Generic Support for Bulk Operations in Grid Applications

Stephan Hirmer
Louisiana State University
Baton Rouge, Louisiana, USA
shirmer@cct.lsu.edu

Hartmut Kaiser
Louisiana State University
Baton Rouge, Louisiana, USA
hkaiser@cct.lsu.edu

Andre Merzky
Vrije Universiteit, Amsterdam
Amsterdam, The Netherlands
andre@merzky.net

Andrei Hutanu
Louisiana State University
Baton Rouge, Louisiana, USA
ahutanu@cct.lsu.edu

Gabrielle Allen
Louisiana State University
Baton Rouge, Louisiana, USA
gallen@cct.lsu.edu

ABSTRACT

Within grid environments, latencies for remote operations of any kind can, as the number of operations increases, become a dominant factor for overall application performance. Amongst various approaches for latency hiding, bulk operations provide one possible solution to reduce latencies for large numbers of similar operations. The identification of bulks can, however, pose a non-trivial exercise for application developers, often requiring changes to the implemented remote API, and hence direct code modifications to the application themselves.

In this paper we show how bulk operations can be integrated into existing API implementations, and identify the required properties of the API to make this approach feasible. We also show that our approach considers any type of bulk operation, and is independent of the underlying middleware support for bulks. We further describe a prototype implementation (within the SAGA C++ reference implementation effort), and present performance measurements for bulks of remote file copy operations.

1. INTRODUCTION

Latencies associated with the invocation of remote operations and inter-process communication can significantly affect application performance in distributed systems [1]. The relatively small and constant latencies seen in traditional non-distributed systems, which often don't require any specific handling at the application level, can become a serious issue when dealing with wide area networks, which typically induce latencies several orders of magnitude larger [1]. One way to reduce the overall visible latency for an application is to cluster related remote operations into a single operation, usually known as a *bulk operation*. This gain is visible especially if the application talks to a single middleware service to perform the required operations.

The application of optimized bulk operations would be beneficial to several ongoing grid projects involving multiple remote file operations, including: (i) *petroleum engineering*[1]: working with grid-enabled sensitivity studies of reservoir simulations the UCOMS project deploys over 1000 simultaneous runs, each run producing around 20Mb data in 20 files which must be archived before it is analyzed; (ii) *numerical relativity*[2]: a new project to archive black hole simulation data must cope with, for each run, around 2000 files containing over 50Gb data; (iii) *coastal modeling*[3]: the SCOOP project, working on coastal ocean observing and prediction systems, using and producing several hundred files of different sizes from 200kb to 500MB.

To be able to use bulk operation optimization in a distributed grid environment, some component between the application and grid middleware layer must have the capability to cluster different operations into bulk operations. However, this process will typically need application level information about (a) operation dependencies, and (b) semantic details of the operations to find 'similar' operations. Generally, both types of information are hard to obtain below the application level, hindering the transparent utilization of bulk optimizations. In particular, information about dependencies between operations are usually not explicitly available: that is one of the reasons why automated parallelization of application code is a very difficult and daunting task [2]. However, we show that, for some APIs, implicit information about such dependencies are in fact available, and can be used for transparent bulk optimizations.

In the following, Section 2 introduces the Simple API for Grid Applications (SAGA), our use case for these optimization efforts. Section 3 describes the notion of asynchronous operations, and Section 4 identifies required API properties for transparent bulk optimizations and describes a prototype implementation. Section 6 presents initial benchmark results and evaluates the approach in respect to expected overhead. After a short presentation of related work in Section 7, Section 8 concludes and presents ideas for future work.

2. GRID API'S – SAGA

Grid APIs, designed for developing grid applications, are naturally concerned with performance problems introduced

MGC '06, November 27, 2006 Melbourne, Australia Copyright 2006 ACM 1-59593-581-9/06/11/... $5.00

[1]`http://www.ucoms.org/`
[2]`http://www.cct.lsu.edu/about/focus/numerical/`
[3]`http://scoop.sura.org/`

by latencies associated with remote operations. Such APIs often expose the means for application level latency hiding, such as asynchronous operations [3, 4, 5, 6]. In this paper, asynchronous operations are called *tasks*. An implicit feature of such tasks is that concurrently running tasks have **no** dependency on each other, the application cannot rely on any order of execution for individual tasks. As described later, that implicit knowledge is enough to provide transparent bulk optimization for a number of use cases.

Within the Open Grid Forum (OGF)[4], the SAGA API represents the most application oriented standard layer. The governing design principle for the SAGA API is the 80:20-rule: *provide 80 % of functionality with 20 % effort.* Instead of full coverage, SAGA tries to provide 80 % of the most used grid paradigms for scientific applications [7]. Currently, the SAGA API covers core areas such as file access, replica management, job submission and control, and data streaming. Auxiliary SAGA APIs provide a consistent look & feel, as well as asynchronous operations, notification, and session management [8].

The 80:20-rule also implies that lower level optimizations, for example control over cache sizes, are usually *not* exposed at the API level. This can have far-reaching implications on the performance of remote operations, as those can benefit significantly from latency hiding techniques. Also, several of the core SAGA use cases *require* the efficient support for certain types of operations, such as for bulk job submissions and bulk file access operations [9]. Hence, the SAGA API developers need to show that the current specification will allow for optimized SAGA implementations for these use cases.

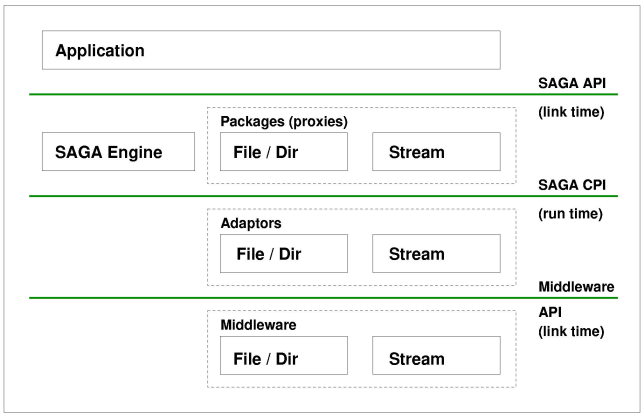


Figure 1: SAGA Architecture: adaptors, loaded at run time, bind a specific SAGA call to the middleware. The SAGA engine coordinates adaptor invocation.

The SAGA specification fostered a C++ reference implementation[5], which learns from the architecture of the GridLab Grid Application Toolkit (GAT) as described in [3] (see Figure 1). In contrast to the GAT engine (written entirely in C), the current SAGA engine (in C++) allows for late binding of middleware plugins (adaptors), and allows fully asynchronous operations by reflecting the SAGA task model at the engine level.

[4]http://www.ogf.org/
[5]provided at http://forge.gridforum.org/sf/projects/saga-core-wg

3. ASYNCHRONOUS OPERATIONS IN SAGA

SAGA defines *tasks* to represent asynchronous remote operations, and provides *task_containers* to simplify handling large numbers of simultaneously running tasks. Simultaneously running tasks have an undefined order of execution, and thus can have no dependencies between each other. The SAGA concept of task containers hence defines, very conveniently, *a set of non-dependent tasks*.

As mentioned earlier, two types of information about remote operations are needed to identify bulk operations: (a) information about operation dependencies, and (b) semantic information about the individual operations, which are used to identify those which can be clustered into a bulk. The SAGA task container obviously allows the retrieval of the first set of information. What about the second set?

Two details of the SAGA task implementation are relevant for this discussion: the encapsulation of a remote operation in a task, and the binding to middleware executing that task. The implementation uses a plug-in oriented architecture to encapsulate middleware specific functionality from the SAGA API. These plug-in's (*adaptors*) are used to dispatch individual SAGA API operations to the corresponding middleware. The adaptors, binding to a specific middleware, are clearly the place to perform possible bulk optimizations, as any bulk support at the middleware level can be exploited here explicitly.

Code Example

```
{
  vector <string> files = ...;
  saga::task_container tc;

  // create file copy tasks
  while ( files.size () )
  {
    saga::file f (files.pop ());
    tc.add (f.copy <saga::Task>
               ("/data/"));
  }

  // run all tasks
  tc.run ();

  // wait for all tasks
  tc.wait ();
}
```

Figure 2: ***Simple example illustrating a implicit bulk file copy operation in the SAGA C++ reference implementation***

The implementation architecture is shown in Figure 1: there, the SAGA engine selects the correct adaptor at runtime, and dispatches the method to that adaptor. For this purpose, meta data information about that method call is evaluated (what object type is the method called on; is the call synchronous or asynchronous etc.). It is interesting that similar meta data can be used to recognize *similar* operations, e.g. operations which represent the same method call on the same object instance! Hence, the fact that our SAGA implementation has, internally, some means of method inspection provides the mechanism for obtaining the additional information needed.

4. BULK OPERATIONS WITHIN THE SAGA C++ REFERENCE IMPLEMENTATION

4.1 The Big Picture

The code excerpt in Figure 2 shows one way to copy a large number of files asynchronously into one (local) directory. As already outlined, due to the general structure of the SAGA API, and more particularly to the existence of the task container, SAGA implementations *can* learn what operations can be executed in a bulk, and which cannot[6]. In our implementation, the application of certain *similarity heuristics* identifies similar tasks in a task container.

The bulk construction capabilities provided by the described approach are not restricted to operations invoked from points close together in the code. Since the bulk detection is performed during the execution of the task container `run()` function, our approach works well even if the related tasks are constructed in distributed points of the code. This function defines the starting point for the bulk detection regardless of the points in code where the different tasks have been created.

The main factor of optimzation provided by our approach stems from avoiding network operations during the *invocation* of remote operations. For instance, instead of needing one network round trip and GSI authorization per file copy invocation the corresponding bundled bulk will require one network and one authorization operation only. If several remote services are involved, this gain may be partially compensated, though, since each of these services may add its own latencies to the overall operation execution time.

This section describes an implementation of optimized bulk handling, and heuristics to define clusters of similar tasks to be bulk handled.

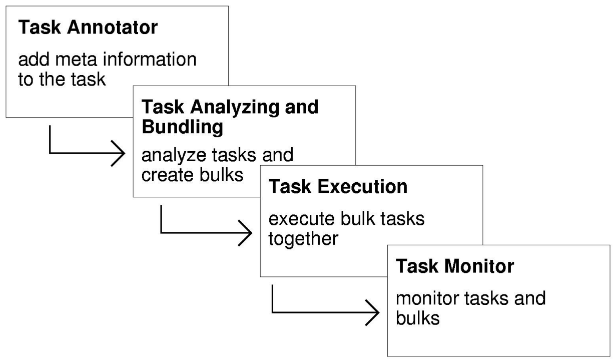

Task Annotator
add meta information to the task
Task Analyzing and Bundling
analyze tasks and create bulks
Task Execution
execute bulk tasks together
Task Monitor
monitor tasks and bulks

Figure 3: Architecture of the automatic bulk handling module

Figure 3 shows the overall schema for the bulk handling module. The arrows indicate the order of component execution. The monitoring component can potentially be invoked after the creation of a task, but usually will only be used after the task has run.

[6]It must be noted that the information allows only a very conservative result – however, by design the SAGA task containers are especially useful for large sets of tasks, which encompasses exactly those use cases where bulk optimizations are most useful.

The information used for identifying bulk operations (4.2 and 4.3), heuristics applied to this information to identify bulks (4.4), and execution by the *Bulk Execution Entity* (4.5) is described in the following. Fail safety issues are also discussed in 4.5

4.2 Required API Properties

In the above, two essential pieces of information necessary for bulk optimizations were identified: knowledge about task (in-)dependencies, and knowledge about semantic task-similarity. These lead to three API properties which must be fulfilled to allow transparent implementation of bulk optimizations.

4.2.1 *Explicit Asynchronous API*

For synchronous function calls, the application programmer assumes immediate execution, which implies that, apart from caching, there is no possibility for delayed execution and latency hiding. Only complex code analysis can reveal possible task dependencies – in general it is assumed that *all* operations must be finished before a new one is started.

Explicitly asynchronous APIs are a precodition for transparent bulk optimizations at runtime, since they provide the ability to identify candidates for bulk operations. Synchronous operations are not taken into account during bulk detection.

4.2.2 *Information about Task Dependencies*

Most APIs with explicit support for bulk optimization, such as DRMAA [4] and gLite [10], require the application developer to specify task dependencies, and to bundle only independent tasks into bulk operations. However, as mentioned earlier, any set of tasks running at the same time can be considered as independent, as tasks, by definition, do not guarantee any execution order (they are completely asynchronous). For that reason it is possible that external code analysis tools are able to find task dependencies at compile time, and are able to perform bulk optimizations (and other optimizations, such as parallel execution) transparently to the application programmer.

Further, the concept of a task container in the SAGA API, which by design allows for multiple simultaneously running tasks, provides such information implicitly, without the need for additional pre-processing steps.

Container classes for sets of concurrently running tasks provide implicit information about independent tasks.

It is obvious that independent tasks which are *not* managed within a task container cannot benefit from a bulk optimization without additional information – pre-processing provides a more optimal set of independent tasks. That problem is not solved in our implementation.

4.2.3 *Information about Task Similarities*

One prominent property of task bulks is that they cluster semantically similar operations, e.g. many read operations, to the same file. In fact it would make no sense to cluster, for example, many reads to different files, as that bulk operation would need to be communicated to all involved resources, which then could only execute parts of the bulk. Hence, another requirement is the ability to identify similarities of remote operations. That poses no specific requirement to the API itself, but it seems advantageous to have an implementation which allows for inspection of remote operations, and which can so gather semantic information about the operations.

An API implementation should allow inspection of remote operation requests in order to find semantically similar tasks.

The SAGA engine meets that requirement: all operations are routed through the engine, and inspection is already performed to route the calls to a 'suitable' adaptor. Most adaptor based API implementations will have similar schemes in place, so it seems that plugin oriented API implementations are likely well prepared to add additional inspection for bulk analysis.

4.3 Adding Metainformation

As the SAGA engine analyses tasks for *similarity*, various meta information attached to the tasks are evaluated, and heuristics are applied to obtain a similarity measure. The information required depends mostly on the adopted clustering strategy (see 4.4), but generic descriptive information (such as function name, parameter values, class names, and class instances) are usually available and promising candidates. Our implementation adds these information while encapsulating the operation into its internal task representation.

4.4 Analyzing & Bundling operations

The *Analyzing and Bundling Entity* is responsible for applying clustering heuristics to a given set of tasks. These heuristics try to determine which tasks are semantically similar, using the attached metadata as described above. So far, the following clustering strategies have proved useful.

4.4.1 Clustering Heuristics

Figure 4 summarizes the different possible clustering heuristics considered during our implementation.

operation	class	bulk
same	same	yes
same	different	no
different	same	yes
different	different	no

Figure 4: Similarity cases considered for bulk clustering heuristics

- *Same functions – Same API class:* the analyzed tasks encapsulate the same operations, which have been called on the same API class. All the tasks are considered as belonging to one bulk bundle.
- *Same functions – Different API classes:* if the originating object instances belong to unrelated classes (in terms of inheritance), tasks are not considered suitable for bulk optimization. If there is an inheritance relation between the classes, optimization might be worthwhile – however, this requires runtime reflection mechanisms, which are difficult to implement and rarely portable.
- *Different functions – Same API-class:* different operations invoked on the same class type might be called on the same object, or not. Either way it seems worthwhile to try bulk optimization, as some middleware allows the execution of different functions on the same object instance. A good example for that are `read()` and `readv()` on the same file instance.
- *Different functions – Different API-class* No obvious relation between different tasks, and they are not considered suitable for bulk optimization.

4.5 Executing Bulk Operations & Fall-Back Mechanism

The *Bulk Execution Entity* (see Figure 3) executes a given bulk of identified similar tasks. By applying standard adaptor selection, an appropriate adaptor is choosen, the bulk is passed to that adaptor, and is executed. If the execution of some tasks fails, the bulk execution entity tries to select another adaptor for the set of failed tasks, which then effectively constitute a new bulk. If all available adaptors have tried to execute a given task bulk and unexecuted tasks still remain, the fall-back-mechanism is applied and the tasks are executed one-by-one. If the middleware does not support bulk operations, bulk handling will obviously not lead to any speedup. However, additional latency hiding mechanisms such as parallel task execution are still applicable.

An adaptor to a bulk enabled middlware has, in our implementation, to implement a specific bulk handler to be called by the Bulk Execution Entity. As seen in Figure 1, a separate adaptor interface (Capability Provider Interface, CPI) is already defined, which was easy to extend with such bulk handlers.

5. THE PROTOTYPE IMPLEMENTATION

5.1 Implementation of Bulks

The described concept for the handling of bulk operations was implemented within the SAGA C++ reference implementation (see Section 2). The SAGA engine was extended to allow the harvesting of semantic information for the operations encapsuled by the tasks, as motivated above. The general task model structure needed no change, which proved, in our opinion, that the design of the SAGA API does actually allow for bulk optimizations as required by the SAGA use cases [7, 9].

5.2 Example Bulk Adaptor

The implemented adaptors are used for testing and benchmarking, and focus on bulk data transfer, because of its importance for a majority of grid applications. The example adaptor interfaces to the GridFTP [11] based GridLab FileService[7]. This service offers basic filesystem operations (e.g. copy, move, remove, change directory etc.) exposed via a web service interface, and (partly) usable for bulks.

As discussed, a number of changes to the adaptor interface (the CPI in Figure 1) have been necessary: apart from adding CPI methods to expose the bulk execution part of the adaptor, it was also necessary to change the parameter passing mechanism. Function call parameters are now stored within the task class, which allows the adaptor to extract them as needed, and to pass them to the underlying middleware. Furthermore, extensions were made to enable the monitoring of tasks, regardless if they run within a bulk or as single tasks.

As usual, simplicity and transparency of the API comes at the cost of complexity of the implementation of the API. However, API simplicity is the primary goal of the SAGA design, implementation simplicity is not [7].

[7]`http://www.gridlab.org/WorkPackages/wp-8/file_service/`

6. RESULTS

It is well known that bulk operations often perform better than the corresponding one-by-one serial execution [12]. The implementation described here however adds additional layers of code for the transparent detection of bulks. This section discusses the measured overhead of this additional processing, which we find acceptable if compared to the latencies induced by a single remote operation.

The wall time elapsed for the execution of file copy operations was measured, both for direct invocation of the bulk web service interface, and for the indirect invocation via the complete chain of SAGA calls, task analysis, bulk creation, adaptor invocation, and finally again service invocation. The tested code example performs a copy of a variable number of files (each 1MB) from one remote location to another. The benchmarks were initiated on a laptop computer (using a wireless network connection), with the Gridlab File Service running on a remote server in the same LAN. Both source and destination for the file copy were also located on two different hosts in the same LAN. The network was not isolated in any way, and shows random traffic. Figure 5 shows the results.

Note that bulk optimizations target the communication between client and server, and not the file transfers themself. Thus, bulk optimizations are typically useful large numbers of small remote requests, as is the case here (we optimize the communication of file copy *requests*). The files copy operations do barely influence the speedup. For a detailed performance model for bulk operations, see [13].

Figure 5(a) compares the time needed to copy n $(1 \ldots 500)$ files by directly calling the service interface, with the time needed to perform the same operations using SAGA. Figure 5(b) shows the procentual overhead induced by the automated bulk detection. These benchmarks were performed under real life conditions (no isolated network, no isolated servers), the data present the mean of 10 measurements. The straight line in figure 5(b) shows the linear regression[8] for the overhead. As expected, the overhead is zero for single tasks (analysis is trivial), and increases steadily with the number of tasks to analyze. A 12 % overhead can be seen for about 500 tasks.

The latency for the used network is fairly low, so the measured overhead is a high boundary if compared to real long-distance networks. As the network latencies increase, the gain from bundling operations into bulks will increase as well, as bulk operations decrease the number of network connection round trip operations.

7. RELATED WORK

Several middleware frameworks offer explicit bulk operations at the middleware level. For example, the *Globus Reliable File Transfer (RFT)*[9] can copy, move and delete several files at a time. GridFTP[10], provides, via its ERET/ESTO mechanism, for clustering multiple operations into a single remote request [14]. *gLite* [10] offers bulk operations for their client side command-line tools.

Many application level APIs use the collation of individual operations into bulks as a means of performance optimization. We list two example influential APIs for this work, many others can be found in the literature.

POSIX scattered read and write (`readv/readv`) are essentially bulk operations. They are used for large numbers of operations on local files. It is interesting that these calls implicitly create bulks of seek and of read or write operations, hence creating bulks with tasks of mixed semantics. However, that scheme targets one very special (though very important) use case, and it is not immediately clear if mixed task bulks are useful or feasible in a more generic context.

The DRMAA API [4] provides support for bulk job submission in grid environments, with explicit, application level expression of task dependencies and semantics. In fact, the DRMAA group in GGF was a major motivating force behind the bulk optimization efforts in SAGA. We are pleased that the bulk DRMAA use cases are realisable with the current SAGA API.

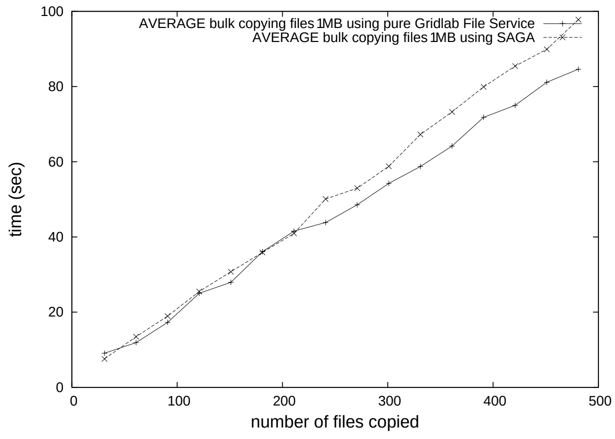
Bulks are not always used for latency hiding – for example, the motivation to include bulk job submission into DRMAA was originally to allow for job parametrization.

We have not been able to identify any related work in terms of automated clustering of asynchronous operations to enable building of bulks.

[8] $a = 0.03 \pm 0.01, b = -0.9 \pm 3.7$

[9] `http://www.globus.org/grid_software/data/rft.php`

[10] `http://www.globus.org/grid_software/data/gridftp.php`



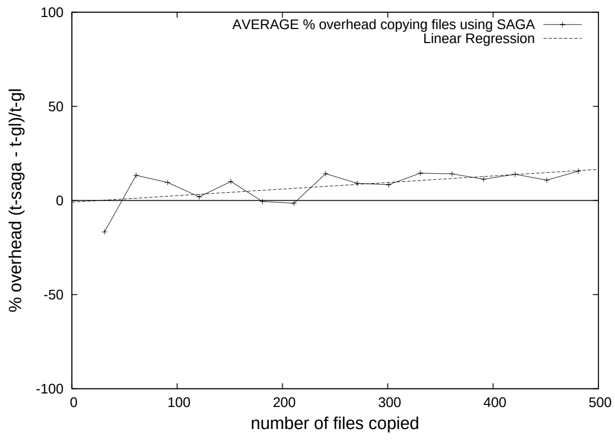


Figure 5: Benchmark results: direct middleware bulk operations vs. indirect SAGA initiated bulk operations. Top: absolute times. Bottom: relative overhead and regression

8. CONCLUSIONS & FUTURE WORK

This paper showed that bulk optimizations can be implemented within the SAGA API specification. In particular the existence of task containers provides the means to identify independent tasks, and to optimize their execution.

Further, we identified three requirements which, when met, allow for bulk optimizations in API implementations without changing the API:

1. the API must be explicitly asynchronous,
2. the API must explicitly or implicitly expose task dependency information,
3. the API implementation must be able to inspect tasks in order to find similar tasks, in respect to some semantic measurement of similarity.

The described prototype implementation implied several changes to the SAGA adaptor interface and engine, to support task analysis and clustering. Given the generic design of our approach, the bulk optimization can be applied to any type of operation, while bulk execution is only possible if middleware support for bulks is available. A fall-back mechanism for non bulk-enabled middleware is provided, which reverts to un-optimized one-by-one execution.

Benchmark runs show that the overhead introduced by bulk optimization amounts to, as a pessimistic estimate, about 12 % per 500 tasks. Far better results are expected on long distance networks, and on larger task granularity.

In the near future, we aim to verify our concept of bulk handling by applying it to other APIs. We would also like to extend our implementation to include other bulk enabled middleware. Further we plan to explore mechanisms which automatically choose between competing bulk-enabled services, by evaluating the type of operation to be executed. That work will increase the amount of data used for the bulk analysis, with the hope to optimize the bulk detection heuristics for semantically diverse tasks.

9. ACKNOWLEDGMENTS

We thank the Center for Computation & Technology (CCT) at LSU and Vrije Universiteit, Amsterdam, the EnLightened project (NSF grant 0509465), and the XtreemOS project (EU grant IST-FP6-033576) for funding the SAGA C++ reference implementation and CCT for hosting Stephan Hirmer for this work. We acknowledge the specification efforts of the GGF SAGA Research Group and Core Working Group. Stephan Hirmer also thanks his thesis advisers, Jacques Calmet and Yves Denneulin.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.11 [**Software**]: Software Architectures—*Domain-specific architectures*; D.2.13 [**Software**]: Reusable Software—*Reusable libraries*

10. REFERENCES

[1] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. *Mobile Object Systems, Lecture Notes in Computer Science, No*, 1222:49–64, 1997.

[2] R.M.J. Badia, J.J. Labarta, R.J. Sirvent, J.M.J. Pérez, J.M.J. Cela, and R.J. Grima. Programming Grid Applications with GRID Superscalar. *Journal of Grid Computing*, 1(2):151–170, 2003.

[3] Gabrielle Allen, Kelly Davis, Tom Goodale, Andrei Hutanu, Hartmut Kaiser, Thilo Kielmann, Andre Merzky, Rob van Nieuwpoort, Alexander Reinefeld, Florian Schintke, Thorsten Schütt, Ed Seidel, and Brygg Ullmer. The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. *Proceedings of the IEEE*, 2004.

[4] H. Rajic, I.A. Inc, W. Chan, I.B.M.F. Ferstl, A. Haas, and J. Tollefsrud. Distributed Resource Management Application API Specification. Technical report, Global Grid Forum, September 2002. GFD.22.

[5] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. *3rd International Workshop on Grid Computing*, 2002.

[6] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, 2001.

[7] Andre Merzky and Shantenu Jha. A Requirements Analysis for a Simple API for Grid Applications. Technical report, Global Grid Forum, 2006. GFD.71.

[8] Tom Goodale, Shantenu Jha, Hartmut Kaiser, Thilo Kielmann, Pascal Kleijer, Gregor von Laszewski, Craig Lee, Andre Merzky, Hrabri Rajic, and John Shalf. SAGA: A Simple API for Grid Applications – High-Level Application Programming on the Grid. *Computational Methods in Science and Technology: special issue "Grid Applications: New Challenges for Computational Methods"*, 8(2), SC05, November 2005.

[9] Andre Merzky and Shantenu Jha. Simple API for Grid Applications – Use Case Document. Technical report, Global Grid Forum, March 2006. GFD.70.

[10] Rüdiger Berlich, Marcel Kunze, and Kilian Schwarz. Grid computing in Europe: from research to deployment. In *CRPIT '44: Proceedings of the 2005 Australasian workshop on Grid computing and e-research*, pages 21–27, Darlinghurst, Australia, 2005. Australian Computer Society, Inc.

[11] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster. The Globus Striped GridFTP Framework and Server. *SC'2005.*

[12] EF Walker, R. Floyd, and P. Neves. Asynchronous remote operation execution in distributed systems. *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 253–259.

[13] Andrei Hutanu, Stephan Hirmer, Gabrielle Allen, and Andre Merzky. Analysis of Remote Execution Models for Grid Middleware. In *Submitted to the 4th International Workshop on Middleware for Grid Computing)*, Melbourne, Australia,, 2006. ACM.

[14] Thorsten Schütt, Andre Merzky, Andrei Hutanu, and Florian Schintke. Remote Partial File Access using Compact Pattern Descriptions. In *IEEE/ACM 4th Intl. Symp. on Cluster Computing and the Grid – CCGrid-2004*, pages 1–8, April 2004. Workshop on Adaptive Grid Middleware.