Easy Composition of Symbolic Computation Software using SCSCP: A New Lingua Franca for Symbolic Computation

S. Linton [a], K. Hammond [a], A. Konovalov [a], C. Brown [a], P.W. Trinder [b], H.-W. Loidl [b], P. Horn [c], D. Roozemond [d]

[a] *School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, Scotland*

[b] *School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, EH14 4AS, Scotland*

[c] *Fachbereich Mathematik, Universität Kassel, Heinrich Plett Straße 40, 34132 Kassel, Germany*

[d] *School of Mathematics and Statistics, Carslaw F07, University of Sydney NSW 2006 Australia*

Abstract

We present the results of the first four years of the European research project SCIEnce – Symbolic Computation Infrastructure in Europe (`http://www.symbolic-computation.org`), which aims to provide key infrastructure for symbolic computation research. A primary outcome of the project is that we have developed a new way of combining computer algebra systems using the Symbolic Computation Software Composability Protocol (SCSCP), in which both protocol messages and data are encoded in the OpenMath format. We describe the SCSCP middleware and APIs, outline implementations for various Computer Algebra Systems (CAS), and show how 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. Additionally, we present several domain-specific parallel skeletons that capture commonly used symbolic computations. To ease use and to maximise inter-operability, these skeletons themselves are provided as SCSCP services and take SCSCP services as arguments.

Key words: OpenMath, SCSCP, interface, coordination, parallelism, skeletons, SymGrid-Par, CASH

⋆ "SCIEnce – Symbolic Computation Infrastructure in Europe" (`www.symbolic-computation.org`) is supported by EU FP6 grant RII3-CT-2005-026133.

1. Introduction

A key requirement in symbolic computation is to efficiently combine computer algebra systems (CAS) in order to collectively solve complex problems that cannot be easily addressed by any single system on its own. In addition, there is often a requirement to have a CAS as a back-end for mathematical databases and web/grid/cloud services, or to combine multiple instances of the same or different CAS as part of a parallel computation.

There are many possible combinations of CAS that are both useful and valuable. Examples include: GAP and Maple in CHEVIE for handling generic character tables (Geck et al., 1996); Maple and the PVS theorem prover to obtain more reliable results (Adams et al., 2001); GAP and nauty in GRAPE for fast graph automorphisms (Soicher, 2004); and GAP as a service for the ECLiPSe constraint programming system for symmetry-breaking in search (Gent et al., 2003). In all these cases, interfacing to a CAS that already possesses the required functionality is far less work than re-implementing the functionality in the "home" system.

Even within a single CAS, users may need to combine local and remote instances of the CAS for a number of reasons, including: accessing remote system features that are not supported in the local operating system; accessing large (and changing) databases; remote access to the latest development version of the CAS or to a configuration at the user's home institution; licensing restrictions permitting only online services, etc. A common quick solution is to cut-and-paste between telnet sessions and web browsers. It would, however, be both more efficient and more flexible to combine local and remote computations so that remotely obtained results can be plugged immediately into the locally running CAS.

CAS authors have attempted to address these user needs in various ways. For example, a CAS may write input files for another program and invoke it; the other program will then write input to the CAS in a file and exit; and finally, the CAS will read this input and return a result. This works, but has fairly serious limitations. A better setup might allow the CAS to interact with other programs while they run and provide a separate interface to each possible external system. The SAGE system (Stein et al., 2010) is essentially built around this approach. However, achieving this is a major programming challenge, and an interface will break as soon as the other system changes its I/O format, for example.

Furthermore, individual CPU cores have essentially stopped increasing in power, but multicore systems are becoming more numerous. A typical workstation now has 4 to 8 cores, and this is only the beginning of the multicore/manycore revolution (Held et al., 2006). If we want to solve larger problems in future, it is essential for us to exploit multiple processors in a way that gives good parallelism for minimal programmer/user effort.

The EU Framework 6 SCIEnce project "SCIEnce – Symbolic Computation Infrastructure in Europe" is a major 6-year project that brings together CAS developers and experts in computational algebra, OpenMath, and parallel computations. It aims to address the problems outlined above by designing a common standard interface that may be used for combining computer algebra systems (and any other compatible software). Our vision is an easy, robust and reliable way for users to create and consume services implemented in any compatible system, ranging from generic services (e.g. evaluation of a string or an OpenMath object) to specialised ones (e.g. a lookup in a specific database; or executing a certain procedure). We have developed a simple lightweight XML-based remote procedure

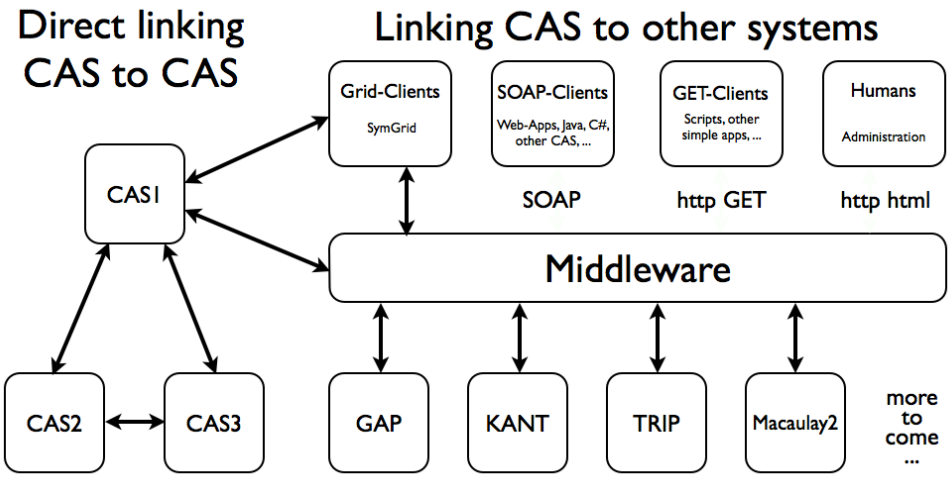

Fig. 1. SCSCP framework

call protocol called **SCSCP** (*Symbolic Computation Software Composability Protocol*) in which both data and instructions are represented as OpenMath objects. SCSCP is now implemented in several computer algebra systems (see Section 2.2 for an overview) and has APIs that make it easy to add SCSCP interfaces to more systems and to develop SCSCP middleware that can translate SCSCP services to a variety of possible clients (see Figure 1; arrows correspond to the SCSCP communication). The full specification of SCSCP can be found online at `http://www.symbolic-computation.org/scscp`. Another important outcome of the project is the development of middleware for parallel computations, SymGrid-Par, which is capable of orchestrating SCSCP-compliant systems into a heterogeneous system for distributed parallel computations (Section 6.4).

This paper extends our earlier ISSAC 2010 paper (Linton et al., 2010). One of its major enhancements is the presentation of the SCSCP API for Haskell and CASH (the Computer Algebra SHell) as an example of an application that can be built on this API. The main contributions of this paper are as follows:

(1) We show how the SCSCP interface can be used to connect different computer algebra systems. Specifically, we demonstrate this approach using several different examples that collectively show the flexibility of the SCSCP approach and demonstrate some SCSCP-specific features and benefits.
(2) We introduce CASH, the Computer Algebra SHell, and show how it can be used to integrate Haskell applications and a CAS, giving examples that show how CASH can be used to call a CAS from Haskell and vice-versa.
(3) We show how to use small-scale parallelism from the CASH command-line to prototype parallel code.
(4) We show how to define *domain-specific*, large-scale parallel skeletons for computer algebra in the Eden dialect of Haskell. In particular, we describe new multiple homomorphic images and orbit skeletons.
(5) We expose these parallel skeletons as SymGrid-Par services, that can be called directly both from the CASH shell and from within the shell of an SCSCP-compliant computer algebra client.

We give an overview of these tools below. First, we briefly characterise the underpinning OpenMath data encoding and the SCSCP protocol (Section 2). Then we outline SCSCP interfaces in two different systems, one open source and one commercial, and provide references for existing implementations in other systems (Section 3). After that we describe several examples that demonstrate the flexibility of the SCSCP approach and some SCSCP specific features and benefits (Section 4). In Section 5, we introduce CASH, the Computer Algebra SHell: an interactive API for calling computer algebra systems from Haskell, and vice-versa. We introduce several SCSCP-compliant tools for parallel computations in various environments (Section 6) and present domain-specific, parallel skeletons, building on the SCSCP interface (Section 7), before concluding (Section 8).

2. A Computer Algebra Lingua Franca

In order to connect different CAS it is necessary to speak a common language, i.e., to agree on a common way of marshalling mathematical semantics. The obvious choice is *OpenMath* (Buswell et al., 2004), a well-established standard that has previously been used in similar contexts (Caprotti and Cohen, 1999; Caprotti et al., 2000). In order to actually use this in practice, it is also necessary to agree on a suitable communication protocol that allows for, e.g., executing computations on a remote system or defining remote objects. The protocol developed in the SCIEnce project is called SCSCP, the Symbolic Computation Software Composability Protocol (Freundt et al., 2009c).

2.1. OpenMath

OpenMath is a flexible language for describing mathematical objects. The entire OpenMath semantics is encapsulated in terms of *symbols* which are defined in *Content Dictionaries* (CDs) and which are strictly separated from the language itself. So, for example, the "normal" addition operation falls under the name *plus* in the *arith1* CD. OpenMath was designed to be efficiently used by computers, and may be represented using several different encodings. The XML representation is the most commonly used, but there also exists a binary representation and a more human-readable representation called Popcorn (Horn and Roozemond, 2009). A large number of CDs are available at the OpenMath website `http://www.openmath.org`, such as *polyd1* for multivariate polynomials, *group4* for cosets and conjugacy classes, etc.

2.2. SCSCP

Figure 2 classifies the OpenMath symbols that have been defined for SCSCP into several groups (Freundt et al., 2009a). SCSCP is built around three types of messages that send remote procedure calls and either return the result or report a failure. Messages have identifiers, and may carry additional information, such as call options and information messages or standard error reports. Special procedures are used to discover information about the procedures that are provided by the SCSCP server, and these are supported by special symbols. SCSCP also allows remote objects to be handled by reference so that clients may work with objects of a type that do not exist in their own system at all (see the example in Section 4.2). For example, to represent the number of conjugacy classes of a group only a knowledge of integers is required, not a knowledge of groups.

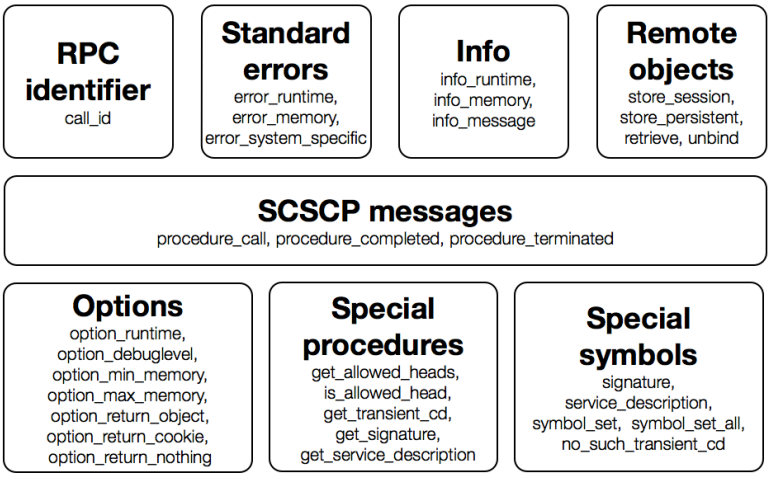

Fig. 2. Symbols defined in OpenMath content dictionaries for SCSCP

The SCSCP protocol is socket-based and by default uses TCP/IP port number 26133, as assigned by the *Internet Assigned Numbers Authority* (IANA). It has been described in earlier publications (Freundt et al., 2008, 2009b; Horn and Roozemond, 2009) and its specification (version 1.3 at the moment of writing) is freely available online (Freundt et al., 2009c). The advantage of supporting SCSCP is that any system that implements SCSCP can immediately connect to all other SCSCP-compliant systems, and may be used as the building block for more advanced cluster and grid/cloud infrastructures (see Section 6). This avoids the need for special cases and minimizes repeated effort.

3. Building blocks for CAS composition

In this section, we briefly describe the implementation of the SCSCP protocol for two systems: GAP (GAP Group, 2008) and MuPAD (SciFace Software, 2007). The main aim of this section is to show that SCSCP is a standard that may be implemented in different ways by different CAS, taking into account their own design principles.

3.1. GAP

In the GAP system, support for OpenMath and SCSCP is implemented in two GAP packages: `OpenMath` and `SCSCP`. The `OpenMath` package (Costantini et al., 2010) is an OpenMath phrasebook for GAP: it converts OpenMath to GAP and vice versa, and provides a framework that users may extend with their own private content dictionaries. The `SCSCP` package (Konovalov and Linton, 2010b) implements SCSCP, using the GAP packages `OpenMath`, `IO` (Neunhöffer, 2010) and `GAPDoc` (Lübeck and Neunhöffer, 2008). It ensures SCSCP communication using three types of messages as outlined on Figure 2 and allows to run GAP as either an SCSCP server or an SCSCP client. The server may be started interactively from the GAP session or as a GAP daemon. When the server accepts a connection from the client, it starts the "accept-evaluate-return" loop, which:

(1) accepts the `"procedure_call"` message and looks up the appropriate GAP function (which should be declared by the service provider as an SCSCP procedure);

(2) evaluates the result (or produces a side-effect); and
(3) either replies with the `"procedure_completed"` message or returns an error in the `"procedure_terminated"` message.

The SCSCP client performs the following basic actions:
(1) establishes connection with the server;
(2) sends the `"procedure_call"` message to the server;
(3) either waits for the completion of the procedure call (blocking version) or checks its completion later (non-blocking version); and
(4) fetches the result from a `"procedure_completed"` message or enters the break loop in the case of a `"procedure_terminated"` message.

We have used this basic functionality to build a set of instructions for parallel computations using the SCSCP framework. This allows the user to send several procedure calls in parallel and then collect all results, or to pick up the first available result. We have also implemented the master-worker parallel skeleton in the same way (see Section 6.2).

A demo SCSCP server can be tried at `chrystal.mcs.st-andrews.ac.uk`, port 26133. It runs the development version of the GAP system plus a selection of public GAP packages. Further details, downloads, and a manual with examples are available online (Konovalov and Linton, 2010b).

3.2. MuPAD

There are two main aspects to the MuPAD SCSCP support: the MuPAD `OpenMath` package (Horn, 2010) and the SCSCP server wrapper for MuPAD. The former offers the ability to parse, generate, and handle OpenMath in MuPAD and to consume SCSCP services, the latter provides access to MuPAD's mathematical abilities as an SCSCP service. The current MuPAD end-user license agreement, however, does not generally allow providing MuPAD computational facilities over the network. We therefore focus on the open-source `OpenMath` package, which can be freely downloaded (Horn, 2010).

OpenMath Parsing and Generation: Two functions are available to convert an OpenMath XML string into a tree of MuPAD `OpenMath::` objects: `OpenMath::parse(`*str*`)` which parses the string *str*, and `OpenMath::parseFile(`*fname*`)` which reads and parses the file named *fname*. Conversely, a MuPAD expression can be converted into its OpenMath representation using `generate::OpenMath`. Note that it is not necessary to directly use OpenMath in MuPAD if the SCSCP connection described below is used: the package takes care of marshaling and unmarshaling in a way that is completely transparent to the MuPAD user.

SCSCP Client Connection: The call `s := SCSCP(`*host*`,` *port*`)` creates an SCSCP connection object that can subsequently be used to send commands to the SCSCP server. Note that the actual connection is initiated on construction by starting the Java program WUPSI (Horn and Roozemond, 2010a) which is bundled with the `OpenMath` package. This uses an asynchronous message-exchange mode, and can therefore be used to introduce background computations. The command `s::compute(...)` can then be used to actually compute something on the server (`s(...)` is equivalent). Note that it may be necessary to wrap the parameter in `hold(...)` to prevent premature evaluation on the client side. The connection may also be used asynchronously via the `send` and `retrieve` commands: `a := s::send(...)` returns an integer which may be used to identify the computation. The

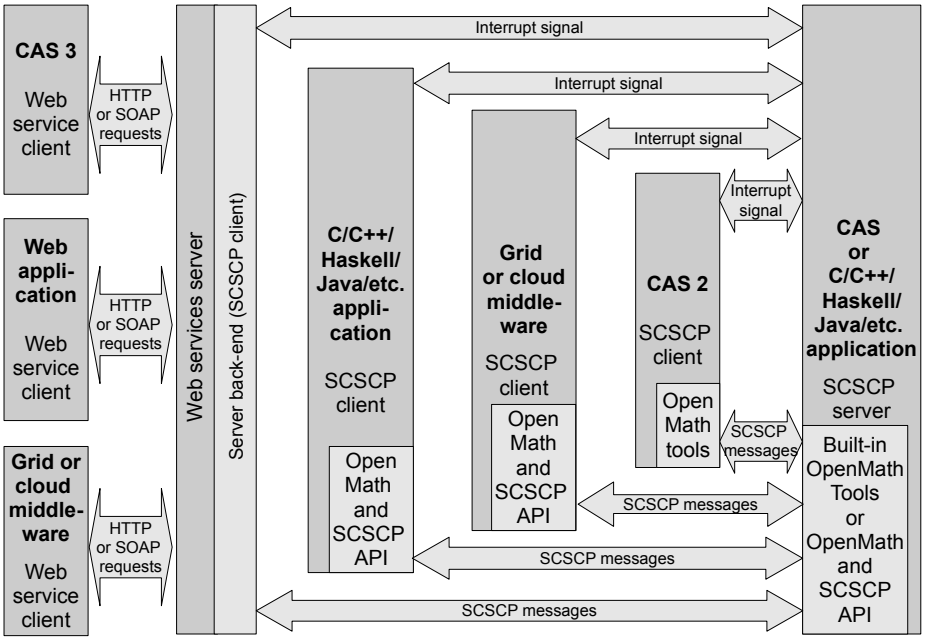

Fig. 3. Combining software applications using SCSCP

result may subsequently be retrieved using `s::retrieve(a)`. Normally, `retrieve` will return `FAIL` if the result of the computation is not yet computed, but a second parameter may be supplied in order to force the call to block.

3.3. Other Implementations of SCSCP

The SCIEnce project has produced a Java library (Horn and Roozemond, 2010b) that acts as a reference implementation for systems developers who would like to implement SCSCP for their own systems. This library is freely available under the Apache2 license. In addition to GAP and MuPAD, SCSCP has also been implemented in two other systems participating in the SCIEnce project: KANT (Freundt and Lesseni, 2010) and Maple (Maplesoft, 2010) (the latter implementation is currently a research prototype and not available in the Maple release). There are third-party implementations for TRIP (Gastineau, 2009), Magma (Cannon and Bosma, 2008) (as a wrapper application, by D. Roozemond), and Macaulay2 (Grayson and Stillman, 2010) (as Macaulay2 packages OpenMath and SCSCP by D. Roozemond), a prototype SCSCP client for the Coq proof assistant (Komendantsky et al., 2010) and a prototype SCSCP client for the Mathematica system by P. Horn. As intended, SCSCP thus allows a large range of CAS and other software applications to interact in various contexts as described on Figure 3.

4. Examples of CAS-to-CAS communication

In this section we provide a number of examples which demonstrate the features and benefits of SCSCP, such as flexible design, composition of different CAS, working

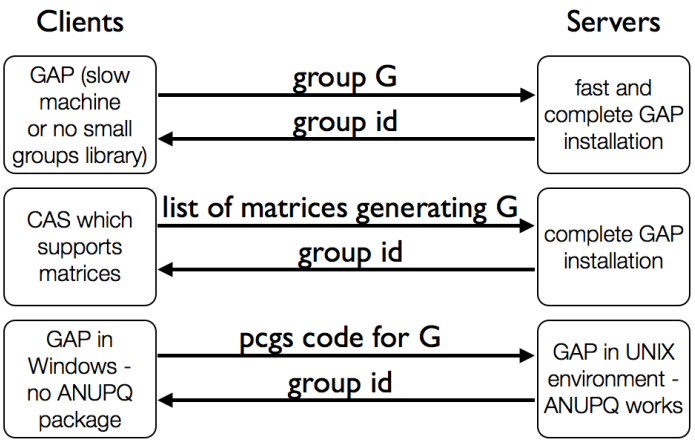

Fig. 4. Three approaches to group identification service

with remote objects and speeding up computations. More examples can be found in e.g. (Freundt et al., 2009b, 2008) and on the web sites for the individual systems or their SCSCP extensions.

4.1. GAP

In order to illustrate the flexibility of our approach, we will describe three possible ways to set up GAP SCSCP server as outlined in 3.1 to provide different procedures for the same kind of problems.

The GAP Small Groups Library (Besche et al., 2008) contains all groups of orders up to 2000, except groups of order 1024. The GAP command `SmallGroup(n,i)` returns the `i`-th group of order `n`. Moreover, for any group G of order $1 \leq |G| \leq 2000$ where $|G| \notin \{512, 1024\}$, GAP can determine its *library number*: the pair `[n,i]` such that G is isomorphic to `SmallGroup(n,i)`. This is in particular the most efficient way to check whether two groups of "small" order are isomorphic or not. Let us consider now how we can provide a group identification service with SCSCP. When designing an SCSCP procedure to identify small groups, we first need to decide how the client should transmit a group to the server. Figure 4 describes three possible scenarios (there may be more), and for each of those we will outline simple steps needed for the design and provision of the SCSCP services within the provided framework.

Case 1. A client supports permutation groups (for example, a client is a minimalistic GAP installation without the Small Groups Library). In this case the conversion of the group to and from OpenMath will be performed straightforwardly, so that the service provider only needs to declare the function `IdGroup` as an SCSCP procedure (under the same or different name) before starting the server:

```
gap> InstallSCSCPprocedure("IdGroup",IdGroup);
InstallSCSCPprocedure : IdGroup installed.
```

The client may then call this, obtaining a record with the result in its `object` component:

```
gap> EvaluateBySCSCP("IdGroup",[SymmetricGroup(6)],"scscp.st-and.ac.uk",26133);
rec( attributes := [ [ "call_id", "hp0SE18S" ] ], object := [ 720, 763 ] )
```

Case 2. A client supports matrices, but not matrix groups. In this case, the service provider may declare the SCSCP procedure `IdGroupByGens` which constructs a group generated by its arguments and return its library number:

```
gap> IdGroupByGens := gens -> IdGroup( Group( gens ) );;
gap> InstallSCSCPprocedure("IdGroupByGens",IdGroupByGens);
InstallSCSCPprocedure : IdGroupByGens installed.
```

Note that validity of any input and the applicability of the `IdGroup` method to the constructed group will be automatically checked by GAP during the execution of the procedure on the SCSCP server, so there is no need to add such checks to this procedure (though they may be added to produce more a informative error message).

Case 3. A client supports groups in some specialised representation (for example, groups given by pc-presentation in GAP). Indeed, for groups of order 512 the Small Groups Library contains all 10494213 non-isomorphic groups of this order and allows the user to retrieve any group by its library number, but it does not provide an identification facility. However, the GAP package ANUPQ (O'Brien et al., 2006) provides a function `IdStandardPresented512Group` that performs the latter task. Because the ANUPQ package only works in a UNIX environment it is useful to design an SCSCP service for identification of groups of order 512 that can be called from within GAP sessions running on other platforms (note that the client version of the SCSCP package for GAP does work under Windows). Now the problem reduces to the encoding of such a group in OpenMath. Should it, for example, be converted into a permutation representation, which can be encoded using existing content dictionaries or should we develop a new content dictionary for groups in such a representation? Luckily, the SCSCP protocol provides enough freedom for the user to select his/her own data representation. Since we are interfacing between two copies of the GAP system, we are free to use a GAP-specific data format, namely the *pcgs code*, an integer that describes the *polycyclic generating sequence* (*pcgs*) of the group, to pass the data to the server (see the GAP manual and (Besche and Eick, 1999) for more details).

First we create a function that takes the *pcgs code* of a group of order 512 and returns the number of this group in the GAP Small Groups library and declare it as an SCSCP procedure with the same name:

```
gap> IdGroup512 := function( code )
>    local G, F, H;
>    G := PcGroupCode( code, 512 );
>    F := PqStandardPresentation( G );
>    H := PcGroupFpGroup( F );
>    return IdStandardPresented512Group( H );
>    end;;
gap> InstallSCSCPprocedure("IdGroup512",IdGroup512);
InstallSCSCPprocedure : IdGroup512 installed.
```

For convenience, the client may be supplied with a function that is specialised to use the correct server port, and which checks that the transmitted group is indeed of order 512:

```
gap> IdGroup512Remote:=function( G )
>    local code, result;
>    if Size(G)<>512 then Error("|G|<>512\n");fi;
>    code := CodePcGroup( G );
>    result := EvaluateBySCSCP("IdGroup512",[code],"scscp.st-and.ac.uk",26133);
>    return result.object;
>    end;;
```

Now the call to `IdGroup512Remote` returns the result in the standard `IdGroup` notation:

```
gap> IdGroup512Remote( DihedralGroup( 512 ) );
[ 512, 2042 ]
```

4.2. GAP and Macaulay2

We now consider interaction between GAP and Macaulay2 (Grayson and Stillman, 2010). Macaulay2 is "a software system devoted to supporting research in algebraic geometry and commutative algebra," that is particularly well known for its efficient procedures for Gröbner bases. We have implemented OpenMath and the SCSCP protocol as packages in the Macaulay2 system, and they have been available in the stable branch since late 2009. All support for OpenMath symbols was implemented directly in the Macaulay2 language, to allow for easy maintenance and extensibility. Macaulay2 is fully SCSCP 1.3 compatible and can act both as a server and as a client. The server is multithreaded so it can serve many clients at the same time, and supports storing and retrieving of remote objects. The client was designed in such a way as to disclose remote computation using SCSCP with minimal interaction from the user. It supports convenient creation and handling of remote objects, as demonstrated below.

An example of a GAP client calling a Macaulay2 server for the Gröbner basis computation, has been described before (Freundt et al., 2008). Although this 2008 implementation used a prototype wrapper implementation of an SCSCP server for Macaulay2 — rather than the full internal implementation that we have now — it nicely demonstrates the possible gain of connecting computer algebra systems using SCSCP.

Below we present an example (produced using the current implementation) of a Macaulay2 SCSCP client calling a remote GAP server. We assume the GAP server has been started (it could even be set up by another GAP expert rather than the Macaulay2 user who may be unfamiliar with GAP). We present only the Macaulay2 side of the example in what follows. First, we load the OpenMath and SCSCP packages and establish a connection to the GAP server that accepts and evaluates OpenMath objects.

```
i1 : loadPackage "SCSCP"; loadPackage "OpenMath";

i3 : GAP = newConnection "127.0.0.1"
o3 = SCSCP Connection to GAP (4.dev) on scscp.st-and.ac.uk:26133
o3 : SCSCPConnection
```

We demonstrate the conversion of an arithmetic operation to OpenMath syntax (note the abbreviated form Macaulay2 uses to improve legibility of XML expressions), and evaluate that expression in GAP.

```
i4 : openMath 1+2
o4 = <OMA
       <OMS cd="arith1" name="plus"
       <OMI "1"
       <OMI "2"
o4 : XMLnode

i5 : GAP <== openMath 1 + 2
o5 = 3
```

We then create two matrices in Macaulay2 (suppressing the output of the creation of the second one), and create a remote object `G` in GAP representing the group they generate.

```
i6 : m1 = id_(QQ^10)^{1,6,2,7,3,8,4,9,5,0}
o6 = | 0 1 0 0 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 1 0 0 0 |
     | 0 0 1 0 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 1 0 0 |
     | 0 0 0 1 0 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 0 1 0 |
     | 0 0 0 0 1 0 0 0 0 0 |
     | 0 0 0 0 0 0 0 0 0 1 |
     | 0 0 0 0 0 1 0 0 0 0 |
     | 1 0 0 0 0 0 0 0 0 0 |
              10        10
o6 : Matrix QQ   <--- QQ

i7 : m2 = id_(QQ^10)^{1,0,2,3,4,5,6,7,8,9};
i8 : G = GAP <=== matrixGroup({m1,m2})
o8 = << Remote GAP object >>
o8 : RemoteObject
```

So at this point in the example `G` is a remote object: Macaulay2 does not have any real information about it, but the remote server (GAP) knows it is a group with some generating matrices. When we ask for the size of the group, Macaulay2 simply creates a new object representing the order of `G`. Finally, evaluating this object in GAP gives the number of elements in the group generated by those matrices.

```
i9 : size G
o9 = << Remote GAP object >>
o9 : RemoteObject

i10 : GAP <== size G
o10 = 10080
```

One of the most important features of this example is that, despite the fact that Macaulay2 has no support for groups at all, using OpenMath and SCSCP we can still create an object that represents a group and obtain useful information about it.

4.3. MuPAD

OpenMath and SCSCP have been implemented in MuPAD in the `OpenMath` package. First, we demonstrate the conversion of $1 + a\sin(x)$ to OpenMath, both as an "internal" OpenMath object, and as its XML representation.

```
>> package("OpenMath"):
>> 1+a*sin(x)
a*sin(x) + 1
>> om := OpenMath(%)
arith1.plus(1, arith1.times(transc1.sin($x), $a))
>> OpenMath::toXml(om)
<OMA>
  <OMS cd='arith1' name='plus'/>
  <OMI>1</OMI>
  <OMA>
    <OMS cd='arith1' name='times'/>
    <OMA>
      <OMS cd='transc1' name='sin'/>
      <OMV name='x'/>
    </OMA>
    <OMV name='a'/>
  </OMA>
</OMA>
```

Now we consider factoring the product of two shifted Swinnerton-Dyer polynomials. This particular polynomial, p in the transcript below, has 49 terms and is of degree 96. It takes 38 seconds to factor if we let MuPAD perform this calculation by itself:

```
>> swindyer := proc(plist) [input abbreviated]
>> R := Dom::UnivariatePolynomial(x,Dom::Rational):
>> p1 := R(swindyer([2,3,5,7,11])):
>> p2 := R(subs(swindyer([2,3,5,7,13,17])),x=3*x-2):
>> p := p1 * p2:
>> nterms(p), degree(p)
49, 96
>> st := time(): F1 := factor(p): time()-st
38431
```

Now we establish an SCSCP connection to a machine 400km away running KANT (Daberkow et al., 1997). Again, the KANT server may have been setup by a KANT expert, and the current user only needs to be aware of the MuPAD side of the transaction. Using this remote server to factor the polynomial is much faster:

```
>> package("OpenMath"):
>> kant := SCSCP("scscp.math.tu-berlin.de",26133):
>> st:=rtime():
>> F2:=kant::compute(hold(factor)(p)):
>> rtime()-st
1221
```

Establishing the connection, marshalling and unmarshalling the objects, sending them over the network, and the actual KANT computation took only 1.2 seconds in total.

This demonstrates the flexibility of the SCSCP approach: the most appropriate system can be used for each task. Users are no longer restricted to performing all aspects of a required computation in a single system that may provide substandard support for some of the required operations.

5. CASH: the Computer Algebra SHell

CASH (the Computer Algebra SHell) (Brown et al., 2010) is an interface that provides direct access to SCSCP-based CA services from an interactive Haskell shell (Peyton Jones and Hammond, 2003). The main benefit to the CA user is the immediate availability of cutting-edge programming language features and a rich set of libraries and tools, provided by a very active research community. CASH provides a simple and effective interface for CAS users to parallelise their algorithms employing easy-to-use parallelism primitives for either small- or large-scale distribution (see Section 6.3 where we develop several domain-specific skeletons written in Haskell, suitable for large-scale distribution). Finally, CASH profits from a highly optimising compiler, the Glasgow Haskell Compiler (GHC), in terms of sequential program performance.

CASH uses the GHCi interpreter, which is part of the Glasgow Haskell Compiler (GHC) distribution (GHC Team, 2010), to provide a Haskell-side shell for accessing computer algebra systems via SCSCP. The underlying libraries provide SCSCP and OpenMath APIs, commands to be used from the interactive front-end to establish a connection to, and interact with, an SCSCP-enabled system, and support for datatypes not natively available in Haskell, e.g., elements of finite fields.

For example, in the CASH shell we may define two matrices `m1` and `m2` over the prime field $GF(3)$. In all CASH examples we use GAP as the underlying CAS. Wherever possible we use the same notation that is used in GAP, so here `Z(3)` denotes the generator of the multiplicative group of $GF(3)$.

```
*Cash> let m1 = [[Z(3)^0, Z(3)^0],[Z(3),0*Z(3)]]
*Cash> let m2 = [[Z(3), Z(3)],[Z(3),0*Z(3)]]
```

Now we can use CASH to compute in GAP multiplicative groups generated by each of these two matrices and then determine their intersection. Note that actual SCSCP calls are embedded into `group` and `intersection`, and that in this particular example the GAP server provides an SCSCP procedure for `intersection` that returns the result as a list of elements (if you run the same example in GAP, the result would be displayed as `<group of 2x2 matrices of size 2 in characteristic 3>`).

```
*Cash> group(m1)
Group([ [ [ Z(3)^0, Z(3)^0 ], [ Z(3), 0*Z(3) ] ] ])
*Cash> group(m2)
Group([ [ [ Z(3), Z(3) ], [ Z(3), 0*Z(3) ] ] ])
*Cash> intersection(group(m1), group(m2))
[ [ [ Z(3)^0, 0*Z(3) ], [ 0*Z(3), Z(3)^0 ] ],
  [ [ Z(3), 0*Z(3) ], [ 0*Z(3), Z(3) ] ] ]
```

Here, the result of each call to `group` operation is a remote object (as specified in SCSCP), existing in the GAP server and only conveniently displayed by CASH as `Group(...)`.

We now give two examples of using CASH that show the usefulness of being able to call CAS functionality from a Haskell environment without having to step out of the functional paradigm.

5.1. Example: Greatest Common Divisor

In this section we give an example of how to use CA functionality directly from CASH. We initialise the SCSCP session, specifying the host and the port number for the connection. Then we define two input polynomials `p1` and `p2` and demonstrate that we can factorise them and compute their GCD in Haskell using the polynomial arithmetic in GAP. Note that this code uses a factorisation function on the GAP side, but implements a simple version of the GCD algorithm on the Haskell side.

```
*Cash> initServer (server "localhost" (Just 26133))
*Cash> let p1 = polyFromString "x_1^3-x_1"
*Cash> factors p1
[x_1-1,x_1,x_1+1]
*Cash> let p2 = polyFromString "x_1^2+x_1"
*Cash> factors p2
[x_1,x_1+1]
*Cash> myGcd p1 p2
x_1^2+x_1
```

Our implementation of a generic greatest common divisor (GCD) function, `myGcd`, first calls existing factorisation functions in the CAS (here, GAP) and then combines the results in Haskell (although in a real application it would be possible to simply call GCD in GAP). The approach taken here highlights some important design features: first, we use Haskell's overloading mechanism to define a generic GCD implementation; and second, the compute-intensive parts, that is, the factorisation and polynomial operations, are all delegated to the underpinning CAS. The `myGcd` algorithm first performs, as we will explain further, two SCSCP calls to factorize its inputs `x` and `y`. It then calls `bagInter` (see below) on the Haskell side, to identify all the common factors. Finally, the GCD is computed by folding the generic multiplication operation, implemented as another SCSCP call over the list of common factors (if it is non-empty).

```
-- generic GCD computation
myGcd :: (Num a, Factorisable a) => a -> a -> a
myGcd x y = let  xs = factors x
                 ys = factors y
                 zs = xs `bagInter` ys
            in if null zs then fromInteger 1 else foldl1 (*) zs

-- intersection on multi-sets (bags)
bagInter :: (Eq a) => [a] -> [a] -> [a]
bagInter [] _ = []
bagInter (x:xs) ys | elem x ys = x:(bagInter xs (delete x ys))
                   | otherwise = bagInter xs ys
```

Marshalling and Unmarshalling Polynomials: The first issue that we face is how to marshal/unmarshal polynomials. While there are several OpenMath content dictionaries for polynomials, SCSCP does not restrict our freedom to choose our own data representation. For the purposes of this example we simply implement `Polynomial`s as Haskell `String`s, which will be easily converted by GAP to polynomials. Admittedly, this limits the inter-operability of the algorithm, but on the other hand also reduces the marshalling overhead. We define polynomials as part of the `OMType` type which represents the abstract syntax tree for Haskell OpenMath types:

```
data OMType = List [OMType]  | Rational [OMType] | Matrix [OMType]
            | MatrixRow [OMType] | Mul FiniteField Int
            | Power FiniteField Int | Num Integer
            | Polynomial String
            | ...
data FiniteField = PrimEl Int
```

For these polynomials as strings, we implement their addition and multiplication via calls to the corresponding procedures `scscp_WS_ProdPoly` and `scscp_WS_SumPoly` available at the GAP SCSCP server. These calls use the `call2` wrapper function, which implicitly applies the CASH functions `toOM`/`fromOM` to convert data to/from values of type `OMType`.

```
instance  Num (OMType) where
  (*) p1@(Polynomial _) p2@(Polynomial _) = call2 scscp_WS_ProdPoly p1 p2
  (*) ...
  (+) p1@(Polynomial _) p2@(Polynomial _) = call2 scscp_WS_SumPoly p1 p2
  (+) ...
```

We can finally write Haskell code calling GAP to factorise polynomials. We create the class `Factorisable` containing a single function `factors` returning a list of all factors of its argument, and define an instance of this class for polynomials that invokes the corresponding GAP procedure:

```
class Factorisable a where
  factors :: a -> [a]

instance Factorisable (OMType) where
  factors = call1 scscp_WS_Factors
```

5.2. A Linear System Solver using Multiple Homomorphic Images

We now show how CASH can connect to GAP to run more serious computer algebra computations. Given a linear system of equations over arbitrary precision integers, represented as a matrix $A \in \mathbb{Z}^{n \times n}$ and a vector $b \in \mathbb{Z}^n, n \in \mathbb{N}$, we want to find a vector $x \in \mathbb{Z}^n$ such that $Ax = b$, if such x exists. A potential problem in solving a system of linear equations such as this is that the values in the matrix tend to become very large, meaning that even simple arithmetic operations can become expensive to compute. A well-known common way to avoid this is to generate new matrices modulo each prime from a fixed list of prime numbers, find the solutions to these smaller problems, and finally combine these solutions using the Chinese Remainder algorithm (CRA) (Knuth, 1997, Ch.4, pp.286–291).

A generalisation of this technique became known as the *Multiple Homomorphic Images* approach (Lauer, 1982), consisting of the following three stages (the names in brackets refer to the Haskell skeleton below):

(1) map the input data into several homomorphic images (`fwd`);
(2) compute the solution in each of these images (`sol`); and
(3) combine the results of all images to a result in the original domain (`comb`).

In this case, the original domain is the set of (arbitrary precision) integers $\mathbb{Z}$, and the homomorphic images are integers modulo p, denoted by $\mathbb{Z}_p$, with p being a prime number. If a set of suitable primes is chosen, then cheap fixed-precision arithmetic can be used for each of the homomorphic images. It is only necessary to use expensive arbitrary precision

arithmetic in the combination phase, when applying the CRA in Step (3). Thus, this is already a very efficient sequential algorithm. It is straightforward to define a skeleton in Haskell to compute multiple homomorphic images:

```
multHomImg :: (Integer   -> OMType -> OMType) ->
              (Integer   -> OMType -> OMType) ->
              ([Integer] -> OMType -> OMType) ->
               [Integer] -> OMType -> OMType

multHomImg fwd sol comb ps x = res
 where xList   = zipWith fwd ps (repeat x)
       resList = zipWith sol ps xList
       res     = comb ps (toOMMatrix resList)
```

Each of the local definitions, `xList`, `resList` and `res` are the result of one of the three steps above.

The Linear Solver: We can use this skeleton to define a linear system solver:

```
linearSolver :: [ Integer ] -> [ [Integer] ] -> [ Integer ] -> OMType

linearSolver ps ms vs
  = multHomImg (call2 scscp_WS_Mod) (call2 scscp_WS_Sol)
               (call2 scscp_WS_CRA) ps
               (toOMList ((toMatrix ms):[toOMList (map toNum vs)]))
```

The functions `scscp_WS_Mod`, `scscp_WS_Sol` and `scscp_WS_CRA` are calls to the appropriate procedures available at the GAP SCSCP server: `scscp_WS_Mod` reduces the matrix and the vector modulo the given prime number. `scscp_WS_Sol` computes the solution for a matrix and vector modulo the prime number p in the field of p elements. Finally, `scscp_WS_CRA` combines the resulting vectors using GAP's built-in Chinese Remainder algorithm.

As an example, we will generate a random 1000x1000 matrix and a solution vector of length 1000, compute `m*v` to get `b` (note overloading of `(*)` so that the matrix-vector multiplication is actually performed in GAP), and then test the solution in CASH:

```
*Cash> let m = generateMatrix(1000)
*Cash> let v = generateVector(1000)
*Cash> let b = m * v
*Cash> (linearSolver [1009, 10007, 100003, 1000003] m b) == v
True
```

6. Infrastructure for parallel computations

In contrast to notations for numerical computations, which have an emphasis on floating point arithmetic, monolithic arrays, and programmer-controlled memory allocation, symbolic computing has an emphasis on functional notations, greater interactivity, very high-level programming abstractions, complex data structures, automatic memory management, etc. With this different evolutionary path, it is not surprising that symbolic computation has parallelisation requirements that differ significantly from those for traditional *numerical* high-performance computing. In particular, parallel symbolic computations are often highly irregular, need to exploit more complex data structures than

their numerical counterparts, and exhibit sophisticated computational patterns that are only just being identified.

We have developed a number of tools that exploit the capabilities of SCSCP for marshalling/unmarshalling symbolic data as part of a parallel computation, outlined below: SPSD (a middleware written in Java using the SCSCP API); the master-worker skeleton implemented directly in GAP; and a general programmable framework for parallelism, SymGrid-Par. These provide increasing levels of capability and scalability.

6.1. WUPSI/SPSD

The Java framework outlined in Section 3.3 has been used to construct WUPSI, an integrating software component that is a universal Popcorn SCSCP Interface providing several different technologies for interacting with SCSCP clients and servers. One of these is the Simple Parallel SCSCP Dispatcher (SPSD), which allows very simple patterns like parallel `map` or `zip` to be used on different SCSCP servers simultaneously. The parallelization functionality is offered as an SCSCP service itself, so it can be invoked not only from the WUPSI command line, but also by any other SCSCP client. Since WUPSI and all parts of it are open source and freely available, they can be exploited to build whatever infrastructure seems necessary for a specific use case.

6.2. GAP Master-Worker Skeleton

Using the SCSCP package for GAP, it is possible to send requests to multiple services to execute them in parallel (or to wait until the fastest result is available), and extend the basic functionality to implement more complex scenarios. One example is the master-worker skeleton, included in the SCSCP package and implemented purely in GAP. The client (i.e. master, which orchestrates the computation) works in any system that is able to run GAP, and it may even orchestrate both GAP-based and other SCSCP servers, exploiting SCSCP mechanisms such as transient content dictionaries to define OpenMath symbols for a particular operation that exists on a specific SCSCP server, and remote objects to keep references to objects that may be supported only on the other CAS. It is quite robust, especially for stateless services: if a server (i.e. worker) is lost, it will resubmit the request to another available server. Furthermore, it allows new workers (from a previously declared pool of potential workers) to be added during the computation. It has flexible configuration options and produces parallel trace files that can be visualised in EdenTV (Berthold and Loogen, 2007). The master-worker skeleton shows almost linear speedup (e.g. 7.5 on 8 cores) on irregular applications with low task granularity and no nested parallelism. The SCSCP package manual (Konovalov and Linton, 2010b) contains further details and examples. See also (Eick and Konovalov, 2011; Konovalov and Linton, 2010a) for two examples of using the package to deal with concrete research problems.

6.3. Prototyping Parallelism in CASH

In this section we will discuss how to easily use parallelism on the CASH command-line, using the primitives of Glasgow Parallel Haskell (GpH) (Trinder et al., 1998) and provided by GHC's `parallel-1.1.0.1` package. In contrast to the distributed-memory implementation of Eden (Loogen et al., 2005), this uses a version of the runtime-system that is optimised for shared-memory parallelism. It therefore represents a model of small-scale parallelism, which is convenient for the use in an interactive shell running on a

multi-core machine. We envision this model to be used for prototyping parallel code, or for asynchronously launching parallel computations, benefitting from the implicit synchronisation on shared variables on the Haskell side. For more complex parallel, symbolic applications, we develop domain-specific skeletons in the subsequent sections.

As a first example we want to demonstrate the following property of resultants of univariate polynomials: $Res(pr, q) = Res(p, q)Res(r, q)$. We first generate three random (univariate) polynomials, `p`, `q` and `r`, each of degree 17. Assuming the resultant computation to be expensive, we want to perform all three calls to it in parallel. We can easily achieve this, by using the binary `par` combinator in GpH. It launches a parallel computation for its first argument [1], and returns the value of the second argument as a result, thus achieving parallel composition. Analogously, the `pseq` combinator evaluates both arguments sequentially. Synchronisation between the parallel computations via the shared variables `z1`, `z2` and `z3` is handled automatically by the underlying runtime-system. With these primitives, and using a function `resultantPoly` which directly calls an SCSCP service for performing the actual resultant computation, we can do the above test, using three parallel computations as follows:

```
*Cash> s'init 26133
*Cash> let p = mkRandPoly 17 ; q = mkRandPoly 17 ; r = mkRandPoly 17
*Cash> let pr=p*r
*Cash> let z1 = resultantPoly pr q
*Cash> let z2 = resultantPoly p q
*Cash> let z3 = resultantPoly r q
*Cash> z1 `par` z2 `par` z3 `pseq` z1==z2*z3
True
```

Returning to our earlier example, we parallelise the `myGcd` function, by simply attaching a strategy to the top-level let expression, adding only one line of code to the sequential version.

```
parGcd :: (Num a, Factorisable a, NFData a) => a -> a -> a
parGcd x y = let
               xs = factors x
               ys = factors y
               zs = xs `bagInter` ys
             in
             if null zs then fromInteger 1 else foldl1 (*) zs
             `using` \ res -> rnf xs `par` rnf ys `pseq` rnf res
```

This strategy, in the `using` clause, specifies that the lists `xs` and `ys`, returning the factors of the inputs `x` and `y`, should be evaluated in parallel. In these simple examples, a direct use of the `par` and `pseq` primitives is sufficient. For more complex parallelism, evaluation strategies can specify parallelism, evaluation order and evaluation degree. The latter two aspects are deliberately undefined in Haskell. When defining a concrete parallel execution, however, it often becomes desirable to be explicit about them. For example, the `rnf` function above specifies a complete evaluation of a data type to normal form.

[1] In fact, in GpH all parallelism is optional, and might be ignored by the runtime system in the case of massive amounts of parallelism. On an idle machine, however, all parallelism will be realised.

We can now test our parallel function as follows:

```
*Cash> parGcd (9827329124::Integer) (83428881::Integer)
1
*Cash> parGcd (17*9827329124::Integer) (17*83428881::Integer)
17
```

6.4. SymGrid-Par

SymGrid (Hammond et al., 2007) provides a new framework for symbolic computations on *computational Grids*: distributed parallel systems built from geographically-dispersed parallel clusters of possibly heterogeneous machines. It builds on and extends standard Globus toolkit capabilities (Foster, 2005), offering support for discovering and accessing Web and Grid-based symbolic computing services (SymGrid-Services) (Cârstea et al., 2007) and for orchestrating symbolic components into Grid-enabled applications (SymGrid-Par) (Al Zain et al., 2007). Both components build on SCSCP in an essential way. This section focuses on SymGrid-Par, which aims to orchestrate multiple sequential symbolic computing engines into a single coherent parallel system.

Implementation details: SymGrid-Par (Figure 5) provides high-level parallel coordination of symbolic components into a single application. Each component executes within an instance of a Grid-enabled engine, which can be geographically distributed to form a wide-area computational grid, built as a loosely-coupled collection of Grid-enabled clusters, with components linked using SCSCP. SymGrid-Par has been designed to achieve a high degree of flexibility in constructing a platform for high-performance, distributed symbolic computation, including multiple CAS. It has three main components:

- **The Client**. The end user works in his/her own familiar programming environment, so avoiding needing to learn a new CAS, or a new language to exploit parallelism. We can imagine here the client being, for example, CASH, connecting through the Haskell middleware coordination layer to the computer algebra systems. The coordination layer is completely hidden from the CASH end user, and they work exactly as they would dealing directly with computer algebra systems. For example, the CASH client could send an SCSCP request to the coordination server to use the multiple homomorphic images skeleton as described in Section 5.2.
- **The Coordination Server**. This middleware provides parallelised services and parallel skeletons to the client. The client may invoke these skeletons as standard higher-order functions. The Coordination Server then delegates work (usually calls to expensive computational algebra routines) to the Computation Server, which is another SCSCP-compliant computer algebra system. Currently the Coordination Server is implemented in Haskell, allowing the user to exploit polymorphism, purity and higher-order functions for effective implementation of high-performance parallelism. In this example, the Coordination Server executes the Eden multiple homomorphic images skeleton as described in Section 7.1, creating GAP instances as separate worker processes.
- **The Computation Server**. This component is typically a dedicated computer algebra system, e.g. GAP. Each server handles the requests that are sent to it, sending the results back to the coordination layer for processing. Finally, the coordination layer returns the result to the client.

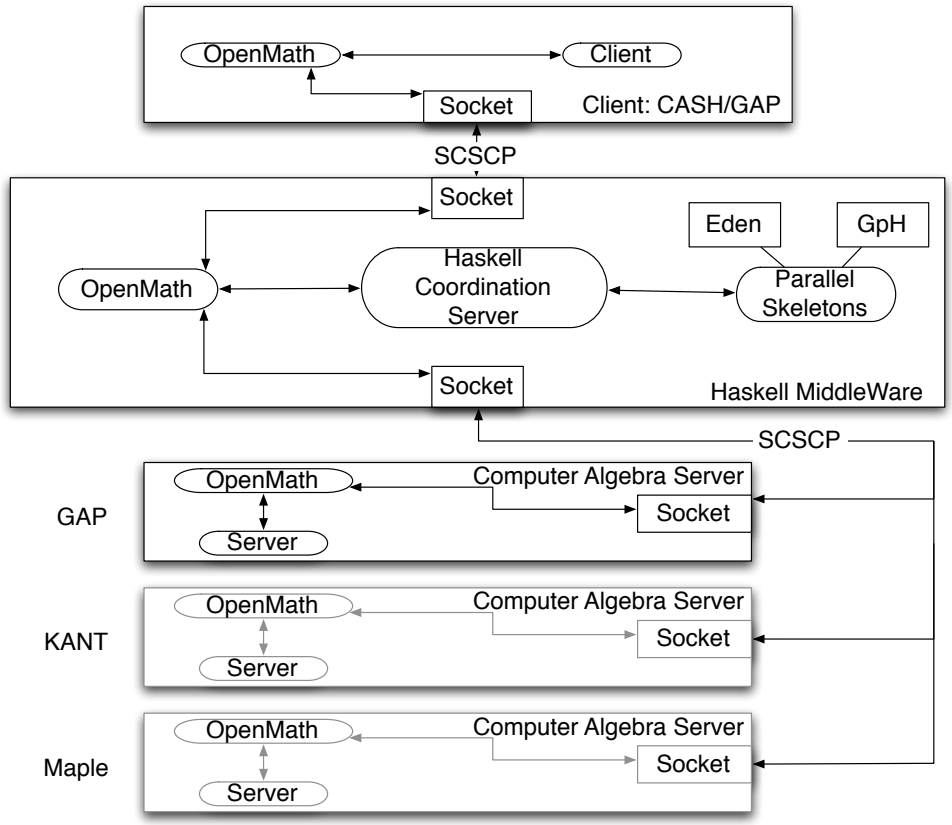

Fig. 5. **SymGrid-Par** Design Overview

7. Domain-specific parallel skeletons for symbolic computation

One of the advantages of using a modern functional language for coordinating computation is the ease with which parallel patterns of computation, so-called *skeletons* (Cole, 1989), can be constructed. In particular, the SymGrid-Par infrastructure provides parallel variants of common higher order functions, such as `parMap`, `parZipWith`, `parFold` and `parMapFold`, implemented in the Eden parallel dialect of Haskell (Loogen et al., 2005). By using these skeletons with concrete, SCSCP-based services the end user can easily create parallel applications without having to be an expert in parallel programming.

In this section we discuss two domain-specific skeletons for symbolic computation in more detail. They encode parallel implementations of frequently used computer algebra functions, and provide them as parallel SCSCP services in the SymGrid-Par infrastructure as discussed in Section 6.4. Making a Haskell-side parallel skeleton directly accessible to the CA user in this way is one major advantage of our design, since most computer algebra systems have limited, if any, support for parallelism.

7.1. Parallelising multiple homomorphic images skeleton

In this section we use Eden to parallelise the multiple homomorphic images skeleton from Section 5.2. Eden provides a model of large-scale distributed computation and

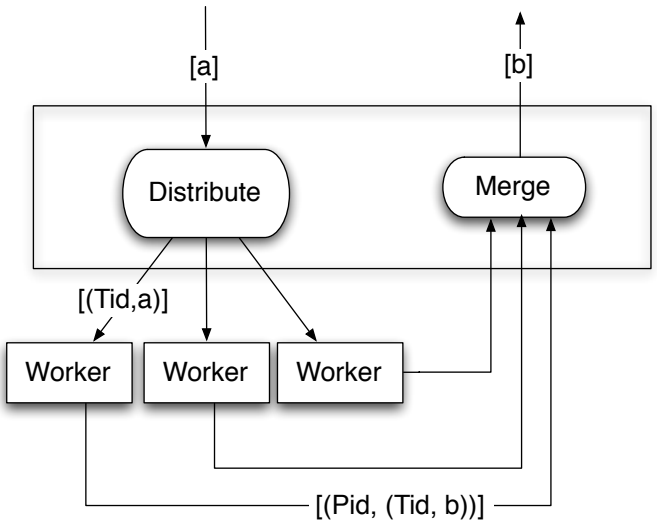

Fig. 6. The workpool skeleton

primarily targets parallel clusters. Using Eden, we can simply transform the original skeleton, which called the sequential `zipWith` function, into an equivalent parallel version that uses `parZipWith` instead.

```
multHomImg fwd sol comb ps x = res where
  xList  = zipWith fwd ps (repeat x)
  resL   = parZipWith sol ps xList
  res    = comb ps (toOMMatrix resL)
```

A parallel zipWith skeleton: The `parZipWith` skeleton uses a *workpool* approach (Klusik et al., 2001), as shown in Figure 6. Workpools are commonly used where tasks have varying granularities, to dynamically balance the allocation of tasks to processors.

```
parZipWith f l1 l2 = newTasks where
  (newReqs, newTasks) = ...

  workerProcs = [process (zip [pe,pe..] . (worker f)) | pe <- [1..noPe]]

  worker f [] = []
  worker f ((v1, v2) : ts) = (f v1 v2) : worker f ts
```

The `parZipWith` skeleton creates one `worker` process for each available processor (PE), where each worker process applies `f` to a pair `(v1, v2)`. Each of these processes is executed in parallel using the Eden `process` construct. The function `workerProcs` defines this set of worker processes, pairing each result with the id of the PE that produced it. This is used by the skeleton to determine which PEs have surplus capacity.

```
(newReqs, newTasks) = (unzip . merge) (zipWith ( # ) workerProcs
                        (distributeLists (l1, l2) requests))

 requests = (concat (replicate 2 [1..noPe])) ++ newReqs
```

The skeleton pairs the input tasks (the pairs of arguments to the worker function `f`) with a list of *requests*. The `requests` value is a list of process ids that is used to map tasks to processes. Each task is paired with a process id using the `distributeLists` function and these ids are then used to assign the task to the corresponding Eden process. The results of executing the tasks on the processes are merged using Eden's non-deterministic `merge` operation, and then unzipped to give the lists of new requests, `newReq`, and new tasks, `newTasks`. The `distributeLists` function simply accumulates the tasks for each PE in an ordered list so that these can be passed on to the correct worker process.

```
distributeLists tasks reqs = [taskList reqs tasks n | n <- [1..noPe]]
  where
    taskList (r:rs) (v1:vs1, v2:vs2) pe
        | pe == r   =  (v1, v2) : (taskList rs (vs1, vs2) pe)
        | otherwise =  taskList rs (vs1, vs2) pe
    taskList _ _ _ = []
```

Note that the GAP functions `scscp_WS_Sol` and `scscp_WS_CRA` used in 5.2 must be modified for this example to take into account not only solution vectors but also primes for which they were obtained.

Performance: In order to demonstrate the effectiveness of the parallel performance of the skeleton, we tested it on several sample matrices of 200 x 200 17-digit integers. Experiments show a 1.795 speedup on 2 cores over the original sequential version of the skeleton discussed in Section 5.2. Clearly, these early results show that the skeleton can be used to increase the performance of the linear solver; and, more importantly, the speedup shows that parallel skeletons can be an effective way of increasing the CAS performance.

7.2. The Orbit Skeleton

Orbit computation is one of the fundamental algorithms in computer algebra. It explores the solution space first applying a set of given generators to a given initial element of the domain, and then iteratively applying same generators to previously unseen images coming from the previous iteration until no new images can be produced.

The Parallel Orbit Skeleton: An algorithm can usually be parallelised in many different ways. The most appropriate technique depends on the structure of the algorithm, the granularity of the input data, the characteristics of the underlying parallel machine executing the program, and many other factors. For the initial Orbit skeleton, we employ an approach that is suitable for irregular parallelism. *Irregular parallelism* occurs when the granularity of tasks varies significantly and/or when tasks cannot easily be derived from the structure of the program or data. This can happen, for example, when the size of the input cannot be guaranteed to be evenly distributed over the parallel architecture; or when the input is highly dynamic, for example if it changes on each iteration. In the case of the *Orbit* algorithm, irregularity will arise both from the changing set of inputs, and because generators will have widely varying computation costs, depending on the input that they are applied to. Figure 8 shows the implementation of the Orbit skeleton as a workpool.

An obvious parallelisation is to distribute the queue of tasks to the available processing elements and then to merge the results into the central queue for subsequent processing.

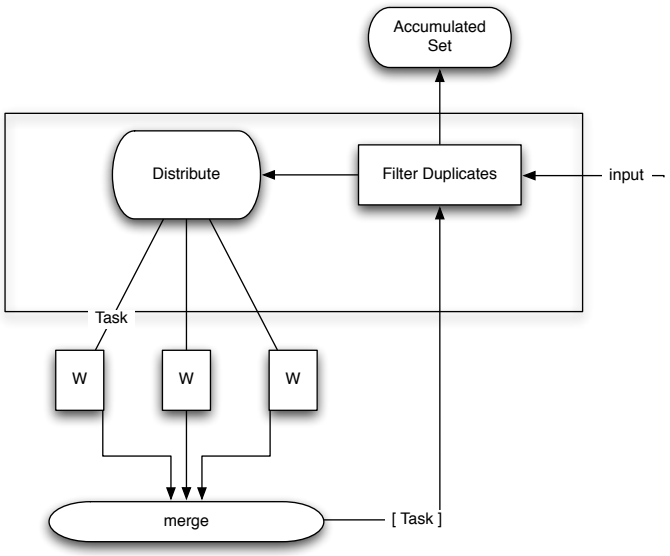

Fig. 7. A General Parallel Orbit.

Each worker process receives a task from its queue; computes the result of that task; and returns the result, which may be used to produce new tasks. The Orbit scheme is shown in Figure 7 and its skeleton implementation in Figure 8. In order to implement the Orbit skeleton, we generalise the workpool approach, adding a number of new features:

(1) the result of the workers must be accumulated and checked for duplicates;
(2) non-duplicate results are used as new tasks;
(3) the orbit terminates once there can be no more tasks.

In Figure 8 `orbitPar` is defined to take (as its arguments):

- `prefetch`: a default argument passed in by CASH to spawn the Eden processes;
- `orbitfun`: a function defining an actual single orbit computation;
- `gens`: a list of generating functions which are fed as an argument to `orbitfun`;
- `init`: an initial seed that is used to initiate the orbit computation.

Parallel computation is performed by equally distributing the tasks that are released by the filter `addNewTask` to the workers (`toWorkers`) via a distribution function `distribute`. The results of the workers are merged using the Eden `merge` operation, and passed to the next iteration of the `addNewTask` function. Rather than assigning equal numbers of tasks to each worker, `distribute` allocates the tasks to the workers that are known to be idle. Each input task is paired with the *id* of the worker that is executing it. When the task completes, the worker *id* is returned to the distribute operation, so that a new task may be allocated to the now idle worker. In order to terminate the orbit a count, `c` has been incorporated into `addNewTask`, to count the tasks released and returned by the task queue. If the task is not a duplicate, then it is released, and the count is incremented by the amount of generators, `nGen`, since `orbitFun` will always return `nGen` new tasks. If a task is a duplicate, then it is not released, but the count is decremented since there is then one fewer task in the queue. Once the count reaches zero, there are no new tasks to be considered and the orbit may terminate.

```
orbitPar prefetch orbitfun gens init = dat
  where
    nGens = length gens
    (newReqs, newTasks) = (unzip . merge) new
    new =  zipWith (#) workerProcs (toWorker dat)
    dat = (addNewTask empty (init' ++ newTasks)
                     (length init'))
    initialReqs = concat (replicate prefetch
                          [1..noPe])
    init' = take noPe (cycle init)

    addNewTask set (t:ts) c
      | not (t `member` set)
          = t : addNewTask (Data.Set.insert t set)
                           ts c'
      | c <= 1      = []
      | otherwise   = addNewTask set ts (c-1)
                           where c' = (c-1) + nGens

    workerProcs = [ process (zip [n,n..] .
                    (concat . (Data.List.map
                               (orbitfun gens))))
                  | n <- [1..noPe] ]
    toWorker tasks = distribute tasks requests
    requests = initialReqs ++ newReqs

    distribute :: [t] -> [Int] -> [[t]]
    distribute tasks reqs = [taskList reqs tasks n
                             | n<-[1..noPe]]
      where taskList (r:rs) (t:ts) pe | pe == r
                    = t:(taskList rs ts pe)
                                    | otherwise
                    = taskList rs ts pe
            taskList _      _      _  = []
```

Fig. 8. The Definition of the Parallel Orbit Skeleton with Balanced Task Distribution (Workpool).

In order to improve the efficiency of the orbit computation, we have modelled the accumulator set in terms of the standard Haskell `Data.Set` set representation. The implementation of `Set` is based on *sized balanced* binary trees (or trees of *bounded balance*) (Adams, 1993). This allows us both to check for membership and to insert a new member into the `Set` in *O(log n)* time.

Calling Orbit from GAP: Now we demonstrate the effectiveness of the Orbit in a simple example with a matrix group over a finite field. To call the Orbit skeleton from GAP client, first we define a wrapper function for the GAP client:

```
ParOrbit:=function( gen_fct, start_list)
  local res;
  res:=EvaluateBySCSCP( "CS_Orbit", [gen_fct, start_list],
                        SCSCPclientHost, SCSCPclientPort);
  return res.object;
end;
```

In order to call the `ParOrbit` wrapper from GAP, we must also define some *generators* that will be used to generate *new* elements in the Orbit. The Haskell coordination ensures that the worker generators are executed on GAP worker nodes, via the SCSCP interface. We therefore define two generating functions, `Fin1` and `Fin2`, using a global variable `mg` to compute the image of their argument:

```
mg := SL(3,3);
Fin1:=function(x) return (x ^ mg.1); end;
Fin2:=function(x) return (x ^ mg.2); end;
```

We then install these procedures as SCSCP services:

```
InstallSCSCPprocedure( "WS_Fin1", Fin1 );
InstallSCSCPprocedure( "WS_Fin2", Fin2 );
```

This allows us to call the Orbit with our wrapper function from the GAP shell passing a vector over $GF(3)$ as the seeded value:

```
gap> ParOrbit(["WS_Fin1", "WS_Fin2"], [AsList(GF(3))]);
[ [ 0*Z(3), Z(3)^0, Z(3) ], [ Z(3), Z(3)^0, 0*Z(3) ], [ Z(3), 0*Z(3), 0*Z(3) ],
[ Z(3), Z(3), 0*Z(3) ], [ Z(3), 0*Z(3), Z(3) ],[ 0*Z(3), 0*Z(3), Z(3) ],
... / some output lines omitted /...
[ 0*Z(3), 0*Z(3), Z(3)^0 ],[ 0*Z(3), Z(3), 0*Z(3) ] ]
```

Performance: As we mentioned above, initial results of the Orbit over a pure Haskell represenation (no link to GAP) recorded a superlinear speedup of over 8.295 on 8 cores (Brown and Hammond, 2010). The skeleton is still in its preliminary stages, and we intend to tune the algorithm further to allow effective parallelisation of both GAP and SCSCP examples.

8. Conclusions

We have presented a framework for combining computer algebra systems using a newly-developed remote procedure call protocol SCSCP (Symbolic Computation Software Composability Protocol). By defining common data and task interfaces for all systems, we allow complex computations to be constructed by orchestrating heterogeneous distributed components into a single symbolic application. Any system supporting SCSCP can immediately connect to all other SCSCP-compliant systems, thus avoiding the need for special cases and minimizing wasted effort. Furthermore, if some CAS changes its internal format then it only needs to update one interface, namely that to the SCSCP protocol (instead of as many interfaces as there are programs it connects to). This change can be completely transparent to the other CAS that connects to it.

We have demonstrated several examples of setting up communication between different CAS, thus exhibiting SCSCP benefits and features including its flexible design, the ability to solve problems that cannot be solved in the "home" system, and the possibility of speeding up computations by sending requests to a faster CAS. We have shown how sequential systems can be combined into heterogeneous parallel systems that can deliver good parallel performance using the SymGrid-Par framework. We have presented several domain-specific, parallel skeletons, capturing frequently used symbolic computations. The parallel implementations of these skeletons are provided as SCSCP services by the SymGrid-Par framework, and therefore easily accessible from other CASs.

SCSCP uses an OpenMath representation to encode both the transmitted data and the protocol instructions, and may be supported not only by a CAS, but by any other software as well. To achieve this, it is necessary only to support SCSCP messages according to the protocol specification, while the support of particular OpenMath constructions and objects is dictated only by the nature of the application. This support may consequently be limited to a few basic OpenMath data types and a small set of application-relevant symbols. For example, a Java applet to display the lattice of subgroups of a group may be able to draw diagrams for partially ordered sets without any support for the group-theoretical OpenMath CDs. Other possible applications may include a web or SCSCP interface to a mathematical database, or, as an extreme proof-of-concept, even a server providing access to a computer algebra system through an Internet Relay Chat bot.

Additionally, SCSCP-compliant middleware may look inside an SCSCP message, extracting all necessary technical information from its outer levels and taking the embedded OpenMath objects as a "black box".

By interfacing SymGrid-Par to the CASH front-end client, we have allowed computer algebra to be easily combined with functional programming. This brings cutting edge programming language technology to the CA field, without the laborious process of integrating this technology into existing CASs. In particular, we have demonstrated the use of overloading and higher-order functions in Haskell to simplify the development of CA code. We have used this interface to exploit extensions of Haskell, supporting both small-scale and large-scale parallelism. The examples demonstrate the prototyping of parallel code in an interactive shell using GpH (Trinder et al., 1998), as well as the development of larger parallel applications, using Eden (Loogen et al., 2005) and a skeletons approach.

A fruitful area of collaboration between the symbolic computation and functional programming communities would be the integration of the standardised mathematical objects and operations, described in this paper, into a general purpose programming language. On the Haskell side, there is active effort involved in developing such a class hierarchy in the form of a "Numeric Prelude" for Haskell (Thurston et al., 2010). However, this covers only a small fraction of the functionality provided by state-of-the-art CA systems. Another interesting direction of future work would be to use Haskell's type classes and sophisticated type system in order to provide enhanced type safety. Encoding advanced properties, such as the sizes of data structures, is a particularly active research area in the Haskell community and would be of immediate benefit to the CA community.

We have already seen examples of SCSCP-compliant software that were created outside the SCIEnce project and we hope that we will have more of them in the future. We anticipate that existing and emerging SCSCP APIs will be useful here as templates for new SCSCP implementations. In conclusion, SCSCP is a powerful and flexible framework for combining CAS, and we encourage developers to cooperate with us in adding SCSCP support to their software.

Acknowledgements

We would like to acknowledge the fruitful collaborative work of all the partners and CAS developers who have been involved in the SCIEnce project. In particular we would like to thank Abyd Al Zain and Jost Berthold, who were heavily involved in the development of SymGrid-Par.

References

Adams, A., Dunstan, M., Gottliebsen, H., Kelsey, T., Martin, U., Owre, S., 2001. Computer Algebra meets Automated Theorem Proving: Integrating Maple and PVS. In: Proc. TPHOLs 2001: Intl. Conf. on Theorem Proving in Higher Order Logics, Springer LNCS 2152. pp. 27–42.

Adams, S., 1993. Efficient Sets - a Balancing Act. Journal Functional Programming 3 (4), 553–561.

Al Zain, A., Hammond, K., Trinder, P., Linton, S., Loidl, H.-W., Costantini, M., 2007. SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In: Proc. ICCS '07: 7th Intl. Conf. on Computational Science, Springer LNCS 4488. pp. 617–624.

Berthold, J., Loogen, R., 2007. Visualizing Parallel Functional Program Runs: Case Studies with the eden trace viewer. In: Proc. PARCO 2007: Intl. Conf. on Parallel Computing: Architectures, Algorithms and Applications. Vol. 15 of Advances in Parallel Computing. IOS Press, pp. 121–128.

Besche, H., Eick, B., 1999. Construction of Finite Groups. J. Symbolic Comput. 27 (4), 387–404.

Besche, H., Eick, B., O'Brien, E., 2008. The Small Groups Library. `http://www-public.tu-bs.de:8080/~beick/soft/small/small.html`.

Brown, C., Hammond, K., 2010. Ever-Decreasing Circles: a Skeleton for Parallel Orbit Calculations in Eden. In: TFP10 — Symposium on Trends in Functional Programming Draft Proceedings. Oklahoma.

Brown, C., Loidl, H.-W., Berthold, J., Sep. 2010. Improving your CASH flow: The Computer Algebra SHell. In: IFL'10: Symposium on Implementation and Application of Functional Languages. Springer, Alphen aan den Rijn, the Netherlands, to appear.

Buswell, S., Caprotti, O., Carlisle, D., Dewar, M., Gaëtano, M., Kohlhase, M. (Eds.), 2004. The OpenMath Standard, Version 2.0. `http://www.openmath.org/`.

Cannon, J., Bosma, W., 2008. Handbook of Magma Functions, Edition 2.15. School of Mathematics and Statistics, University of Sydney `http://magma.maths.usyd.edu.au`.

Caprotti, O., Cohen, A., September 1999. Connecting Proof Checkers and Computer Algebra using OpenMath. In: The 12th International Conference on Theorem Proving in Higher Order Logic. Nice, France, pp. 109–112.

Caprotti, O., Cohen, A., Riem, M., 2000. Java Phrasebooks for Computer Algebra and Automated Deduction. In: SIGSAM Bulletin, 2000. Special Issue on OpenMath. pp. 33–37.

Cârstea, A., Frîncu, M., Macariu, G., Petcu, D., Hammond, K., 2007. Generic Access to Web and Grid-based Symbolic Computing Services: the SymGrid-Services Framework. In: Proc. ISPDC 07: Intl. Symp. on Parallel and Distributed Computing, Castle Hagenberg, Austria, IEEE Press. pp. 143–150.

Cole, M., 1989. Algorithmic Skeletons: Structure Management of Parallel Computations. MIT Press.

Costantini, M., Konovalov, A., Solomon, A., 2010. OpenMath – OpenMath functionality in GAP, Version 10.1. GAP package. `http://www.cs.st-andrews.ac.uk/~alexk/openmath.htm`.

Daberkow, M., Fieker, C., Klüners, J., Pohst, M., Roegner, K., Schörnig, M., Wildanger, K., 1997. KANT V4. J. Symbolic Comput. 24 (3-4), 267–283, Computational Algebra and Number Theory, 1993.

Eick, B., Konovalov, A., 2011. The Modular Isomorphism Problem for the Groups of Order 512. In: Groups St. Andrews 2009. London Math. Soc. Lecture Note Ser. Accepted.

Foster, I., 2005. Globus Toolkit Version 4: Software for Service-Oriented Systems. In: Jin, H., Reed, D., Jiang, W. (Eds.), Network and Parallel Computing. Vol. 3779 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 2–13.

Freundt, S., Horn, P., Konovalov, A., Lesseni, S., Linton, S., Roozemond, D., 2009a. OpenMath Content Dictionaries `scscp1` and `scscp2`. `http://www.win.tue.nl/SCIEnce/cds/scscp1.html`, `http://www.win.tue.nl/SCIEnce/cds/scscp2.html`.

Freundt, S., Horn, P., Konovalov, A., Lesseni, S., Linton, S., Roozemond, D., 2009b. OpenMath in SCIEnce: Evolving of Symbolic Computation Interaction, in James H. Davenport, ed.: 22nd OpenMath Workshop; July 2009.

Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D., 2008. Symbolic computation software composability. In: AISC/MKM/Calculemus, Springer LNCS 5144. pp. 285–295.

Freundt, S., Horn, P., Konovalov, A., Linton, S., Roozemond, D., 2009c. Symbolic Computation Software Composability Protocol (SCSCP) specification, Version 1.3. `http://www.symbolic-computation.org/scscp`.

Freundt, S., Lesseni, S., 2010. autokash – KANT 4 SCSCP Package, Version 3.1. `http://www.math.tu-berlin.de/~kant/kantscscp.html`.

GAP Group, 2008. GAP – Groups, Algorithms, and Programming, Version 4.4.12. `http://www.gap-system.org`.

Gastineau, M., 2009. SCSCP C Library – A C/C++ library for Symbolic Computation Software Composibility Protocol, Version 0.6.0. IMCCE, `http://www.imcce.fr/Equipes/ASD/trip/scscp/`.

Geck, M., Hiss, G., Lübeck, F., Malle, G., Pfeiffer, G., 1996. CHEVIE – A System for Computing and Processing Generic Character Tables for Finite Groups of Lie type, Weyl Groups and Hecke algebras. Appl. Algebra Engrg. Comm. Comput. 7, 175–210.

Gent, I., Harvey, W., Kelsey, T., Linton, S., 2003. Generic SBDD Using Computational Group Theory. In: Proc. CP 2003: Intl. Conf. on Principles and Practice of Constraint Programming, Kinsale, Ireland. pp. 333–347.

GHC Team, 2010. GHC — The Glasgow Haskell Compiler. Web page, `http://www.haskell.org/ghc/`.

Grayson, D., Stillman, M., 2010. Macaulay2, a Software System for Research in Algebraic Geometry, Version 1.4. `http://www.math.uiuc.edu/Macaulay2/`.

Hammond, K., Al Zain, A., Cooperman, G., Petcu, D., Trinder, P., August 2007. SymGrid: a Framework for Symbolic Computation on the Grid. In: Proc. EuroPar'07. LNCS. Rennes, France, pp. 457–466.

Held, J., Bautista, J., Koehi, S., 2006. From a Few Cores to Many: A Tera-Scale Computing Research Overview. Intel Corporation, `http://download.intel.com/research/platform/terascale/terascale_overview_paper.pdf`.

Horn, P., 2010. MuPAD OpenMath Package. `http://mupad.symcomp.org/`.

Horn, P., Roozemond, D., 2009. OpenMath in SCIEnce: SCSCP and POPCORN. In: Intelligent Computer Mathematics – MKM 2009. Vol. 5625 of Lecture Notes in Artificial intelligence. Springer, pp. 474–479.

Horn, P., Roozemond, D., 2010a. Java Library for SCSCP and OpenMath. `http://java.symcomp.org/`.

Horn, P., Roozemond, D., 2010b. WUPSI –Universal Popcorn SCSCP Interface. http://java.symcomp.org/wupsi.html.

Klusik, U., Loogen, R., Priebe, S., Rubio, F., 2001. Implementation Skeletons in Eden: Low-Effort Parallel Programming. In: Proc. IFL '00: 12th International Workshop on Impl. of Functional Languages. LNCS 2011 Springer-Verlag, pp. 71–88.

Knuth, D. E., 1997. The Art of Computer Programming. Vol. 2: Seminumerical Algorithms, 3rd Edition. Addison-Wesley.

Komendantsky, V., Konovalov, A., Linton, S., 2010. Interfacing Coq + SSReflect with GAP. In: Proceedings of 9th International Workshop On User Interfaces for Theorem Provers. UITP '10. Accepted.

Konovalov, A., Linton, S., 2010a. Parallel Computations in Modular Group Algebras. In: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation. PASCO '10. ACM, New York, NY, USA, pp. 141–149.

Konovalov, A., Linton, S., 2010b. SCSCP – Symbolic Computation Software Composability Protocol, Version 1.2. GAP package. http://www.cs.st-andrews.ac.uk/~alexk/scscp.htm.

Lauer, M., 1982. Computing by Homomorphic Images. In: Computer Algebra — Symbolic and Algebraic Computation. Springer-Verlag, pp. 139–168.

Linton, S., Hammond, K., Konovalov, A., Al Zain, A. D., Trinder, P., Horn, P., Roozemond, D., 2010. Easy Composition of Symbolic Computation Software: a New Lingua Franca for Symbolic Computation. In: Proceedings of the 2010 International Symposium on Symbolic and Algebraic Computation. ISSAC '10. ACM, New York, NY, USA, pp. 339–346.

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

Lübeck, F., Neunhöffer, M., 2008. GAPDoc – A Meta Package for GAP Documentation, Version 1.2. GAP package. http://www.math.rwth-aachen.de/~Frank.Luebeck/GAPDoc.

Maplesoft, 2010. Maple 14. http://www.maplesoft.com/.

Neunhöffer, M., 2010. IO – Bindings for low level C library IO, Version 3.1. GAP package. http://www-groups.mcs.st-and.ac.uk/~neunhoef/Computer/Software/Gap/io.html.

O'Brien, E., Nickel, W., Gamble, G., 2006. ANUPQ – ANU p-Quotient, Version 3.0. GAP package. http://www.math.rwth-aachen.de/~Greg.Gamble/ANUPQ/.

Peyton Jones, S., Hammond, K., December 2003. Haskell 98 Language and Libraries, the Revised Report. Cambridge University Press.

SciFace Software, 2007. MuPAD Pro 4.0.6. http://www.gap-system.org.

Soicher, L., 2004. Computing with Graphs and Groups. In: Topics in Algebraic Graph Theory. Cambridge University Press, pp. 250–266.

Stein, W., et al., 2010. Sage Mathematics Software (Version 4.6). The Sage Development Team, http://www.sagemath.org.

Thurston, D., Thielemann, H., Johansson, M., 2010. Numeric prelude (0.2). http://www.haskell.org/haskellwiki/Numeric_Prelude.

Trinder, P., Hammond, K., Loidl, H.-W., Peyton Jones, S., Jan. 1998. Algorithm + Strategy = Parallelism. J. of Functional Programming 8 (1), 23–60.