ACM Communications in Computer Algebra, Vol. 44, No. 4, December 2010

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 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. 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 can adjust the dynamic execution to the widely different characteristics of the parallel hardware.

The design of SymGrid-Par addresses all of the above challenges and focuses on exploiting parallel hardware. 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. 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 to connect different computer algebra systems (CAS) to co-operate in the execution of a program. To efficiently exploit modern parallel hardware, which is increasingly heterogeneous and hierarchical, we apply different implementations of parallel Haskell, performing dynamic and adaptive management of parallelism.

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, based on a bespoke interface between Haskell and the underlying CAS [1]. The right stack describes the latest, standards-compliant version of SymGrid-Par, supporting a distributed collection of servers. Most notably we use established standards, such as sockets, where possible, and build on newly developed standards, such as OpenMath and the Symbolic Computation Software Composability Protocol (SCSCP) [3], which is reported on separately in this issue. For coordination we are using parallel implementations of the functional programming language Haskell [5, 6] as a high-level parallel programming model. Access to the system at-large is provided through the separate SymGrid-Services interface.

2 Examples of high-level orchestration

We exemplify our high-level parallel programming approach by computing the sum of the Euler totient function over a list of integers, exposing a common fold-of-map structure. 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.

	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: SymGrid-Par Layers (left): bespoke (middle) and SCSCP-based (right)

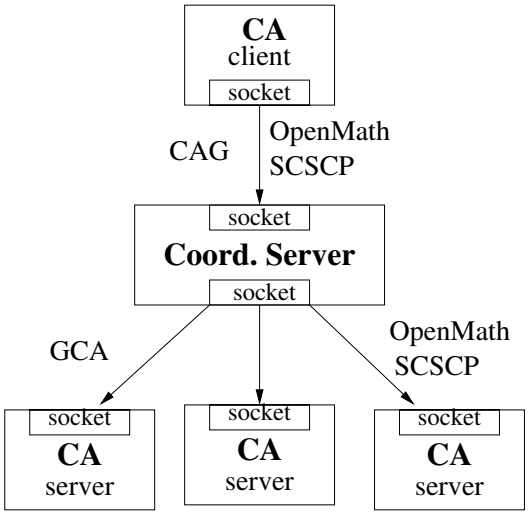

Figure 2: Current SymGrid-Par Architecture

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 computation, or skeletons [2] can be used. Thus the end user, working for example in a GAP shell [4], 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. This parameter provides a handle to tune the granularity of the CAS calls, improving the parallel performance. 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. 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 rgs = [[i*c+1, (i+1)*c] | i <- [0..(num c n)-1]]
 let xs' = map (createProcess (process (\ns ->unsafePerformIO (sumEulerRange ns)))) rgs
           `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 `xs` we can start the parallel computation like this

```
EvaluateBySCSCP("CS_parMapFold", ["WS_Phi","WS_Plus", 0, xs], "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, as well as a domain specific orbit skeleton, which is currently being integrated into SymGrid-Par. More symbolic computation examples that have been parallelised on multi-core machines using the older bespoke SymGrid-Par design are currently being ported to the new system. These include a parallel version of the summatory Liouville function, polynomial GCD and resultant computations, and a Gröbner Bases algorithm.

3 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 based on algorithmic skeletons. The novelty of SymGrid-Par lies in the following features: (i) it is the first high-level parallel system for coordinating symbolic computations based on the SCSCP

standard; (ii) 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; (iii) it makes access to parallel orchestration directly available in the familiar environment of a computer algebra shell; (iv) 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. The latest version of SymGrid-Par, with links to background on (parallel) Haskell, skeletons and GAP, is on-line available at: `http://www.symbolic-computation.org/SymGrid`

References

[1] 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, 2007.

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

[3] 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.

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

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

[6] 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.