Programming Models for Grid Applications and Systems: Requirements and Approaches

Thilo Kielmann

Vrije Universiteit, Amsterdam, The Netherlands
kielmann@cs.vu.nl

Abstract

History repeats itself. Since the invention of the programmable computer, numerous computer scientists keep dedicating their professional lives to the design of "the single, best" programming model, whereas programmers "vote" by choosing their favourite languages and tools.

Interestingly, these choices have always been guided by non-functional properties. For programming a single computer, the most widely used models have become object-oriented and component-based programming, a choice driven by their high abstraction level, leading to high programmer productivity. For parallel computers, the winner turned out to be message passing, providing by far not the highest-possible abstraction level, but the closest match between machine architecture and programming model, leading to efficient program execution.

For grids, the race is still open. Here, additional non-functional properties like fault-tolerance, security, and platform independence enter the scene. In this paper, we explore the scope of grid programming problems and argue for a palette of programming abstractions, each suitable for its respective problem domain.

1 Introduction

Grid computing had once been introduced via the analogy to the electrical power grid: the idea of "plug and run" suggested that grids could execute user programs on some unknown, remote resource, without any further user intervention of effort. Reality turned out to be different. Users typically have to spend lots or efforts to make their applications suitable for grids.

Grids turned out to be highly complex environments, with frequent variations of resource availability and operativeness. Due to the combination of multiple, independent administrative domains, along with different security policies and deployed middleware, writing and deployment of grid applications turned out to be a challenging task.

To address this problem, many grid programming environments have been introduced, the authors of many, if not all of them, are claiming that their tool provides the single, right way of writing grid applications.[1] [2]

In this paper, we outline the different views of application programmers and of middleware developers on the right properties of grid programming environments (Sections 2 and 3). In Section 4, we explore the different levels of virtualization provided by different programming approaches, and in Section 5, we map different types of grid applications and tools to appropriate programming environments.

2 Required properties of grid application programming environments

In previous work [15], we have identified a set of required properties for grid application programming environments. We summarize our earlier findings as follows, distinguishing between functional and non-functional properties. Doing so, we follow the perspective of the application developer, as shown in Figure 1.

2.1 Functional properties

Functional properties describe the functionality required by applications to run in grid environments.

- Job submission, spawning, and scheduling
  Users enter application jobs to the grid via some form of job submission tool, like *globusrun* [27], or a portal like GridSphere [22], typically using a resource broker service [24] for making decisions about job placement and scheduling. A running job may also spawn off further jobs to available grid resources.

[1] This, of course, also includes the author.

[2] However, these claims typically need to be taken with a "grain of salt."

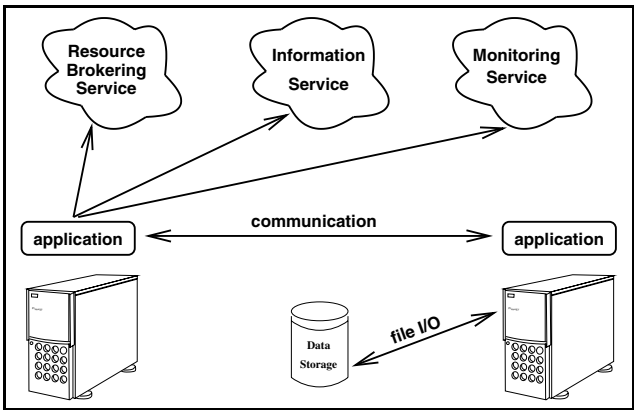

Figure 1. The application developers view on an application and its interaction with the grid environment.

- Access to file and data resources
  Any real-world application has to process some form of input data, be it files, data bases, or streams generated by devices like radio telescopes [26]. A special case of input files is the provisioning of program executable files to the sites on which a job has been scheduled. Similarly, generated output data has to be stored on behalf of the users.

- Inter process communication
  Besides data access, the processes of a parallel application need to communicate with each other. Several programming models for grid applications have been developed, among which are MPI [14], shared objects [18], or remote procedure calls [21].

- Application monitoring and steering
  In case of long running applications, users need to track their progress in order to avoid costly repetition of unsuccessful jobs. For this purpose, users need to inspect and possibly modify the status of their application while it is running on some nodes in a grid. Hence, monitoring and steering interfaces have to be provided, such that users can interact with their applications.

2.2 Non-functional properties

The non-functional properties are determining the constraints on grid API functionality. As such, issues like performance, security, and fault-tolerance influence the suitability of certain programming abstractions.

- Performance
  As high-performance computing is one of the driving forces behind grids, performance is the most prominent, non-functional property of grid operations. Job scheduling and placement are mostly driven by expected execution times, while file access performance is strongly determined by network proximity between computation and data storage. The trade-off between abstract functionality and controllable performance is a classic since the early days of parallel programming [4]. In grids, it even gains importance due to the large physical distances between the sites of a grid.

- Fault tolerance
  According to [19], grid applications fail due to faulty configurations, middleware failures, application failures, and also due to hardware failures and network outages. These factors, in combination with administrative site autonomy, cause various error conditions. (Transient) errors are common rather than the exception. Consequently, error handling becomes an integral part of grid runtime environments and grid APIs.

- Security and trust
  Grids integrate users and services from various sites. Communication is typically performed across insecure connections of the Internet. Both properties require mechanisms for ensuring security of and trust among partners. A grid API thus needs to support mutual authentication of users and resources, as well as access control to resources (authorization).

- Platform independence
  It is an important property for programming environments to keep the application code independent from details of the grid platform, like machine names or file system layouts for application executables and data files, or even the middleware *du jour*.

3 Required properties of programming environments for grid systems

As listed so far, we consider these properties as the direct needs of grid application programs. In contrast, architects of grid middleware have a view that is more biased towards middleware internals and less towards which properties get exposed to applications. In this section, we summarize architecture (Figure 2) and the corresponding, required properties from [8], following its bottom-up description.

- Fabric layer
  The fabric layer covers most of the functional requirements from Section 2.1, except for the *collective* services of a grid. These are computing resources, storage resources, network resources, code repositories, and catalogs.

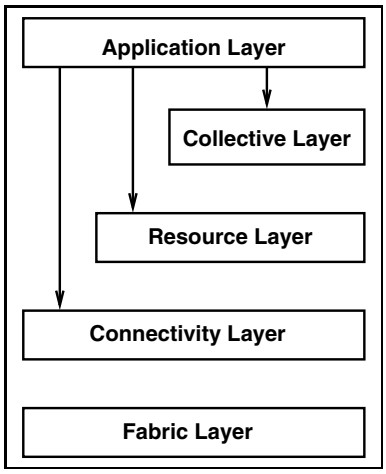

Figure 2. The layered Grid middleware architecture, simplified from [8].

- Connectivity layer
  The connectivity layer combines security mechanisms (authentication, delegation, trust) with providing network connectivity, a functional property that is typically overlooked, or simply assumed to be available, by application programmers.

- Resource layer
  The resource layer provides protocols for both obtaining information about structure and state of a resource, and for managing policies for accessing resources. The information protocols can be used for delivering monitoring information about resources, which is only of indirect interest to applications. Management of resource access policies likewise is part of what application programmers consider "security" aspects of accessing computing, data, or other resources.

- Collective layer
  The collective layer provides protocols and services capturing interactions across collections of resources. These are directory, monitoring, and information discovery services as well as workload management, scheduling and brokering services. Also, collective authorization and accounting services are located here. This collective service layer has been refined in the Open Grid Services Architecture (OGSA) [9] to comprise execution management services, data services, resource management services, security services, self-management services, and information services.

  Interestingly, also grid-enabled programming systems like special MPI versions [10, 14] and manager-worker frameworks [6, 12] are located in the collective layer. This layer provides a very diverse set of functionalities, ranging from non-functional properties (e.g., security), via functional properties (e.g., scheduling) to grid programming models.

- Application layer
  User applications, according to [8], are constructed by calling services from the three layers underneath (from Figure 2). This leads to applications that are written based on using the terms of the underlying services, via service API's that are implemented by software development kits (SDK's), using the service protocols in order to access the provided functionality.

3.1 Relating application programmer and middleware developer viewpoints

Contrasting the two different viewpoints, it becomes obvious that what is "non functional" to application programmers becomes "functional" to middleware architects. Clearly, the non-functional aspects like connectivity and security need to be implemented in some layer. The layered middleware architecture is indicating that there are different layers of abstraction and functionality, each contributing a different piece of the puzzle. Consequently, different API's are providing the respective abstractions.

The applications as anticipated in the application layer of [8] cover only one possible class of grid applications, namely those that are aware of running in a particular grid environment.[3] In the following, we will describe the scope of grid applications and systems software, along with their respectively required levels of programming abstractions.

4 Virtualization layers in grid environments

Virtualization is the predominant purpose of all (grid) middleware. Likewise, operating systems virtualize the resources of a single computer; and cluster operating systems like, e.g., Kerrighed [20] virtualize the resources of a cluster computer in order to provide a single, powerful system instead of a collection of individual nodes. Recent approaches to building a grid operating system extend this idea [30].

Grid middleware virtualizes resources in a grid. The connectivity layer provides a virtual network between the individual resources, the resource layer virtualizes the different resources via a uniform access and management interface, and the collective layer virtualizes the individual (virtualized) resources to form an overall system of resources. Each level of virtualization provides a uniform interface to a variety of heterogeneous, individual entities. With each virtualization layer, the use of the resources becomes simpler,

[3] The possibility of building grid-enabled programming environments, however programmed directly using service API's, is mentioned, too.

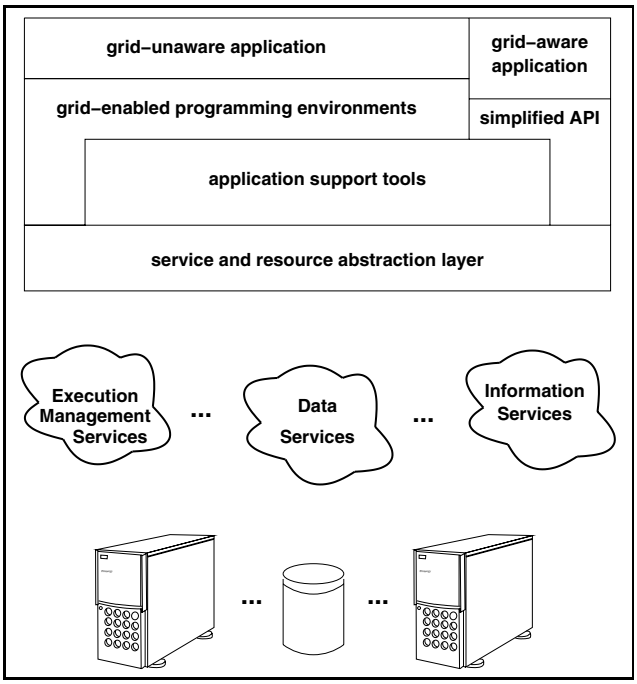

Figure 3. A virtualization hierarchy for grid programming systems.

however at the price of loosing part of the control over the resources which may cause problems with non-functional aspects, like system performance.

Figure 3 is providing a more detailed view on virtualization abstractions provided to grid applications. The bottom of the figure shows (virtualized) grid resources, and services from the collective layer. The top shows a detailed stack of programming abstractions.

The bottom of the stack forms the service and resource abstraction layer. This layer provides an API that virtualizes the different grid middleware systems like Globus [27] or Unicore [7]. Other virtualizations are possible as well, even of non-grid infrastructures, like for example ssh. The most prominent example of such a layer is the Grid Application Toolkit (GAT) [2].

On top of this layer sits a set of (optional) application support tools. Such tools provide additional functionality, adding value to plain middleware services, likely tailored to application needs. Examples for these tools can be found in CoreGRID's proposed mediator component toolkit [16]. These components are supposed, for example, to provide application-level metadata or to dynamically tune and steer running applications. Such components add another layer of virtualization by providing more comprehensive functionality, on top of the plain service and resource virtualization.

An important, original design goal of the GAT was to provide a simplified API for application programmers. Meanwhile, the GAT has been recognized as an attractive programming platform because of its service and resource abstraction functionality. Many developers are currently also trying to build application support tools using the GAT, requesting and leading to additional functionality and hence less simplicity in the GAT interface.

Independent of the GAT, within the Open Grid Forum, we are working on a Simple API for Grid Applications (SAGA) that will provide a standardized, simplified grid API [13]. With SAGA, simplicity stems from both uniformity across different middleware platforms and also from the reduction to functionality that is needed to run application code, purposefully excluding features needed for management or monitoring of the grid services and resources themselves. The current divergence of GAT and SAGA interfaces is another indication that different programming tasks require different API's. Implementing the simpler and more abstract SAGA API on top of the more detailed GAT interface will be a viable design option.

Both GAT and SAGA interfaces provide virtualization interfaces for grid-aware (or "grid-enabled") applications. These are applications that are explicitly using grid resources like submitting additional jobs to other machines or accessing remote files or data bases. This is the class of applications as envisioned in the application layer of [8].

A different class of applications is grid-unaware. Such applications treat a grid as a completely virtualized execution environment. We will distinguish two different kinds of grid-unaware applications in the next section. What they have in common is the need for grid-enabled programming environments that provide the right degree of virtualization. Examples of such programming environments are special MPI versions [10, 14], remote object-based systems like Ibis [29] or ProActive [23], manager-worker frameworks [6, 12], the Satin divide-and-conquer system [28], the GridSuperscalar task flow system [3], systems for parameter studies like SEGL [16] or for workflows like Triana [25], or systems providing algorithmic skeletons [1] or higher-order components [11]. While these systems differ largely in API and functionality, they all have in common that they provide some execution abstraction for (parallel) applications allowing to write applications that can run on grids without being explicitly programmed for.

To summarize, the spectrum of virtualization layers provide different degrees of abstraction. They all trade the benefits of virtualization (simplicity, uniformity) against its drawback: lack of control, negatively impacting both performance and granularity of security and trust policies. In the following section, we will detail the different kinds of grid applications and tools, along with their respectively suitable programming environments.

5 Types of grid programs and their programming environments

Table 1 summarizes the types of grid programs and which kind of programming environments suit them most. We begin our discussion with the lowest level of virtualization.

Table 1. Grid applications and tools, and their most suitable programming environments.

type of code	environment
legacy codes	sandboxing
parallel applications	grid-enabled environment
grid-aware codes	simplified API
support tools	service and resource abstraction layer
service and resource management	service and resource interfaces

- Service and resource management
  Software for management and administration of resources and services needs access without any further virtualization or abstraction. Here, full control over status and features is the dominant goal. In fact, there is little need for virtualization unless a single administrator will be in charge of a large variety of services and resouces. This type of software will best be programmed directly using the respective service API's.

- Support tools
  Application support tools are providing value-added functionality for applications and as such are operating across multiple services and resources. While they need virtualization and abstraction of the individual entities in a grid, they still need detailed control, for example of security policies or performance information. These tools are thus best programmed using a service and resouce abstraction layer like the GAT.

- Grid-aware codes
  Grid-aware codes are those application programs that are programmed (or re-programmed) to use grid resources explicitly, like spawning off jobs to other machines or using remote files and data bases. These programs are best written using a simplified API like SAGA that restricts the programming environment to application needs, abstracting away such features needed to monitor or manage services and resources.

- Parallel applications
  Parallel applications are written in order to achieve fast execution by using many compute resources together. For running parallel applications in grids, they need to be programmed by explicitly using a grid-enabled, parallel programming environment.

  Many such grid-enabled environments exist, varying largely by programming abstraction and also by non-functional properties. For example, MPI provides high-performance communication but has difficulties dealing with failures, with dynamically changing sets of resources, and with heterogeneous compute resources. Other environments, like Satin or higher-order components, allow for more flexibility by higher abstractions which is beneficial for dynamic environments. It may, in return, have adversary effects on application performance if the programmer looses too much control which might not always be compensated by automatic runtime optimizations.

- Legacy codes
  Legacy codes are such applications that have been written without any grid envrionment in mind and which can not be modified to become grid-aware or at least parallelized by a grid-enabled environment. Running such codes in a grid environment requires a level of virtualization that emulates the machine for which the codes once had been written, including processor architecture, operating system, and file system layouts. Different approaches have been proposed to achieve this goal [5, 17].

6 Conclusions

We have outlined the different viewpoints of both application programmers and middleware designers on properties and API's of grid middleware and systems. We have discussed the various degrees of virtualization and abstraction provided by different systems and programming environments. A discussion of different types of applications and support tools has shown that there is no single, "one size fits all" programming model for grid software. Different problems require different approaches in order to cope with non-functional aspects of programming that in fact determine the right choice of tools. There will always be some trade-off between programmer productivity (by good abstractions) and program efficiency or resilience to failures and changing environments. Depending on how much effort programmers are able or willing to spend on converting their codes to become grid-aware, different solutions are to be preferred.

Acknowledgments

This work is partially supported by the *CoreGRID* Network of Excellence, funded by the European Commission's FP6 programme (contract IST-2002-004265). The author would like to thank his many colleagues from the network for the many fruitful and stimulating discussions.

References

[1] M. Aldinucci, M. Coppola, S. Campa, M. Danelutto, M. Vanneschi, and C. Zoccolo. Structured implementation of component based grid programming environments. In *Future Generation Grids*. Springer Verlag, 2006.

[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] 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.

[4] H. Bal, J. Steiner, and A. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, 1989.

[5] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *ACM SOSP*, pages 164–177, 2003.

[6] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. In *Proc. SC'2000*, 2000.

[7] D. Erwin, editor. *Joint Project Report for the BMBF Project UNICORE Plus*. UNICORE Forum e.V., 2003.

[8] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International J. Supercomputer Applications*, 15(3), 2001.

[9] I. Foster, H. Kishimoto, A. Savva, D. Berry, A. Djaoui, A. Grimshaw, B. Horn, F. Maciel, F. Siebenlist, R. Subramaniam, J. Treadwell, and J. V. Reich. The Open Grid Services Architecture, Version 1.0. Grid Forum Document, GFD.30, 2005. Global Grid Forum.

[10] E. Gabriel, M. Resch, T. Beisel, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *5th European PVM/MPI Users' Group Meeting*, volume 1497 of *LNCS*, pages 180–187, Liverpool, UK, 1998. Springer Verlag.

[11] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*. Springer Verlag, 2006.

[12] J.-P. Goux, S. Kulkarni, J. Linderoth, and M. Yoder. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *Proc. HPDC*, 2000.

[13] S. Jha and A. Merzky. A Collection of Use Cases for a Simple API for Grid Applications. Grid Forum Document, GFD.70, 2006. Global Grid Forum.

[14] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 2003.

[15] T. Kielmann, A. Merzky, H. Bal, F. Baude, D. Caromel, and F. Huet. Grid Application Programming Environments. In *Future Generation Grids*, pages 283–306. Springer Verlag, 2006.

[16] T. Kielmann, G. Wrzesinska, N. Currle-Linde, and M. Resch. Redesigning the SEGL Problem Solving Environment: A Case Study of Using Mediator Components. In *Integrated Research in Grid Computing*. Springer Verlag, 2006.

[17] S. Klous, J. Frey, S. Son, D. Thain, A. Roy, M. Livny, and J. van den Brand. Transparent access to Grid resources for user software. *Concurrency and Computation: Practice and Experience*, 18:787–801, 2006.

[18] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8–9):681–712, 2001.

[19] R. Medeiros, W. Cirne, F. Brasieiro, and S. J. Faults in grids: Why are they so bad and what can be done about it? In *Fourth International Workshop on Grid Computing*, pages 18–24. IEEE Computer Society, 2003.

[20] C. Morin, P. Gallard, R. Lottiaux, and G. Vallée. Towards an Efficient Single System Image Cluster Operating System. *Future Generation Computer Systems*, 20(2), 2004.

[21] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and H. Casanova. A GridRPC Model and API for End-User Applications. Grid Forum Document, GFD.52, 2005. Global Grid Forum.

[22] J. Novotny, M. Russell, and O. Wehrens. GridSphere: A Portal Framework for Building Collaborations. In *1st International Workshop on Middleware for Grid Computing*, Rio de Janeiro, 2003.

[23] ProActive. http://www.inria.fr/oasis/ProActive.

[24] J. M. Schopf, J. Nabrzyski, and J. Weglarz, editors. *Grid resource management: state of the art and future trends*. Kluwer, 2004.

[25] I. Taylor, M. Shields, I. Wang, and O. Rana. Triana Applications within Grid Computing and Peer to Peer Environments. *Journal of Grid Computing*, 1(2):199–217, 2003.

[26] The GEO600 project. http://www.geo600.uni-hannover.de/.

[27] The Globus Alliance. http://www.globus.org/.

[28] R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient Load Balancing for Wide-area Divide-and-Conquer Applications. In *Proc. PPoPP '01: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 34–43, 2001.

[29] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, 2005.

[30] XtreemOS: Building and Promoting a Linux-based Operating System to Support Virtual Organizations for Next Generation Grids. http://www.xtreemos.org, 2006.