SIMPLIFYING GRID APPLICATION PROGRAMMING USING WEB-ENABLED CODE TRANSFER TOOLS

Cătălin L. Dumitrescu, Jan Dünnweber, Philipp Lüdeking, Sergei Gorlatch
Department of Mathematics and Computer Science, University of Münster, Germany
CoreGRID Institute on Programming Models
{dumitres,duennweb,muli,gorlatch}@uni-muenster.de

Ioan Raicu
Department of Computer Science, University of Chicago, USA

Ian Foster
Department of Computer Science, University of Chicago, USA
Mathematics and Computer Science Division, Argonne National Laboratory, USA
{iraicu,foster}@cs.uchicago.edu

Abstract This paper deals with one of the fundamental properties of grid computing – transferring code between grid nodes and executing it remotely on heterogeneous hosts. Contemporary middleware relies for this purpose on Web Services, which makes application programs complicated and low-level and requires much additional expertise from programmers. We compare two mechanisms for grid application programming with regard to their handling of code transfer – the de-facto standard WS-GRAM in Globus and the higher-level approach based on HOCs (Higher-Order Components). We study the advantages and problems of each approach using a real-world application case study – the sequent alignment problem from bioinformatics. Our experiments show the trade-off between reduced development costs and software complexity when HOCs are used and the higher performance of the applications on the grid when using WS-GRAM.

Keywords: Distributed Systems, Resource Management, Service Level Agreements

This research work was supported in part under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

1. Introduction

Grids aim to provide a transparent access to large-scale computing, networking, and storage resources. A fundamental property of grid applications is the transfer of not only data but also executable code between the nodes of the grid. Contemporary middleware, for example the Globus Toolkit, relies on Web Services for this purpose, such that a typical grid application usually consists of two parts: a) an operational part that handles computations in a parallel and distributed manner, and b) a declarative part which includes resource specifications, interface definitions, deployment descriptors etc., in an XML-based format; it describes, e. g. , how to encode a program for transmission and how to invoke code on a remote host. Therefore, grid application programming remains a quite low-level, complicated task that requires from the programmer much additional expertise beyond his particular application area.

In this paper, we study and compare two approaches to grid application programming with regard to how they manage the aspect of code transfer:

- The first approach is the Globus Resource Allocation Manager (WS-GRAM [2]) which is currently the most often used solution. In WS-GRAM, code is packaged as a job which is a Web service parameter of a special type. Each job carries an executable program, packaged together with a description of the parameters this program expects and the program's requirements concerning the processors and libraries available on the execution platform. WS-GRAM extends the Web service standards (WSDL and SOAP [15]) by a descriptive XML-based language RSL [2] for job definitions, since usually the types used for the parameters of Web services are plain data types rather than executable programs. Contrary to the static interface of a Web service, users upload RSL definitions to the service at runtime.
- The second approach seeks to raise the level of programming abstraction by relying on Higher-Order Components (HOCs [6]). HOCs are application-level components: they provide implementations of typical, recurrently used coordination patterns in parallel applications. HOC users implement only application-specific operations and pass them to the HOCs as code parameters. The Service Architecture for HOCs (HOC-SA [3]) is currently a Globus incubator project [4]. Using HOCs, the work of application programmers and the amount of code transferred in the grid is fairly reduced, because much work is done by HOC-SA.

In the remainder of the paper, we compare the process of grid application programming using these two approaches: WS-GRAM and HOC-SA. After describing the general properties of the both approaches in Section 2, we consider in Section 3 a particular application example - detection of similarities in genome sequences - and present how this application is implemented using each of the two approaches. In Section 4, we describe the results of extensive experiments with the genome sequence application which demonstrate the performance and the development costs of the application when using WS-GRAM and HOC-SA. We discuss related work and conclude in Section 5.

2. Grid Programming with WS-GRAM and the HOC-SA

From the user's viewpoint, a possibility to request the processing of application tasks over the Internet instead of only downloading application data is probably the feature which makes the grid most distinct from the Web.

2.1 Code transfer and Web services

The technology enabling the remote processing of application tasks is currently Web services in most grid middleware systems. Web services were created with the aim of increasing the interoperability among heterogeneous platforms by handling the exchange of data over the network using portable formats. The parameters accepted by a Web service must be encoded in an XML document. For any Web service this document must adhere to an XML Schema which the service developer writes into the WSDL description of the service interface [15]. However, Web services were not designed to exchange executable code. There is no representation for executable code in the XML Schema format. Therefore, Web service developers must declare code as plain data skipping the context information about the code, i. e., the information about how to invoke the code from within another program is potentially not available to the execution host, as explained in the following.

Generally, there are two different types of code a client can upload to a server in a distributed system: a self-contained program or a part of a program. Both types must be invoked in a different manner. In the case that clients upload a self-contained program, the context information required to invoke this program consists of the data format and sequence of the program input. Moreover the libraries, command line arguments, environment variables and also parameters like, e. g., the number of MPI processes used in the program must be communicated to the executing server. A Java class containing its own `main` method also falls under the category of self-contained programs: it is a portable binary that requires the same context information for remote execution as a native binary program.

In the second case, if the code uploaded by a client is only a part of a program, it must be a well-defined entity in the full program, e. g., a class or block which contains one or multiple procedure definitions. To insert this entity remotely into the proper context, the executing server requires information about the code format (source or binary) and programming language, which is not necessarily the case for self-contained programs, as these are either in a binary format or scripts that can be interpreted by the remote shell. To execute non-self-contained code remotely, the interface for accessing this code within the server-side context, i. e. , the signatures of the procedures inside, must be declared in a file available to the execution host which must assign proper variable types to input and output. Both, command line options and interfaces have no standard representation in XML Schema making it difficult to define Web service interfaces for transferring any code, full or partial programs.

Thus, the code transfer problem that we address in this paper originates from the fact that the context information of a local code, i. e. , the required information on how to execute it, is potentially lost when the code is transferred as a Web service parameter from one context (e. g. , a client program) to another.

2.2 Web-enabled Code Transfer with WS-GRAM

Fig. 1 shows how the WS-GRAM service of Globus avoids the potential loss of information discussed above, when full programs are transferred: The client code (left in the figure) contains a call to WS-GRAM where application code is submitted as a job to the execution hosts. The shaded hexagons in the WS-GRAM job represent processes connected by arrows representing message exchange.

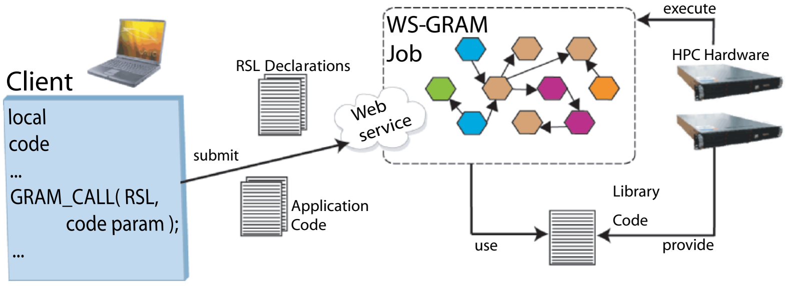


Figure 1. Code Transfer via Job submission in WS-GRAM

Besides the application code, the client sends one RSL declaration per job. The programmer must write these declarations to provide the information about how to invoke the transferred code to the execution hosts. RSL files are uploaded by WS-GRAM users at runtime together with the code described therein. RSL belongs to the declarative part of a WS-GRAM application and extends the static configuration which any Web service-based software requires (WSDD & WSDL [15]). While the introduction of RSL to the Web service configuration enables the use of different kinds of executable code as Web service parameters, WS-GRAM requires users to be familiar both with the service configuration and with RSL. In Fig. 1, there is one RSL declaration document per application unit (transferred in a single submission, typically a class).

2.3 Web-enabled Code Transfer with HOC-SA

Higher-Order Components (HOCs) handle the platform-specific work automatically (data format conversions etc.) and require only the application-specific pieces of code being sent to them. This principle is depicted in Fig. 2: The client runs an application that uses a HOC which executes on the remote High-Performance Computers (HPC) with code parameters provided by the client. The HOC - Service Architecture (HOC-SA) provides a solution for code transfer in which only a part of a program is sent over the network. While the generic components (HOCs) are pre-installed, the application-specific code parameters are transferred on demand (i. e. , at runtime as explained in the following). HOC-SA adds to WS-GRAM two elements, the Code Service and the Remote Code Loader. However, HOC-SA makes the situation simpler for the user as compared to WS-GRAM, since the upload of code (step ① in Fig. 2) is decoupled from using the code (step ②), as explained in the next paragraph.

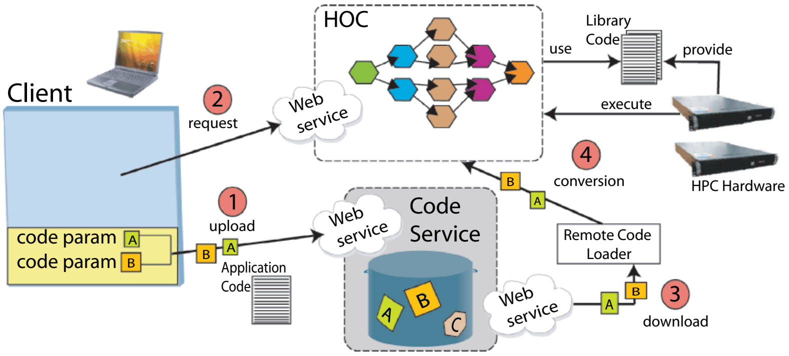


Figure 2. Code Transfer to a component (HOC) via the HOC-SA

In the upload step ①, the application code is intermediately stored in the Code Service, a Web service connected to a database (via OGSA-DAI [11]). Identifiers specified by the user ($\boxed{A}$ and $\boxed{B}$ in the figure) are linked to the uploaded code, making it a code parameter. Users can pass a code parameter to a HOC by referring to its identifier. Both HOCs and their code parameters can be reused in many different applications. The code parameter transfer (lower part of the client-side code) is not necessarily contained in the client application, but becomes rather an administrative action: HOC-SA includes a Web-based portal allowing programmers to browse the Code Service and check if a code parameter with the required functionality is available: if not, then a new code parameter can be stored using our portal (or hand-written code). Thus, in the HOC-SA, code transfer is separated from the application code, such that both Code Service and Remote Code Loader are not visible to the programmer.

In the `HOC(A,B)`-call in step ②, no code is transferred. It is an ordinary Web service request that is served by a HOC. HOCs execute recurring communication patterns (the pattern in Fig. 2 is called wavefront [3]; see Section 3 for an example). For transparently inserting code parameters into appropriate positions in the pattern implementation, the HOC-SA performs two steps invisible to the application programmer: in the download step ③, the code that the identifiers refer to is transferred to the HPC hardware (which is more than one host, in case of a distributed HOC implementation). The conversion step ④ is performed by the Remote Code Loader which is locally placed on the execution host(s) and makes the downloaded code parameters executable there. This conversion is done by cast operations which assign the code parameters their proper types. The type of HOC being used also determines the type of the code parameters (e. g. a certain class definition), such that RSL for describing HOC code parameters is unnecessary. In Fig. 2, the application code is much smaller than in Fig. 1 since it represents only a piece of a program (HOC parameters)

3. Application case study for the HOC-SA and WS-GRAM

Both WS-GRAM and HOC-SA delegate the handling of grid-specific requirements like file transfer and (to a certain extend) fault-tolerance and security to the Globus middleware [5]. However, WS-GRAM and HOC-SA provide different means for code transfer on top of Globus, which have advantages in different scenarios. In the following case study, we demonstrate best practices for code transfer, using HOCs for simplifying the use of the middleware and using WS-GRAM for giving programmer more control over the middleware and potentially better performance.

As a case study, we present an application that detects similarities in genome sequences. Although there are already many implementations for genome sequence alignment, our component-based solution features exchangeable processing modes, such that the same distributed procedure is employed to detect different kinds of similarities. The implementation is based on the Alignment HOC (a GRAM-based variant without code parameters is shown in Section 4). It fills a scoring matrix that rates differences between character sequence pairs: each matrix element holds the result of a user-defined scoring function applied to the two input subsequences, delimited by the element's indices. Besides the scoring function, the Alignment HOC has two more code parameters: the alignment function and the traceback function. The alignment function is used to iterate over the sequences and compute the scores, i. e., the users are in control over the computation order allowing them, e. g., to run a parallel schema. Eventually, the traceback operation is applied to the scoring matrix to produce the result (e. g., this is often a path through the matrix). While the code parameters allow users to run any kind of sequence rating with the Alignment HOC, a standard Smith & Waterman alignment [13] can be computed without writing any code parameter, since the Alignment HOC provides a default scoring, alignment and traceback function for this purpose.

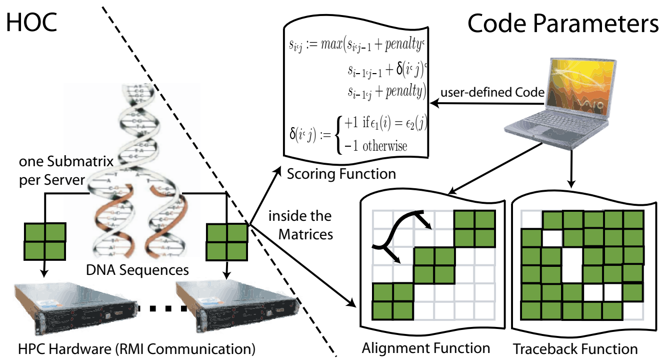

HOC
Code Parameters
$s_{i,j} := max(s_{i,j-1} + penalty,$
$s_{i-1,j-1} + \delta(i,j),$
$s_{i-1,j} + penalty)$
$\delta(i,j) := \begin{cases} +1 \text{ if } \epsilon_1(i) = \epsilon_2(j) \\ -1 \text{ otherwise} \end{cases}$
user-defined Code
Scoring Function
one Submatrix per Server
inside the Matrices
DNA Sequences
HPC Hardware (RMI Communication)
Alignment Function
Traceback Function

Figure 3. Computation Schema of the Alignment HOC for DNA Similarity Detection

As shown in Fig. 3, the Alignment HOC is a distributed component which uses RMI for dispatching parts of its input to multiple RMI servers (they are to be launched by the user in advance). For our example application, we specified a scoring and a traceback function that search for circular permutations [14]. While the default alignment function of the Alignment HOC allows for any kind of data dependences (even among non-neighboring input elements) and works sequentially, we implemented an alignment function that executes the parallel processing schema known as the wavefront pattern [3]. It partitions the matrix into submatrices which are positioned along the matrix antidiagonals and form a wavefront. The submatrices are processed by the servers which need no synchronization, since the wavefront pattern guarantees that there are data dependences only inside the submatrices but not amongst them. While the computations proceed through the matrix, there is a varying number of parallel processes active, as shown in the processing schema in Fig. 2. The alignment function is an example for code reuse (and, thus, reduced code transfer costs) in the HOC-SA: it can be used as a code parameter in different applications (with potentially different HOCs), even in applications of the wavefront pattern that neither process genome data nor use RMI servers.

For implementing the sequential processing inside the code parameters, we used the JAligner library [10]. As shown in Section 2.3, the HOC-SA provides to each HOC a specific library code in a server-sided repository. The Alignment HOC, e. g., has access to JAligner, allowing us to use it without transferring it to the servers ourselves.

The decision to choose one particular code transfer technology depends on the relation between application-independent parts that can be handled by a HOC and the size of the code parameters. An advantage of using HOC-SA, is that it has a relatively small footprint, i. e., can outsource tasks to remote machines which do not run Globus containers of their own. If the target environment has the Globus container installed on every node per default, it is always worth to consider using both WS-GRAM and HOC-SA. In the next section, we compare the two technologies regarding their performance.

4. Performance Comparison: HOC-SA vs. WS-GRAM

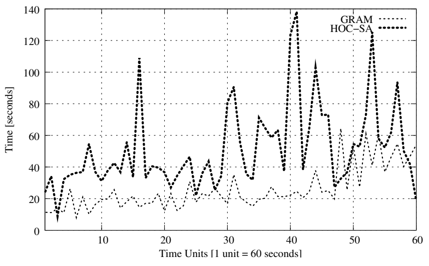
In this section, we compare the performance of the HOC-SA implementation as a Globus Incubator project [4] with WS-GRAM [2] from the Globus Toolkit.

We use two implementations of the genome similarity detection application described in Sec. 3: one on top of HOC-SA and one on top of WS-GRAM. The amount of operational code including the Alignment HOC is approximately 18K lines of code for both versions. The application-specific code, i. e., the part the user writes in the HOC-SA version is only 400 lines of code long. Since the Alignment HOC is an application-independent component, it is typically pre-installed in a server-side repository in the HOC-SA. Therefore, HOC-SA reduces the network traffic at startup of our application by factor 45.

All our tests were generated and submitted by means of the DiPerF tool in ServMark [1], a specialized tool for grid performance evaluation. HOC-SA was installed on grid nodes at the University of Chicago, with the following characteristics: Dual-Intel Xeon(TM) 3.0 GHz with Hyper-threading, 2.0 Gb of Memory and 100Mb/s network connectivity. Clients were deployed on the PlanetLab nodes [12], which are Linux PCs connected to the PlanetLab overlay network with worldwide distribution. Most nodes are connected via 100 Mb/s network links (some still have 10Mb/s links) over a wide-area network (WAN), have processor speeds exceeding 1.0 GHz IA32 Pentium III processors, and at least 512 MB RAM.

Our metrics used for quantifying the performance of the two implementations are as follows: (1) execution time (response) is the average time elapsed from job submission to finish; (2) throughput quantifies the number of computations that terminated and returned a result (i. e., completed requests) of the service per second, and (3) load represents the number of clients that use the service concurrently at a certain moment in time.

Figure 4 captures the performance WS-GRAM and HOC-SA in terms of response time. In some of experiments, the performance of WS-GRAM was up to two times higher than the performance of HOC-SA. We explain this by the highly optimized processing of repeated requests in WS-GRAM (e. g., via caching). On average, WS-GRAM showed lower variations for response time; the spikes can be explained by the temporal incapacity of the service to serve all requests, caused by the communication with the OGSA-DAI and the RMI servers. The performance of HOC-SA improves when the Code service and the RMI servers are placed in the same LAN and depends on the LAN



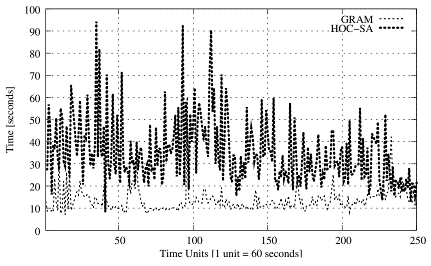
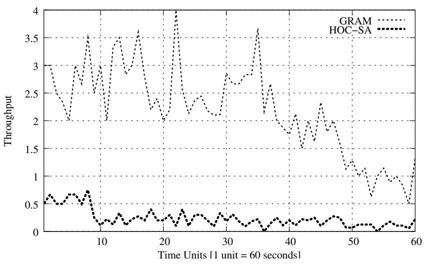


Figure 4. GRAM and HOC-SA response time in seconds on the dual Xenon processor with 40 (left) and 20 (right) clients running in PlanetLab for 1800 (left) or 3600 seconds (right)

capabilities [8]. Thus, the use of the HOC-SA Code service and Remote Code Loader for code transfer comes at a certain cost when the resources are widely dispersed.

However, we also measured that WS-GRAM, while delivering better response times, caused a processor load that was on average 20% higher than in the HOC-SA scenario. Thus, HOC-SA can provide a higher availability (in terms of responsiveness), when the number of concurrent clients rises.



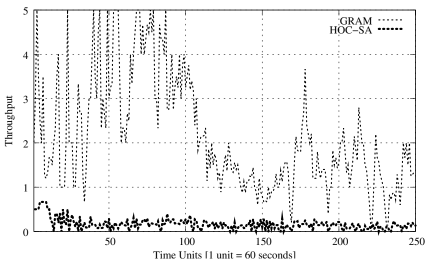


Figure 5. GRAM and HOC-SA throughput on the dual Xenon processor with at most 40 (left) and 20 (right) clients running on PlanetLab. Tunable parameters: starts a new client every 30 (left) and 180 (right) seconds, each client runs for 1800 (left) and 3600 (right) seconds

Figure 5 shows that the performance of WS-GRAM in terms of throughput is up to three times higher as in HOC-SA. This is explained by the use of gridFTP instead of SOAP for transferring the sequence files in the WS-GRAM version.

5. Conclusions and Related Work

This paper has compared two approaches to grid application programming – WS-GRAM and HOC-SA – with regard to how they handle code transfer over the network and its execution on the grid nodes. For our experiments we used a real-world application, which is successfully used in bioinformatics: this program recently scanned the genome database ProDom [14] and found pattern matches that were not known previously [8].

An important advantage of HOC-SA is that it frees the user from writing any declarative code. The traditional declaration of communicated data and code in grid applications becomes unnecessary due to pre-built components and a special portal. HOC-SA also allows users to perform code transfer by accessing the Code Service directly from within the application code. Other grid portal projects, e. g. , Ganga [7] and gridPort [9], also support the transfer of code. But these projects cannot be used for building new applications: their purpose is the placement of existing applications onto grid nodes. Ganga and gridPort also do not allow to submit to the grid application code that has data dependencies.

In our experiments, HOC-SA demonstrated an advantage of simplified application programming and reduced network traffic. If an application performs computations that are so time-consuming that they justify the costs for handling every part of them with hand-tuned code, then WS-GRAM is probably the best choice for implementing it. Since the time costs of most grid applications depend on the amounts of data being processed, a HOC that provides a distributed processing schema can help speeding up these applications by increasing the number of execution nodes. Thus, HOC-SA proves to be a viable extension to WS-GRAM in today's grids. Grid applications can also benefit from both technologies, HOC-SA and WS-GRAM, simultaneously: WS-GRAM can transfer a HOC together with its code parameters or any other self-contained part of a HOC-SA application.

References

[1] C. Dumitrescu, A. Iosup, I. Raicu, M. Ripeanu. ServMark: A Distributed Grid Testing Framework, 2006. http://dev.globus.org/wiki/Incubator/ServMark.

[2] K. Czajkowski, I. Foster, C. Kesselman et al.. A Resource Management Architecture for Metacomputing Systems . In *IPPS/PDP'98 Workshop*, pages 62–82, 1998.

[3] J. Dünnweber et al. Adaptable parallel components for Grid programming. *Integrated Research in GRID Computing*, pages 43–59. Springer Verlag, 2006.

[4] J. Dünnweber, P. Lüdeking, C. L. Dumitrescu, E. Argollo, and S. Gorlatch. The HOC-SA Globus Incubator Project. Web page: http://dev.globus.org/incubator/hoc-sa/, 2006.

[5] I. Foster and C. Kesselman. Globus Toolkit *Supercomputer Journal*, 11(2):115–128, 1997.

[6] S. Gorlatch and J. Dünnweber. From Grid Middleware to Grid Applications: Bridging the Gap with HOCs. In *Future Generation Grids*, pages 299–306. Springer Verlag, 2005.

[7] K. Harrison et al. Ganga: a Grid User Interface for Distributed Analysis. In S. J. Cox, editor, *Proceedings of Fifth UK e-Science All-Hands Meeting*. EPSRC, 2006.

[8] P. Luedeking. MS thesis: Proteine sequence analysis in the Grid with the HOC-SA, 2006.

[9] Maytal Dahan et al. Grid Portal Toolkit 3 (gridport). In *Proceedings of the 13th HPDC*, pages 272–273, Washington, DC, USA, 2004. IEEE.

[10] A. Moustafa. The JAligner library http://jaligner.sourceforge.net, 2007

[11] Grid Data Access and Integration OGSA-DAI www.ogsadai.org.uk, 2007

[12] L. Peterson et al. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the ACM HotNets)*, October 2002.

[13] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[14] J. Weiner, G. Thomas, and E. Bornberg-Bauer. Rapid Motif-based Prediction of Circular Permutations in Multi-domain Proteins. *Bioinformatics*, 21:932 – 937, 2005.

[15] W3C: . XML protocol recommendations, http://www.w3.org/2002/ws, 2002.