{H}$ is an R-algebra homomorphism from $\mathcal{H}$ to $M_n(R)$, the R-algebra of $n \times n$ matrices over $R = \mathbb{Z}[x, x^{-1}]$, the ring of Laurent polynomials in indeterminate x. Being a homomorphism, ρ is uniquely determined by the images of the generators $T_1, \ldots, T_m$. Thus, each n-dimensional representation ρ can be succinctly expressed by $n \times n$ matrices $\rho(T_1), \ldots, \rho(T_m)$ of polynomials; in a slight abuse of terminology, we call the $\rho(T_i)$ *generators*.

For any given ρ, there exists a non-trivial symmetric matrix $Q \in M_n(R)$ such that

$$Q \cdot \rho(T_i) = \rho(T_i)^{\mathrm{T}} \cdot Q \tag{1}$$

for all generators $\rho(T_i)$. We call Q the *matrix of an invariant bilinear form*. It suffices to compute Q for so-called *cell representations* ρ, of which there are only finitely many. Depending on ρ, finding the invariant bilinear form Q may require substantial computation. The table below lists the number of cell representations ρ, the range of dimensions of ρ and the range of degree bounds of Laurent polynomials in Q. These numbers, and hence, the difficulty of the problem, vary by several orders of magnitude reflecting the extreme irregularity of symbolic computations.

Hecke algebra type	E_6	E_7	E_8
Cell representations	25	60	112
Dimension of reps	6–90	7–512	8–7168
Degree of Laurent polys	29–54	45–95	65–185

6.2.2. Sequential algorithm. In principle, Q can be computed by viewing Equation (1) as a system of linear equations and solving for the entries of Q. However, solving linear systems over $\mathbb{Z}[x, x^{-1}]$ is too expensive to obtain solutions for high dimensional representations. Instead, we solve the problem by interpolation. We view each entry of Q as a Laurent polynomial with $u - l + 1$ unknown coefficients, where l and u are the lower and upper degree bounds of polynomials in Q. Solving Equation (1) at $u - l + 1$ data points will provide enough information to compute the unknown coefficients by solving linear systems over the rationals instead of $\mathbb{Z}[x, x^{-1}]$. To avoid computing with very large rational numbers (because of polynomials of high degree), we solve homomorphic images of Equation (1) modulo small primes and use the Chinese Remainder Theorem to recover the rational values. The algorithm takes as input m generators $\rho(T_i)$ of dimension n, lower and upper degree bounds l and u and a finite set of small primes P. From the degree bounds, we construct a set V_{lu} of $u - l + 1$ small integers to be used as data points for interpolation.

The algorithm runs in three phases:

1. For all $p \in P$ and $v \in V_{lu}$, GENERATE a modular interpolated solution Q_{vp} of (1) by instantiating the unknown x with v and solving the resulting system modulo p.
2. REDUCE the modular matrices Q_{vp} to a matrix of Laurent polynomials Q as follows. First, for all $v \in V_{lu}$, construct a rational interpolated solution Q_v of (1) by Chinese remaindering. Second, construct each Laurent polynomial q_{ij} in Q by gathering the (i, j)-entries of all Q_v and solving a rational linear system for the coefficients q_{ij}. Because Q is symmetric, there are $(n + 1)n/2$ such systems, each of dimension $u - l + 1$.
3. For all generators $\rho(T_i)$, CHECK that the resulting Q satisfies (1) over $\mathbb{Z}[x, x^{-1}]$.

The theory of Hecke algebras admits a particularly efficient way to GENERATE Q_{vp}. Instead of solving a linear system, the rows of Q_{vp} can be computed by *spinning a basis* (i.e. multiplying a basis vector with certain matrix products).

6.2.3. Parallel algorithm. Each of the three phases of the sequential algorithm to compute Q contains significant amounts of parallelism. Figure 21 shows the parallel structure, where n is the dimension of Q and the m generators $\rho(T_i)$, l and u are lower and upper degree bounds of the

DOI: 10.1002/cpe

Laurent polynomials and P is the set of primes used in the GENERATE phase. Note that there is a sequential duplicate filtering phase at the beginning of the REDUCE phase.

6.2.4. Evaluation. A systematic evaluation of the SymGridPar2 implementation on all cell representations of E_6, E_7 and the smallest 16 representations of E_8 is reported in [6]. Here, we present performance results for E_8 on two different architectures: a 48 core NUMA server (Cantor), and 1024 cores (distributed over 32 nodes) of the HECToR supercomputer [61].

Runtime. Figure 22 plots the runtime of SGP2 with 40 GAP workers on Cantor. It also plots the total work, that is, the cumulative runtime of all tasks, and the time spent in the sequential part of the REDUCE phase. The reported times reflect single experiments as a statistically significant number of repetitions would be prohibitively expensive. We observe that the amount of (total and sequential) work and the parallel runtime appear to be correlated, yet oscillate noisily between representations. This is due to the high degree of irregularity in the number of REDUCE tasks.

Speedup. Figure 23 plots the estimated speedup, computed as total work divided by parallel runtime, on Cantor and on two configurations of HECToR (for representations 11 to 15). On Cantor, speedups stabilise just below 40, which is the maximum expected with 40 GAP workers. With

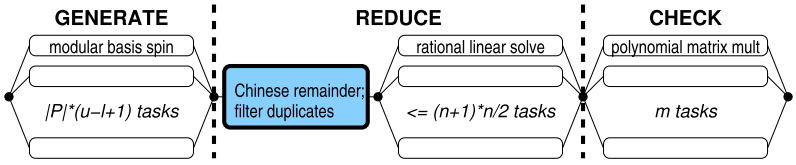


Figure 21. Structure of parallel algorithm for computing Q.

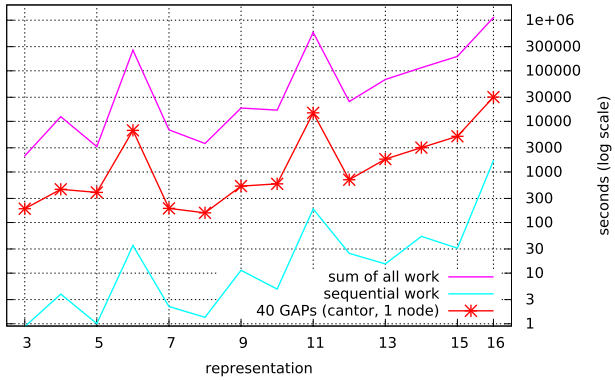


Figure 22. SymGridPar2 computing Hecke algebra Q runtimes (NUMA and HPC).

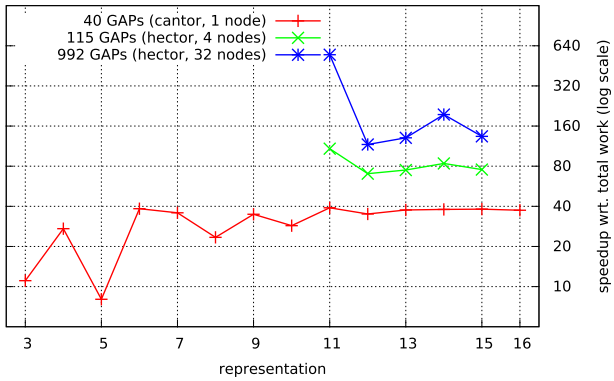


Figure 23. SymGridPar2 computing Hecke algebra Q speedups (NUMA and HPC).

DOI: 10.1002/cpe

HECToR's 4 node configuration (128 cores, running 115 GAP workers), SGP2 achieves speedups of around 80; yet with the 32 node configuration (1024 cores, 992 GAP workers), speedup hovers around 160; representation 11 is the exception achieving a speedup of 548.

7. CONCLUSION

We have provided a systematic description of HPC-GAP, a thorough re-engineering of the GAP computational algebra system for the 21st Century, which incorporates mechanisms to deal with parallelism at the multicore, distributed cluster and HPC levels. At the GAP5 base level, this has involved major work to remove single-threaded assumptions in the runtime system and libraries. We have developed MPIGAP to exploit ubiquitous small clusters and designed and implemented a highly sophisticated coordination system for HPC systems, SymGridPar2, which uses parallel Haskell to coordinate large-scale GAP computations. The result is the most advanced, free and open source, computational algebra system currently in existence.

We evaluate the scalability and performance of the HPC-GAP components using a common SumEuler benchmark on a range of systems from small-scale multicores to HPC platforms. Key results include showing that while the GAP5 concurrency overheads are typically between 10% and 40% (Section 5.1); these overheads are almost completely ameliorated on medium-scale architectures (Section 5.4); we demonstrate good weak scaling to 32K cores on the HECToR HPC (Section 5.3). The Orbit and Hecke Algebra case studies convincingly demonstrate the capability of HPC-GAP to manage large and highly irregular algebraic computations, for example, achieving a maximum speedup of 548 using 992 GAP instances on 1024 cores of the HECToR HPC (Section 6). The datasets for these experiments are available in [2].

There is enormous potential for further exploitation of this work. GAP5 is gaining widespread adoption in the algebraic computation community. We are exploring alternate parallelism options for GAP, like a PGAS model with UPC-GAP. Moreover, reliability is increasingly an issue at HPC scale, and here, the statelessness of many algebraic computations means failed computations can be safely recomputed. The HdpH-RS extension tracks the location of computations and reinstates any that may have failed [60, 63].

ACKNOWLEDGEMENTS

This work has been partially supported by EPSRC grants HPC-GAP (EP/G05553X), AJITPar (EP/L000687/1) and by EU grants ICT-288570 (ParaPhrase), ICT 287510 (RELEASE), ICT-644235 (RePhrase), by the EU COST Action IC1202: Timing Analysis On Code-Level (TACLe).

REFERENCES

1. GAP Group. GAP – Groups, algorithms, and programming, 2007. (Avialable from: http://www.gap-system.org) [Accessed on October 2015].
2. Behrends R, Maier P, Janjic V. HPC-GAP open access repository, 2015. DOI: 10.5281/zenodo.34012.
3. Behrends R, Konovalov A, Linton S, Lübeck F, Neunhöffer M. Parallelising the computational algebra system GAP. *Proceedings of the 4th International Workshop on Parallel Symbolic Computation, PASCO 2010, july 21-23, 2010*, Grenoble, France, 2010; 177–178. DOI: 10.1145/1837210.1837239.
4. Maier P, Stewart R, Trinder PW. Reliable scalable symbolic computation: the design of SymGridPar2. *Computer Languages, Systems & Structures* 2014; **40**(1):19–35.
5. Janjic V, Brown CM, Neunhoeffer M, Hammond K, Linton SA, Loidl HW. Space exploration using parallel orbits. *ParCo 2013: Parallel Computing, Advances in Parallel Computing*, Vol. 25, IOS Press: Munich, Germany, 2014; 225–232.
6. Maier P, Livesey D, Loidl HW, Trinder P. High-performance computer algebra: a Hecke algebra case study. *Euro-Par 2014*, Springer: Porto, Portugal, 2014; 415–426.
7. Besche HU, Eick B, O'Brien EA. A millennium project: constructing small groups. *International Journal of Algebra and Computation* 2002; **12**(5):623–644.
8. Maplesoft. Maple, 2015. (Available from: http://www.maplesoft.com/products/Maple/academic/) [Accessed on October 2015].
9. Wolfram Research. Mathematica, 2015. (Available from: http://www.wolfram.com/mathematica/) [Accessed on October 2015].

DOI: 10.1002/cpe

10. Mathworks. Matlab, 2015. (Available from: http://uk.mathworks.com/products/matlab/) [Accessed on October 2015].
11. Computational algeb group School of Mathematics, Statistics University of Sydney. MAGMA computational algebra system. (Available from: http://magma.maths.usyd.edu.au/magma/) [Accessed on October 2015].
12. System Sage Mathematical Software. Sage, 2015. (Available from: http://www.sagemath.org/) [Accessed on October 2015].
13. Kaltofen EL. Fifteen years after DSC and WLSS2: what parallel computations I do today. *PASCO 2010: International Workshop on Parallel Symbolic Computation*, ACM Press: Grenoble, France, 2010; 10–17.
14. Mattson TG, Sanders BA, Massingill BL. *Patterns for Parallel Programming* (1st edn). Addison-Wesley: Boston, MA, USA, 2004. ISBN 0321940784.
15. Cole MI. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press: Cambridge, MA, USA, 1989. ISBN 0-262-53086-4.
16. McCool M, Reinders J, Robison A. *Structured Parallel Programming*. Morgan Kaufmann: Burlington, MA, USA, 2012. ISBN 0124159931.
17. Campbell C, Johnson R, Miller A, Toub S. *Parallel Programming with Microsoft. NET — Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft, 2010. (Available from: http://msdn.microsoft.com/en-us/library/ff963553.aspx) [accessed on October 2015].
18. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Communications of the ACM* 2008; **51**(1):107–113.
19. Apache Hadoop NextGen MapReduce (YARN), 2013. (Available from: http://hadoop.apache.org/docs/current2/hadoop-yarn/hadoop-yarn-site/YARN.html) [Accessed on 10 February 2014].
20. Bacci B, Danelutto M, Orlando S, Pelagatti S, Vanneschi M. P^3L: A structured high-level parallel language, and its structured support. *Concurrency — Practice and Experience* 1995; **7**(3):225–255.
21. Botorog GH, Kuchen H. Efficient parallel programming with algorithmic skeletons. *Euro-Par '96: Proceedings of the Second International Euro-par Conference on Parallel Processing, Lyon, France*, LNCS 1123. Springer: Lyon, France, 1996; 718–731. DOI: 10.1007/3-540-61626-8 95.
22. Darlington J, Field AJ, Harrison PG, Kelly Paul HJ, Sharp DWN, Wu Q. Parallel programming using skeleton functions. *PARLE '93: Proceedings of the 5th International Parle Conference on Parallel Architectures and Languages Europe*, Springer: London, UK, 1993; 146–160.
23. Enmyren J, Kessler CW. SkePU: A multi-backend skeleton programming library for multi-GPU systems. *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, HLPP '10, ACM: New York, NY, USA, 2010; 5–14.
24. Ciechanowicz P, Poldner M, Kuchen H. The Münster skeleton library Muesli—a comprehensive overview, European Research Center for Information Systems: Münster, Germany, 2009. (Available from: http://www.ercis.org/sites/default/files/publications/2012/ercis_wp_no_7.pdf) [Accessed on 18 December 2015].
25. Benoit A, Cole M, Gilmore S, Hillston J. Flexible skeletal programming with Eskel. *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, Springer: Berlin, Heidelberg, 2005; 761–770.
26. Dumas JG, Gautier T, Pernet C, Saunders B. LinBox founding scope allocation, parallel building blocks, and separate compilation. *ICMS'2010: Mathematical Software*, LNCS 6327, Springer: Kobe, Japan, 2010; 77–83. DOI: 10.1007/978-3-642-15582-6 16.
27. Chamberlain B, Callahan D, Zima H. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications* 2007; **21**(3):291–312.
28. Charles P, Grothoff C, Saraswat VA, Donawa C, Kielstra A, Ebcioglu K, von Praun C, Sarkar V. X10: An object-oriented approach to non-uniform cluster computing. *OOPSLA 2005: Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press: San Diego, USA, 2005; 519–538.
29. Allen E, Chase D, Hallett J, Luchangco V, Maessen JW, Ryu S, Steele GL Jr., Tobin-Hochstadt S. The Fortress language specification version 1.0. *Technical Report*, Sun Microsystems, Inc.: Santa Clara, CA, United States, 2008.
30. UPC Consortium. Unified parallel C language spec. v1.3. *Teachnical Report*, Lawrence Berkeley National Lab: Berkeley, CA, USA, 2013.
31. Numrich RW, Reid J. Co-Array Fortran for parallel programming. *ACM SIGPLAN Fortran Forum* 1998; **17**(2):1–31.
32. Grabmeier J, Kaltofen E, Weispfenning V. *Computer Algebra Handbook*. Springer, 2003. DOI: 10.1007/978-3-642-55826-9. ISBN 978-3-540-65466-7.
33. Roch JL, Villard G. Parallel computer algebra. *Technical Report*, IMAG: France, 1997. Tutorial at ISSAC 1997: International Symposium on Symbolic and Algebraic Computation.
34. Maza MM, Stephenson B, Watt SM, Xie Y. Multiprocessed parallelism support in ALDOR on SMPs and multicores. *International Workshop on Parallel Symbolic Computation*, PASCO '07, ACM Press: London, Ontario, Canada, 2007; 60–68.
35. Küchlin W. PARSAC-2: A parallel SAC-2 based on threads. *AAECC-8: Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, LNCS 508, Springer, Tokyo, Japan, 1991; 341–353. DOI: 10.1007/3-540-54195-0_63.
36. Schreiner W, Neubacher A, Hong H. The design of the saclib/paclib kernels. *Journal of Symbolic Computation* 1995; **19**(1–3):111–132.
37. Schreiner W, Mittermaier C, Bosa K. Distributed Maple: parallel computer algebra in networked environments. *Journal of Symbolic Computation* 2003; **35**(3):305–347.

DOI: 10.1002/cpe

38. Char BW. Progress report on a system for general-purpose parallel symbolic algebraic computation. *ISSAC 1990: International Symposium on Symbolic and Algebraic Computation*, ACM Press: Tokyo, Japan, 1990; 96–103. DOI: 10.1145/96877.96902.
39. Cooperman G. GAP/MPI: facilitating parallelism. *Proceedings of DIMACS Workshop on Groups and Computation II 28, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, AMS, Piscataway, NJ, USA, 1997.
40. Cooperman G. TOP-C: task-oriented parallel C for distributed and shared memory. *Workshop on Wide Area Networks and High Performance Computing*, LNCS 249, Springer: Essen, Germany, 1999; 109–117. DOI: 10.1007/BFb0110081.
41. Cooperman G, Tselman M. New sequential and parallel algorithms for generating high dimension Hecke algebras using the condensation technique. *ISSAC 1996: International Symposium on Symbolic and Algebraic Computation*, ACM Press: Zurich, Switzerland, 1996; 155–160. DOI: 10.1145/236869.236927.
42. Kunkle D., Slavici V., Cooperman G. Parallel disk-based computation for large, monolithic binary decision diagrams. *PASCO 2010: International Workshop on Parallel Symbolic Computation*, ACM Press: Grenoble, France, 2010; 63–72. DOI: 10.1145/1837210.1837222.
43. Sibert EE, Mattson HF, Jackson P. Finite field arithmetic using the connection machine. *CAP 1990: International Workshop on Computer Algebra and Parallellism*, LNCS 584, Springer: Ithaca, USA, 1992; 51–61. DOI: 10.1007/3-540-55328-2-4.
44. Roch JL, Sénéchaud P, Françoise Siebert-Roch F, Villard G. Computer algebra on MIMD machine. *ISSAC 1998: International Symposium on Symbolic and Algebraic Computation*, LNCS 358, Springer: Rome, Italy, 1989; 423–439. DOI: 10.1007/3-540-51084-2_40.
45. Neun W, Melenk H. Very large Gröbner basis calculations. *International Workshop on Computer Algebra and Parallellism, CAP 1990*, LNCS 584, Springer: Ithaca, USA, 1992; 89–99. DOI: 10.1007/3-540-55328-2_7.
46. Loustaunau P, Wang PY. Towards efficient parallelizations of a computer algebra algorithm. *Frontiers of Massively Parallel Computation*, IEEE, McLean, VA, USA, 1992; 67–74. DOI: 10.1109/FMPC.1992.234903.
47. Maplesoft. Maple grid computing toolbox. (Available from: http://www.maplesoft.com/products/toolboxes/GridComputing) [Accessed on October 2015].
48. Linton S, Hammond K, Konovalov A, Brown C, Trinder PW, Loidl HW, Horn P, Roozemond D. Easy composition of symbolic computation software using SCSCP: a new Lingua Franca for symbolic computation. *Journal of Symbolic Computation* 2013; **49**:95–19.
49. Halstead RH Jr. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 1985; **7**(4):501–538.
50. Konovalov A, Linton S. Parallel computations in modular group algebras. *PASCO 2010: International Workshop on Parallel Symbolic Computation*, ACM Press: Grenoble, France, 2010; 141–149.
51. Gastineau M. TRIP home page. Web page. (Available from: http://www.imcce.fr/Equipes/ASD/trip/trip.php) [Accessed on October 2015].
52. Gastineau M, Laskar J. Development of TRIP: fast sparse multivariate polynomial multiplication using burst tries. *ICCS 2006: Computational Science*, LNCS 3992, Springer: Reading, UK, 2006; 446–453. DOI: 10.1007/11758525_60.
53. Metzner T, Radimersky M, Sorgatz A, Wehmeier S. User's guide to macro parallelism in MuPAD 1.4.1, 1999.
54. GridMathematica. (Available from: http://www.wolfram.com/gridmathematica/) [Accessed on October 2015].
55. Cooperman G. Parallel GAP: mature interactive parallel computing. *Conference on Groups and Computation III*, De Gruyter, Columbus, OH, USA, 2001; 123–138.
56. Al Zain A, Hammond K, Trinder PW, Linton S, Loidl HW, Constanti M. SymGrid-Par: designing a framework for executing computational algebra systems on computational grids. *International Conference on Computational Science, ICCS 2007*, LNCS 4488, Springer, Beijing, China, 2007; 617–624. DOI: 10.1007/978-3-540-72586-2_90.
57. Maier P, Trinder PW. Implementing a high-level distributed-memory parallel Haskell in Haskell. *IFL 2011: International Symposium on Implementation and Application of Functional Languages*, LNCS 7257, Springer: Lawrence, Kansas, USA, 2012; 35–50. DOI: 10.1007/978-3-642-34407-7_3.
58. Maier P, Stewart R, Trinder PW. The HdpH DSLs for scalable reliable computation. *Haskell 2014*, ACM Press: Gothenburg, Sweden, 2014; 65–76. DOI: 10.1145/2633357.2633363.
59. Janjic V, Hammond K. Granularity-aware work-stealing for computationally-uniform grids. *CCGRID 2010*, IEEE, Melbourne, Australia, 2010; 123–134. DOI: 10.1109/CCGRID.2010.49.
60. Stewart R. Reliable massively parallel symbolic computing: fault tolerance for a distributed Haskell. *Ph.D. Thesis*, School of Mathematical and Computer Sciences, Heriot-Watt University, 2013.
61. Edinburgh Parallel Computing Center (EPCC). HECToR National UK Super Computing Resource, Edinburgh, 2013. (Available from: http://www.hector.ac.uk/) [Accessed on 18 December 2015].
62. Geck M. Hecke algebras of finite type are cellular. *Inventiones Mathematicae* 2007; **169**:501–517.
63. Stewart R, Maier P, Trinder P. Transparent fault tolerance for scalable functional computation. *Journal Functional Programming* 2016. Accepted for publication.

DOI: 10.1002/cpe