GRID superscalar and SAGA: forming a high-level and platform-independent Grid programming environment

Raül Sirvent[1], Andre Merzky[2], Rosa M. Badia[1], and Thilo Kielmann[2]

[1] Barcelona Supercomputing Center and UPC, Barcelona, Spain
{rosa.m.badia|raul.sirvent}@bsc.es
[2] Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands
{merzky|kielmann}@cs.vu.nl

Abstract. The Simple API for Grid Applications (SAGA), as currently standardized within GGF, aims to provide a simple yet powerful Grid API; its implementations shielding applications from the intricacies of existing and future Grid middleware. The GRID superscalar is a programming environment in which Grid-unaware task flow applications can be written, for execution on Grids. As such, GRID superscalar can be seen as a client application to SAGA. In this paper, we discuss how SAGA can help implementing GRID superscalar's runtime system, together forming a high-level and platform-independent programming environment for the Grid.

1 Introduction

Core Grid technologies are rapidly maturing, but there remains a shortage of real Grid applications. One important reason is the lack of a simple and high-level application programming toolkit, bridging the gap between existing Grid middleware and application-level needs.

Experience shows that both Grid-aware and Grid-unaware aplications exist for their respective purposes, however having different API requirements. Typical Grid-aware applications can be taylored to the Grid, so achieving a specific use of the services provided, for example, explicitly use remote resources like data repositories. The main drawback of Grid-aware applications is the bigger effort required from the user to develop the application. Grid-unaware applications are easier from users point of view, because they don't have to care about the details in the underlying Grid infrastructure. They simply need to complete their task, irrespective of the machinery involved. The drawback in this case is that you cannot use more advanced services offered in the Grid, because the Grid-unaware infrastructure uses the services by you. From this we can see that the two groups of users, Grid-aware and Grid-unaware, are completely opposite. The first want to extract all the functionality available to the Grid, while the last want to run their work in the Grid with a minimum effort.

For both categories of applications, targeted programming environments exist. In this paper, we anticipate an integrated approach, as sketched in Fig. 1.

In particular, we investigate the integration of GRID superscalar [4] and of SAGA [10], aiming at merging, kind of, “the best of both worlds.” An important part of this exercise is to implement GRID superscalar’s runtime system using the SAGA interface only, allowing for a both high-level and platform-independent Grid programming environment.

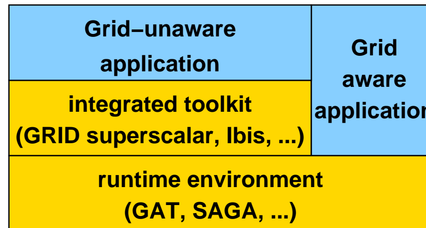

Fig. 1. Integrated runtime platform

The remainder of this paper is organized as follows. Sections 2 and 3 will sketch the SAGA API and GRID superscalar, respectively. In Section 4, our anticipated integration of both will be presented. Section 5 will briefly sketch related work, and Section 6 concludes.

2 The Simple API for Grid Applications (SAGA)

The SAGA API specification as it is currently being defined by GGF focuses on four components widely cited in Grid computing use cases. These components and their respective, most important method calls are:

1. **Files:** management and access to files on remote storage resources:
   - `directory.{cd, list, copy, link, move, delete, exists, create, open, open_dir, ...}`
   - `file.{read, write, seek}`
2. **Logical Files:** management and access to replica systems for large distributed data Grids:
   - `logical_directory.{cd, list, copy, link, move, delete, exists, create, open, open_dir, ...}`
   - `logical_file.{add_location, remove_location, list_locations, replicate}`
3. **Jobs:** starting and controlling jobs running on remote Grid resources:
   - `job_description.{set_attribute, get_attribute, ...}`
   - `job_server.{list_jobs, run_job, submit_job}`
   - `job.{suspend, resume, hold, release, checkpoint, migrate, ...}`

   - job.{get_id, get_status, get_exit_status, ...}

4. **Streams:** exchange of application specific data via a secured, high throughput streaming channel:
   - stream_server.{wait_for_connection}
   - stream.{connect, read, write, status, wait}
   - multiplexer.{watch, unwatch, wait}

This list of components and methods is very concise on purpose, but it allows the implementation of a large number of simple Grid use cases. As the SAGA API is abstracting and simplifying the paradigms provided by the various Grid middleware incarnations, SAGA implementation can provide a very simple and stable environment, protecting the user from the evolution of Grid middleware.

Along with the mentioned API components, the API draft also defines a common look-and-feel for the components, and for future extensions. Additionally, the so called API core encompasses:

- session handling,
- encapsulation of security information,
- error handling,
- a generic task model.

In particular the task model is important to many Grid applications, as it allows both the asynchronous execution of potentially slow, remote operations, and the ability to react on status changes for pending remote operations. Fig. 2 shows an example of an asynchrounous file read operation using the task model. The SAGA API will likely be extended in the future, with additional paradigms such as GridRPC, access to information services, monitoring and steering.

3 The GRID superscalar programming environment

In contrast to SAGA, GRID superscalar [4] tries to hide the Grid from the user, in such a way that writing an application for a computational Grid may be as easy as writing a sequential application. From the source code provided by the user, and taking into account the functions in the code selected to be Grid-enabled, the run-time system generates a directed, acyclic graph (DAG) of these functions where the dependencies are defined via data files. From that graph, the run time system executes all those functions for which data dependencies become resolved, therefore achieving automatic functional parallelism for the original sequential program. The GRID superscalar framework is mainly composed of the programming interface, the deployment center and the run-time system. It acts as an assembly language for the Grid, as it allows to use different programming languages on top of it.

The programming interface offers a very small set of primitives. These primitives must be used in the main program of the application for different purposes.

Fig. 2. Code Snippet: *asynchronous file reading with the SAGA task model*

```
{
  // synchronous operations
  saga::file   f (url);
  string out = f.read (100);
  std::cout << out << std::endl;
}
{
  // asynchronous operations
  saga::file f (url);
  saga::file::task_factory ftf = f.get_task_factory ();

  string out;
  saga::task t = ftf.read (100, out);

  t.run ();

  while ( saga::task::Done != t.get_status () )
    { sleep (1); }

  std::cout << out << std::endl;
}
```

GS_On() and GS_Off() primitives are provided for initialization and finalization of the run-time. GS_Open(), GS_Close(), GS_FOpen() and GS_FClose() for handling files. The GS_Barrier() function has been defined to allow the programmers to wait till all Grid tasks finish. Also the GS_Speculative_End() function allows an easy way to implement parameter studies. In order to specify the functions to be executed in the Grid, an IDL file must be used. For each of these functions, the type and direction of the parameters must be specified (where direction means if it is an input, output or input/output parameter.)

The deployment center is a Java-based Graphical User Interface. It implements the Grid resource and application configuration, an early failure detection, the source code file transfer to the remote machines, the generation of additional source files required for the master and the worker parts (using the gsstubgen tool), the compilation of the main program in the localhost and worker programs in the remote hosts and finally the generation of the configuration files needed for the run-time. The call sequence is presented in Fig. 3).

The run-time library is able to detect the inherent parallelism of the sequential application and performs concurrent task submission. Techniques such as file renaming, file locality, disk sharing, checkpointing or ClassAds [11] constraints specification are applied to increase the application performance, save computation time already performed or select resources in the Grid. Currently, the machine list has been enabled to be dynamic, so at execution time a user can specify new machines added to the computation, and erase machines which are no more available, as well as changing features from machines (i.e. limit of jobs, submission queue, software available, etc.) Different versions of the run-time exist in order to use different Grid middleware technologies, such as Globus 2.4 [8], Globus 4 [8], Ninf-G2 [12] and ssh/scp. Extra layers between GRID superscalar

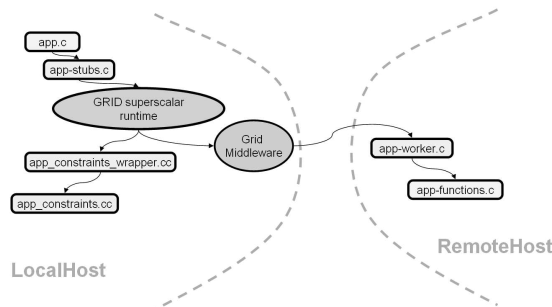

Fig. 3. GRID superscalar behavior

and the underlying Grid middleware, such as SAGA and GAT [2], could ease the step of porting GRID superscalar to new Grid middleware, which is known to be a tedious task.

4 Implementing the GRID superscalar runtime system using SAGA

As shown in Fig. 1, a SAGA layer within the GRID superscalar runtime system replaces the current, direct middleware bindings. This section will show how this replacement works, using the Globus based GRID superscalar run-time version as example. We picked central Globus code snippets as used in GRID superscalar, and compare them to the respective SAGA replacements[3].

4.1 Initialization and Session Management

The Globus module initialization is performed once per session, and does not require more than a single call for each Globus module used. SAGA initializes a default session handle, which is transparently used (see listing in Fig. 4). However, the user *can* have tighter control on that handle, if needed.

4.2 File Movement

Moving files in a Grid from A to B is probably *the* most common use case for Grid applications. Globus provides various means to perform such operations, with different complexity, performance, and behavior. One of the simplest versions is provided by the Global Access to Secondary Storage (GASS) module, as shown in Fig. 5.

One of the major problems for file movement in Grids is that the application programmer or even the end user needs to be aware of the transport protocols

[3] the code snippets are slightly simplified, e.g. do not include proper error handling.

Fig. 4. Code Snippet: *Middleware Initialization*

```
globus_module_activate   (GLOBUS_GASS_COPY_MODULE);
globus_module_activate   (GLOBUS_IO_MODULE);
globus_module_activate   (GLOBUS_GRAM_CLIENT_MODULE);
globus_module_activate   (GLOBUS_COMMON_MODULE);

globus_module_deactivate_all ();
```

```
// no initialization needed in SAGA
```

available: in the code example, the user needs to know that gsiftp is actually available for that host. Although difficult to solve in general, the SAGA API provides a more abstract notion of that transport, by allowing the 'protocol' *any*. The SAGA implementation is then selecting an available protocol for the data transfer.

Fig. 5. Code Snippet: *Copying files (from a remote host to the master)*

```
globus_gass_copy_handle_init (&handle, GLOBUS_NULL);
globus_io_file_open ("/home/workingdir/file.txt",
       GLOBUS_IO_FILE_WRONLY | GLOBUS_IO_FILE_CREAT
                             | GLOBUS_IO_FILE_TRUNC,
       GLOBUS_IO_FILE_IRUSR  | GLOBUS_IO_FILE_IWUSR
                             | GLOBUS_IO_FILE_IRGRP,
       GLOBUS_NULL, &fileHandle);
globus_gass_copy_url_to_handle (&handle,
       "gsiftp://hostname/path/file.txt",
       GLOBUS_NULL, &fileHandle);
globus_io_close (&fileHandle);
```

```
saga::directory dir ("/home/workingdir/");
dir.copy ("file.txt", "any://hostname/path/file.txt");
```

4.3 Remote Job Submission

The simplest way to submit a remote job via Globus is to provide a RSL description of the job to the GRAM module (see Fig. 6). This module will forward the job submission request to the Globus gatekeeper on the remote host, which will then run the job. A job handle is returned for later reference, e.g., for checking the job's status.

The SAGA API has a similar notion of a job description. The keywords used for describing the job are compatible to those specified in GGF's JSDL [3] and DRMAA [5] specifications. As indicated in Section 2, the returned job object also allows performing a number of operations on the job, among which is checking the current job status.

Fig. 6. Code Snippet: *Job submission*

```
// The callback_contact is passed in order
// to receive notifications
sprintf (RSL, "&(executable=/path/exec)(directory=/path)"
  "(arguments=-l)(queue=short)"
  "(file_stage_in = gsiftp://host/path/file1 /path/file1)"
  "(file_stage_out= /path/file2 gsiftp://host/path/file2)"
  "(file_clean_up = /path/file3)"
  "(environment=(NAME1 val1)(NAME2 val2))");
globus_gram_client_job_request (hostname, RSL,
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE  |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_PENDING |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE    |
  GLOBUS_GRAM_PROTOCOL_JOB_STATE_FAILED,
  callback_contact, &job_contac);
```

```
saga::jobdescription jd;

std::list <const char*> hosts; hosts.push_back ("host");
jd.add_vector_attribute ("SAGA_HostList", hosts);

jd.add_attribute ("SAGA_JobCmd",    "/path/exec");
jd.add_attribute ("SAGA_JobCwd",    "/path");
jd.add_attribute ("SAGA_JobArgs",   "-l");
jd.add_attribute ("SAGA_Queue",     "short");
jd.add_attribute ("SAGA_FileTransfer",
                  "/path/file1 < gsiftp://host/file1");
jd.add_attribute ("SAGA_FileTransfer",
                  "/path/file2 > gsiftp://host/file2");
jd.add_attribute ("SAGA_JobEnv", "NAME1=val1 NAME2=val2");
saga::job_server js;
saga::job        job = js.submit_job (jd);
```

4.4 Job State Notification

Via callbacks and blocking polls, Globus provides a very convenient way for GRID superscalar to wait for the completion of submitted Jobs (see Fig. 7). The SAGA API does currently not provide similar mechanisms. However, a monitoring extension to SAGA is in the planning stage, and will similarily allow to add callbacks for changes in the job status metric. SAGA also does not yet provide blocking synchronization points.

4.5 Job Manipulation

The manipulation of jobs, such as `cancel`, `kill` or `signal`, is solved similar in Globus and SAGA (see Fig. 8). However, the SAGA API includes more complex operations such as `migrate`, which are more difficult to implement in the less abstract Globus API's.

4.6 Job State

The set Globus and SAGA job states are somewhat different. We have focused on the equivalence of states needed from GRID superscalar's point of view,

Fig. 7. Code Snippet: *Job state notifications*

```
// TaskEnds treats the state change
globus_gram_client_callback_allow (TaskEnds, NULL,
                                   &callback_contact);
// Blocked waiting for notifications
globus_poll_blocking ();
```

```
saga::metric m = job.find_metrics ("Status");
             m.add_callback (TaskEnds);
while ( ! all_jobs_done ) { sleep (1); }
```

Fig. 8. Code Snippet: *Job cancellation*

```
globus_gram_client_job_cancel (job_contact);
```

```
job.cancel ();
```

thus some states can have a slight semantic difference between them, but they accomplish the minimum semantic needed when working with GRID superscalar. The SAGA API state diagram is very complete, so we can see the Globus states included in the SAGA states. A mapping between both is straight forward: Globus states Pending, Active, Failed and Done are known as Queued, Running, DoneFail and DoneOK in SAGA, respectively.

5 Related work

The SAGA developments are mostly driven by the Grid Application Toolkit (GAT) [2] and the Java CoG kit [15]. While the latter aims at providing a simple API to the Globus toolkit [8], the GAT aims both at a simple API and at middleware-independent implementations. The SAGA API, once standardized by GGF, will allow implementations that are either independent of or integrated into Grid middleware packages.

While GAT and SAGA provide simple API's for Grid-aware applications, higher-level toolkits like GRID superscalar aim at supporting Grid-unaware applications, thus completely hiding the underlying Grid infrastructure. Systems like Ibis [14], Assist [1], and ProActive [6] have similar aims, each providing their own flavors of high-level API's. We believe, however, that GRID superscalar's task-flow model offers a very widely applicable and efficient programming model, which is why we combine is with SAGA to form a Grid programming environment which is both high-level and platform-independent.

Overall, an integration of GRID superscalar and SAGA fits into the bigger picture of building an integrated Grid platform, as envisioned within the Core-

GRID community [7, 9, 13]. In such an integrated platform, both Grid-aware and unaware runtime interfaces will be embedded in a component system with enriched functionality like information providers, resource brokers, and application steering interfaces.

6 Conclusions

The Simple API for Grid Applications (SAGA), as currently standardized within GGF, aims to provide a simple yet powerful Grid API; its implementations shielding applications from the intricacies of existing and future Grid middleware. The GRID superscalar is a programming environment in which Grid-unaware task flow applications can be written, for execution on Grids. As such, GRID superscalar can be seen as a client application to SAGA.

In this paper, we have discussed how SAGA can help implementing GRID superscalar's runtime system, together forming a high-level and platform independent programming environment for the Grid. We have shown that the SAGA API, although simplistic on purpose, already provides almost all functionality needed for enabling a system as powerful as GRID superscalar. The only deficiency is with supporting upcalls and asynchronous event notifications, an issue currently being worked on within GGF's SAGA group. Once completed, the main benefit that GRID superscalar will get is that SAGA will provide an interface that will allow to run unmodified across various Grid middleware systems such as various versions of Globus, Unicore, and even using ssh/scp, or purely local on individual computers. So, SAGA acts as an extra layer, that makes GRID superscalar independent from Globus implementation details. The price that GRID superscalar has to pay is indeed having to add this extra layer in order to use the Grid services, which we think is low in contrast to the benefit obtained.

Acknowledgments

This work is partially funded by the European Commission, via the Network of Excellence *CoreGRID* (contract 004265). Part of the work was carried out in the context of the Virtual Laboratory for e-Science project (*www.vl-e.nl*), supported by the Dutch Ministry of Education, Culture and Science (OC&W). It is also supported by the Ministry of Science and Technology of Spain under contract TIN2004-07739-C02-01.

The SAGA API was defined by GGF's SAGA Research Group, and in particular by the SAGA design team: Shantenu Jha, Tom Goodale, Gregor von Laszewski, Andre Merzky, Hrabri Rajic, John Shalf, and Chris Smith.

References

1. M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured iplementation of component based grid programming environments. In *Future Generation Grids*. Springer Verlag, 2005.

2. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel, and B. Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 93(3):534–550, 2005.
3. A. Anjomshoaa, F. Brisard, R. L. Cook, D. K. Fellows, A. Ly, S. McGough, and D. Pulsipher. Job Submission Description Language (JSDL) Specification Version 1.0. Draft Recommendation, Global Grid Forum (GGF), 2005.
4. R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.
5. R. Brobst, W. Chan, F. Ferstl, J. Gardiner, J. P. Robarts, A. Haas, B. Nitzberg, H. Rajic, and J. Tollefsrud. Distributed Resource Management Application API Specification Version 1.0. GFD.022 - Proposed Recommendation, Global Grid Forum (GGF), 2004.
6. D. Caromel, W. Klauser, and J. Vayssiere. Towards seamless computing and metacomputing in java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, September-November 1998.
`http://www-sop.inria.fr/oasis/proactive/`.
7. CoreGRID Virtual Institute on Problem Solving Environments, Tools, and GRID Systems. Roadmap version 1 on Problem Solving Environments, Tools, and GRID Systems. CoreGRID deliverable D.ETS.01, 2005.
8. I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Int. Journal of Supercomputer Applications*, 11(2):115–128, 1997.
9. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. ICENI: Optimisation of Component Applications within a Grid Environment. *Parallel Computing*, 28(12), 2002.
10. GGF. Simple API for Grid Applications Research Group, 2004.
`http://forge.gridforum.org/projects/saga-rg/`.
11. M. Solomon. The ClassAd Language Reference Manual.
`http://www.cs.wisc.edu/condor/classad/`,.
12. Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, 2003.
13. J. Thiyagalingam, S. Isaiadis, and V. Getov. Towards Building a Generic Grid Services Platform: a component-oriented approach. In V. Getov and T. Kielmann, editors, *Component Models and Systems for Grid Applications*. Springer Verlag, 2005.
14. R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. Bal. Ibis: A Flexible and Efficient Java-based Grid Programming Environment. *Concurrency & Computation: Practice & Experience*, 17(7-8):1079–1107, June-July 2005.
15. G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):643–662, 2001.
`http://www.cogkits.org`.