SymGrid-Par: Parallel Orchestration of Symbolic Computation Systems

The SCIEnce project
http://www.symbolic-computation.org
support@symbolic-computation.org

1. INTRODUCTION

Primary challenges for modern symbolic computation systems are the transparent access to complex, mathematical software, the exchange of data between independent systems with specialised tasks and the exploitation of modern parallel hardware. Transparent access is increasingly delivered through Grid services that standardise the access to remote software on a global scale. For the end user this provides *ease-of-use*, relieving him/her of tasks such as installing and maintaining software, often with numerous dependencies. To facilitate data exchange meta-languages have been developed and standardised. The most prominent effort in this community is the OpenMath standard, on which we build. Finally, and for us most importantly, to *exploit parallel hardware* an execution model is needed, that matches the mathematical abstractions employed in symbolic computation applications. The constructs for parallelism should be non-intrusive, thus avoiding large-scale code restructuring, and largely advisory, so that the implementation has the flexibility to adjust the dynamic execution to the widely different characteristics of the parallel hardware. We use variants of parallel Haskell [7, 9, 8] as a language with implementations that dynamically and adaptively distribute the parallelism on the parallel hardware.

The SCIEnce project addresses all of the above challenges. More specifically, the **SymGrid-Par** system focuses on exploiting parallel hardware. It provides an easy-to-use platform for parallel computation that connects several underlying computer algebra systems, communicating through a standardised protocol for symbolic computation. The platform is directly accessible from the computer algebra shell, and parallelism can be easily specified using pre-defined patterns of parallel computation.

2. DESIGN OF SYMGRID-PAR

SymGrid-Par orchestrates symbolic components into a Grid-enabled 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. Components communicate using the standardised OpenMath data-exchange protocol.

SymGrid-Par has been designed to achieve a high degree of flexibility in constructing a platform for high-performance, distributed symbolic computation. The most visible aspect of this flexibility is the possibility connect different computer algebra systems (CAS) to co-operate in the execution of a program. This requires a common data and communication protocol, which has been defined earlier in the SCIEnce project in the form of the SCSCP protocol [5] and is being reported on separately.

Efficiently exploiting modern parallel hardware is challenging because it is increasingly heterogeneous and connected to form deep hierarchies, with widely varying costs of communication between sub-networks. For example, multi-core machines may be used as high-performance compute engines, with very low communication costs and the potential to exploit fine-grained parallelism. These machines can be connected to local area networks adding one level to the hierarchy. Such Grid-enabled clusters can be connected on a global scale for massively parallel computation.

The different levels in this hierarchy require different forms of resource management to optimise efficiency. Whereas on a multi-core machine the migration of a thread to a different core is a cheap process, and worthwhile in order to optimise load balance, such communication between sub-clusters on Grid level might be prohibitively expensive. Therefore, we apply different implementations of parallel Haskell on the different levels in the hierarchy. Our overall vision is a hierarchy encompassing all of these levels (Figure 3). The level that we focus on in this report is that of a local area network, with possibly distributed and heterogeneous CAS.

Figure 1 depicts the SymGrid-Par design as a stack of layers (left) of increasing levels of abstraction. The middle stack presents an early version of SymGrid-Par [2], based on a bespoke interface between Haskell and the underlying CAS. The right stack describes the latest version of SymGrid-Par, based on established standards and supporting a distributed collection of servers. The realisation of the levels is specific to the application domain of parallel, symbolic computation and provides at the highest level a Grid-enabled interface to access it in a location transparent way.

	Bespoke interface	***SCSCP interface***
Access Layer:	Grid	Grid
Service Layer:	Grid Service	Grid Service
Application Layer:	Skeletons/Strategies	Skeletons
Coordination Layer:	parallel Haskell (GpH)	parallel Haskell (Eden)
Communication Layer:	Bespoke	SCSCP
Data Layer:	Strings	OpenMath
Connection Layer:	Pipes	Sockets

Figure 1: Layers (left) in the SymGrid-Par Design: bespoke (middle) and SCSCP-based (right)

Connection Layer. The connection layer defines the software level of connecting physically distributed machines. As a flexible, standardised way of establishing and managing connections between the SymGrid-Par middleware and the computer algebra systems, *sockets* have been used.

Data Layer. The data layer defines the data format for items to be exchanged. Here we build on the preceding standardisation process in the community and use the OpenMath standard. This is an XML based data format designed specifically for representing symbolic and mathematical objects. More specifically, we use a specialised content dictionary within this general framework, which is defined together with the SCSCP protocol.

Communication Layer. The communication layer defines a protocol of messages that are exchanged in realising communication between two components of the system. The standard developed earlier in this project, specifying this layer, is the *Symbolic Computation Software Composability Protocol, SCSCP* [5].

Coordination Layer. The coordination layer specifies in which form parallelism is specified and managed. Several approaches, with varying levels of abstraction, are possible to realise this layer. We are using parallel implementations of the functional programming language Haskell [7, 9, 8] as a high-level parallel programming model.

Application Layer. The application layer presents a high-level API to the application programmer, for realising parallel symbolic applications. It should abstract over the low level details of the orchestration on the level below and should be powerful enough to specify the parallel execution. As a domain specific instance of this layer, tailored to the characteristics of symbolic computation, we have defined an interface that is based on the concept of *algorithmic skeletons* [4]. The concrete interfaces, namely CGA and GCA between parallel system and computer algebra system, are defined in [2].

Service Layer. The service layer defines in which way an application, or service, is made available on the web, in the form of a Grid-enabled application. This level is part of the SymGrid-Services component of the overall infrastructure.

Access Layer. Finally, the access layer defines in what form the end-user accesses a concrete service. Again, this level is part of the SymGrid-Services component.

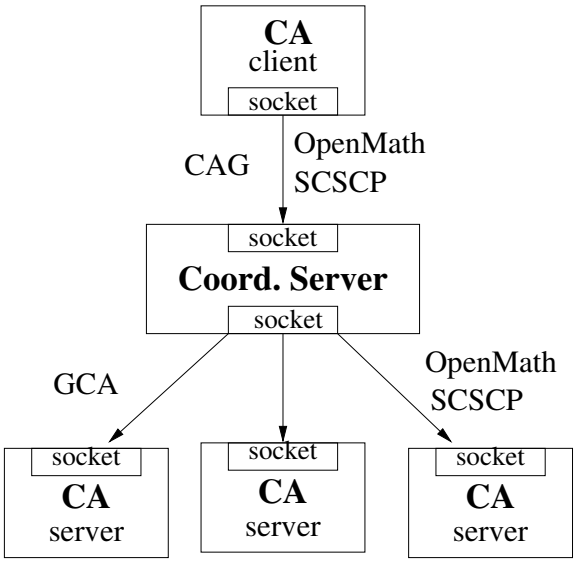

Figure 2: Current SymGrid-Par Architecture

3. EXAMPLES OF HIGH-LEVEL ORCHESTRATION

We exemplify our approach of using a high-level programming language to control the parallelism exploited by a collection of computer algebra systems by computing the sum of the Euler totient function over a list of integers. This program has been used as a simple parallel benchmark, exposing a common fold-of-map structure, in comparing the performance of several parallel functional languages. Here we focus on how to express the parallelism and how to interface with the underlying computer algebra systems that perform the bulk of the computation.

The system structure of SymGrid-Par is shown in Figure 2. Notably the end user continues to work in the shell of his/her computer algebra system, avoiding the overhead of learning a different system or language just to exploit parallelism. The SCSCP interface of the system is used to emit calls to the Coordination Server, whose role is to coordinate the parallelism in the application. It completely hides all aspects of the parallelism to the end user. Thus, a parallelised algorithm becomes indistinguishable from a sequential one, only exhibiting better performance. To additionally provide the end-user with an easy way of specifying parallelism in his/her applications, a set of patterns of symbolic computa-

tion, or skeletons can be used. Thus the end user, working for example in a GAP shell [6], uses the following call to invoke a parallel execution:

```
EvaluateBySCSCP("CS_SumEuler",
                [ 8000, 2000 ], "localhost", 26133);
```

The Coordination Server implements the parallelism by either using the primitives of the parallel Haskell extension or by using pre-defined patterns. In our example the service `CS_SumEuler` is mapped to the function `sumEuler` below. The first argument specifies the list length and the second the chunk size, i.e. the size of blocks in the input list for which parallel tasks are generated. Hence, the algorithm first determines the ranges for all blocks and then instantiates processes, using `createProcess`, for each of these ranges. This exemplifies how the Coordination Server performs small Haskell computations to organise the parallel coordination. Of course, it is also possible to perform more substantial computations in Haskell in order to avoid the overhead of performing an external call. Adjusting the granularity of the calls to the underlying CAS is one of the main activities in tuning the parallel performance of the application. In our example, the computation performed by each parallel process is specified in `sumEulerRange`. This function directly corresponds to a function in GAP. Therefore, we only need wrapper code, that transforms input and output from/to OpenMath objects, using `fromOM` and `toOM`, respectively. Then, the function `callSCSCP` is used to emit an SCSCP call to the GAP-side service `WS_SumEulerRange`. Note that the result of this wrapper function is of type `IO Int`, since SCSCP calls interact with the outside world, from Haskell's point of view. Since we know that this call is to a side-effect free function, we can immediately extract the result by using an `unsafePerformIO`, which simplifies the handling of the result list `xs`, and reduces the amount of monadic code.

```
sumEuler :: Int -> Int -> IO Int
sumEuler n c = do
 let ranges = [[i*c+1, (i+1)*c] | i <- [0..(num c n)-1]]
 let xs' = map (createProcess (process (\ns ->
            unsafePerformIO (sumEulerRange ns)))) ranges
           `using` whnfspine
 let xs :: [Int]
     xs =  map deLift xs'
 return (sum xs)

sumEulerRange :: [Int] -> IO Int
sumEulerRange  = return .
  fromOM . (callSCSCP WS_SumEulerRange) . (map toOM)
```

Finally, on the GAP side we use the following code to perform the computation over a segment of the list, where `euler` is a GAP-side implementation of the Euler totient function, and `Sum` is a built-in higher-order function, combining a map of a function with the summation of the results.

```
SumEulerRange:=function(n,m)
local result, x;
result:=Sum( [ n..m ], x -> euler(x) );
return result; end;
```

The code required to make a function available as a service is minimal, and only defines a binding of the service to a function on the CAS or Coordination Server side, e.g.

```
InstallSCSCPprocedure("WS_SumEulerRange", SumEulerRange,
                      "see sumEuler.g", 1, 2);
```

Alternatively, we can compute the sum of the Euler totient function, by using a `parMapFold` pattern, which implicitly generates parallelism for each of the calls and then applies a second function in the fold phase. Both of these functions are specified as SCSCP services provided by the underlying CAS. The Coordination Server performs SCSCP calls to invoke these functions, which are `WS_Phi` for the totient function in the map-phase and `WS_Plus` for the summation in the fold-phase. For a concrete input list we can start the parallel computation like this

```
EvaluateBySCSCP("CS_parMapFold",
                ["WS_Phi","WS_Plus", 0, [87, 88, 89]],
                "localhost", 26133);
```

This simple example demonstrates, how the user of the computer algebra system can easily express parallelism, without having to know anything about parallel programming per-se and without leaving the familiar shell. We currently provide a repertoire of common higher-order functions with built-in parallel execution, such as map, fold and zipWith [2], which together form the application layer GCA and CGA interfaces between parallel Haskell and the computer algebra systems. A skeleton for the well-known Google map-reduce pattern, which has a similar but more complex structure than the above `parMapFold`, has been developed already and will be integrated into the system shortly.

Several symbolic computation examples that have been parallelised on multi-core machines using the older bespoke SymGrid-Par design [1] are currently being ported to the new system. These examples include a program that searches for groups, whose order is not greater than a given constant, and a parallel version of the summatory Liouville function. Further examples of symbolic applications involve early versions of polynomial GCD and resultant computations, being parallelised using variations of the above skeletons. Some larger-scale applications with broad impact that we plan to implement in the SymGrid-Par infrastructure are Gröbner Bases computation, polynomial factorisation and the transformation of a non-deterministic into a deterministic finite state automaton. One common pattern in symbolic computation is an orbit computation, which explores a solution space, given some seed values and a number of generators. This pattern is used in many concrete applications and therefore represents a good basis for a domain-specific skeleton. An initial parallel version of this general orbit skeleton achieves excellent speedups on a multi-core machine and is currently being integrated into SymGrid-Par [3].

4. SUMMARY

We have presented the latest implementation of SymGrid-Par, a heterogeneous system for parallel symbolic computation, capable of coordinating several computer algebra systems and exploiting a high-level model of parallel execution

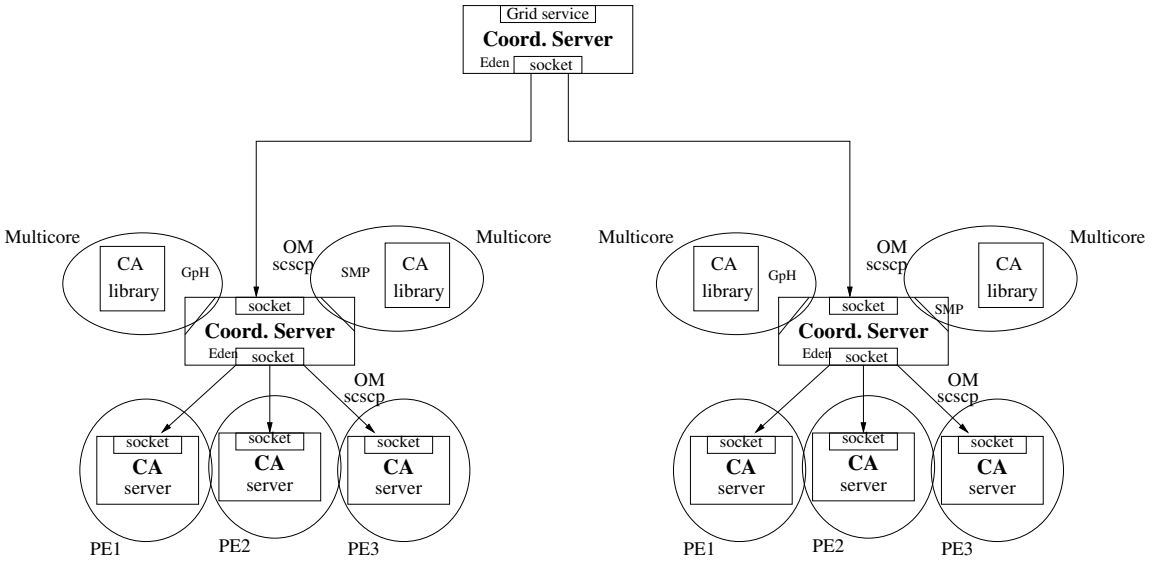

Figure 3: The Vision of Hierarchical SymGrid-Par

based on algorithmic skeletons. The novelty of SymGrid-Par lies in the following features:

- it is the first high-level parallel system for coordinating symbolic computations based on the SCSCP standard;
- it uses high-level coordination and most notably higher-order functions in the form of domain-specific skeletons to allow easy parallelisation for non-specialists in the area of parallel programming;
- it makes access to parallel orchestration directly available in the familiar environment of a computer algebra shell;
- it potentially can co-ordinate several systems, based on the SCSCP protocol, to enhance both functionality and (parallel) performance;

The current, publicly available implementation of SymGrid-Par has been used to parallelise simple, but representative, algorithms in the GAP system. Several skeletons, capturing domain-specific patterns of symbolic computation are currently under development. While we aim at large-scale, distributed parallelism, our system can already be used on novel multi-cores, albeit without optimised control of parallelism for this hardware. Our longer term vision for SymGrid-Par is a hierarchical system, as shown in Figure 3, which combines the low-overhead, bespoke version of SymGrid-Par on individual multi-core nodes with the latest, SCSCP-based version on clusters of such machines. In combination with related work in the SCIEnce project, on the SCSCP standard and on Grid services, this infrastructure presents a platform for high-performance symbolic computation that will enable the application of compute intensive programs on a much wider scale than currently possible on sequential systems.

The latest version of SymGrid-Par is on-line available at:
`http://www.symbolic-computation.org/SymGrid`

5. REFERENCES

[1] A. Al Zain and J. Berthold and K. Hammond and P.W. Trinder and G.J. Michaelson and M.K. Aswad. Low-Pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on MultiCore Architectures. In *DAMP'09 — Workshop on Declarative Aspects of Multicore Programming*. ACM Press, January 2009.

[2] A. Al Zain and K. Hammond and P. Trinder and S. Linton and H-W. Loidl and M. Costantini. SymGrid-Par: Designing a Framework for Executing Computational Algebra Systems on Computational Grids. In *ICCS'07 — Intl. Conference on Computer Science*, Beijing, China, 2007.

[3] C. Brown and K. Hammond. Ever-Decreasing Circles: a Skeleton for Parallel Orbit Calculations in Eden. In *TFP10 — Symposium on Trends in Functional Programming*, Oklahoma, May 2010.

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

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

[6] The GAP Group. The GAP Group, GAP – Groups, Algorithms, and Programming, 2008. Version 4.4.12 (`http://www.gap-system.org`).

[7] *The Haskell 98 Report*. `http://haskell.org/onlinereport/`.

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

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