A COMPONENT-BASED INTEGRATED TOOLKIT

Enric Tejedor and Rosa M. Badia
Universitat Politècnica de Catalunya
Barcelona, Spain
etejedor@ac.upc.edu
rosab@ac.upc.edu

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

Vladimir Getov
University of Westminster
London, UK
V.S.Getov@westminster.ac.uk

Abstract This paper presents the Integrated Toolkit, a framework which enables the easy development of Grid-unaware applications. While keeping the Grid transparent to the programmer, the Integrated Toolkit tries to optimize the performance of such applications by exploiting their inherent concurrency when executing them on the Grid. The Integrated Toolkit is designed to follow the Grid Component Model (GCM) and is therefore formed by several components, each one encapsulating a given functionality identified in the GRID superscalar runtime.

Currently, a first functional prototype of the Integrated Toolkit is under development. On the one hand, we have chosen ProActive as the GCM implementation and, on the other, we have used JavaGAT as a uniform interface to abstract from the underlying Grid middleware when performing job submission and file transfer operations. Thus far, we have tested our prototype with several simple applications, showing that they maintain the same behaviour as if they were executed locally and sequentially.

Keywords: Integrated Toolkit, components, Grid Component Model, Grid-unaware applications, concurrency exploitation, performance optimization.

1. Introduction

This paper focuses on the specification and design of the *Integrated Toolkit*: a framework which enables the easy development of Grid-unaware applications (those to which the Grid is transparent but that are able to exploit its resources). The Integrated Toolkit is mainly formed by an *interface* and a *runtime*. The former should give support to different programming languages, graphical tools and portals, and should provide the application with a small set of API methods. The latter should provide the following features:

- The Grid remains as transparent as possible to the application. The user is only required to select the tasks to be executed on the Grid and to use few API methods.
- Performance optimization of the application by exploiting its inherent concurrency. The possible parallelism is checked at task level, automatically deciding which tasks can be run at every moment. The most suitable applications for the Integrated Toolkit are those with coarse-grain tasks.
- Task scheduling and resource selection taking into account task requirements and performance issues.

This paper is organized as follows. We begin by proposing a design for the Integrated Toolkit in Section 2. Then, using a simple example, we give some usage and operation details of a first Integrated Toolkit prototype in Section 3. After that, we present some preliminary tests of the prototype in Section 4. Finally, we describe some related work in Section 5 before the conclusions and future work of Section 6.

2. A GCM-based design of the Integrated Toolkit

This document proposes an Integrated Toolkit based on the Grid Component Model (GCM) [1], a component model intended for the Grid which takes the Fractal specification [2] as reference. Therefore, the Integrated Toolkit runtime is defined as a set of Fractal components, each of them in charge of a given functionality. The design, inspired on the GRID superscalar framework [3], comprises the following components:

- *Task Analyser* (TA): receives incoming tasks and detects their precedence, building a *task dependency graph*. It implements the interface used by the application to submit tasks: when such a request arrives, it looks for data dependencies between the new task and all previous ones. When a task has all its dependencies solved, the TA sends it to the Task Scheduler.
- *Task Scheduler* (TS): decides where to execute the dependency-free tasks received from the TA. This decision is made accordingly to a certain

scheduling algorithm and taking into account three information sources: first, the available Grid resources and their capabilities; second, a set of user-defined constraints for the task; and third, the location of the data required by the task. The scheduling strategy could also be changed on demand, thanks to the dynamic and reconfigurable features of GCM.

- *Job Manager* (JM): in charge of *job submission and monitoring*. It receives the scheduled tasks from the TS and delegates the necessary file transfers to the File Manager. When the transfers for a task are completed, it transforms the task into a Grid job in order to submit it for execution on the Grid, and then controls the proper completion of the job. It could implement some kind of fault-tolerance mechanism in response to a job failure.

- *File Manager* (FM): takes care of all the operations where files are involved, being able to work with both logical and physical files. It is a composite component which encompasses the *File Information Provider* (FIP) and the *File Transfer Manager* (FTM) components. The former gathers all information related with files: what kind of file accesses have been done, which versions of each file exist and where they are located. The latter is the component that actually transfers the files from one host to another; it also informs the FIP about the new location of files.

3. Usage and operation example of the Integrated Toolkit

Taking the design presented in Section 2 as reference, we are working on a first Integrated Toolkit prototype. Regarding the implementation choices, we took Java as the programming language and ProActive 3.2 and JavaGAT 1.6 as the base technologies.

ProActive [4] is a Java Grid middleware library for parallel and distributed computing. Among some other features, it provides an implementation of the Fractal specification with some extensions, thus contributing to the development of GCM. Hence, our Integrated Toolkit is in fact formed by ProActive components and benefits from the following GCM properties: hierarchical composition, separation between functional and non-functional interfaces, synchronous and asynchronous communications, collective interactions between components and ADL-based description of the component structure.

JavaGAT is the Java version of the Grid Application Toolkit [5], which is a generic and flexible API for accessing Grid services from application codes, portals and data management systems. The calls to the GAT API are redirected to specific adaptors which contact the Grid services, thus offering a uniform interface to numerous types of Grid middleware. Our Integrated Toolkit uses JavaGAT for job submission and file transfer operations.

The following subsections explain, through a simple example, how to write an application that uses the Integrated Toolkit and which call sequences between subcomponents take place when executing it.

3.1 Original code of the sample application

Consider a Java application which generates random numbers and cumulatively sums them (from now on, we will call it *Sum*). Figure 1 shows its main code.

```
initialize(f1);
for (int i = 0; i < 2; i++) {
    genRandom(f2);
    add(f1, f2);  // f1 <- f1 + f2
}
print(f2);
```

Figure 1. Original code of Sum. All method parameters (f1, f2) are file names. After putting a zero value in f1 (initialize), random numbers are generated (genRandom) and then added to the accumulated sum stored in f1 (add).

3.2 Selecting the tasks and inserting API method calls

In order to make the application use the Integrated Toolkit, the programmer is only required to write a Java interface declaring the tasks that will be executed on the Grid and to use few API methods.

```
public interface SumItf {
    @MethodConstraints(operatingSystemType = "Linux")
    void genRandom(
        @ParamMetadata(type = Type.FILE, direction = Direction.OUT)
        String f
    );
    @MethodConstraints(processorArch = "Intel", processorSpeed = 1.8f)
    void add(
        @ParamMetadata(type = Type.FILE, direction = Direction.INOUT)
        String f1,
        @ParamMetadata(type = Type.FILE, direction = Direction.IN)
        String f2
    );
}
```

Figure 2. Annotated interface for Sum

Concerning the interface, Java annotations [11] must be used to specify some metadata about the tasks. On the one hand, it is mandatory to state, for each

parameter of a task, its type (currently, we only support the file type) and its direction (IN, OUT or INOUT). On the other, the programmer can also impose the constraints that a given resource must fulfil to execute a certain task (regarding, for instance, the operating system or the architectural characteristics) Figure 2 corresponds to the interface of Sum, containing the mentioned metadata.

Regarding the API, it offers methods to start and stop the Integrated Toolkit, request the execution of tasks and open files to work with them locally. Figure 3 shows the final code of Sum, resulting from the inclusion of API calls. Currently, the programmer has to deal with all these methods but, in the future, we will implement a mechanism to free (totally or partially) the application developers from that duty; for that purpose, some possible alternatives could be a source-to-source compiler, a code generation tool or a modified Java class loader.

```
initialize(f1);
IntegratedToolkit it = new IntegratedToolkitImpl("Sum");
it.startIT();
ITExecution itExe = (ITExecution)it;
for (int i = 0; i < 2; i++) {
    itExe.executeTask("genRandom", 1,
                      f2, ParamType.FILE_T, ParamDirection.OUT);
    itExe.executeTask("add", 2,
                      f1, ParamType.FILE_T, ParamDirection.INOUT,
                      f2, ParamType.FILE_T, ParamDirection.IN);
}
String finalF2 = it.openFile(f2, OpenMode.READ);
print(finalF2);
it.stopIT(true);
```

Figure 3. Code of Sum with calls to the Integrated Toolkit API

3.3 Internal processes and communications

This section describes the main internal processes of the Integrated Toolkit which are triggered when executing the Sum application, that is, what each subcomponent does, which communications take place between subcomponents and in which order.

3.3.1 Initialization. After being deployed and started, the components must be initialized. For that purpose, the Integrated Toolkit has a multicast interface which transforms a single initialization invocation on the runtime into a list of invocations and forwards them to all the subcomponents.

3.3.2 Task analysis, scheduling and job submission. When the initialization phase finishes, the task processing can begin. As said in Section 2, the

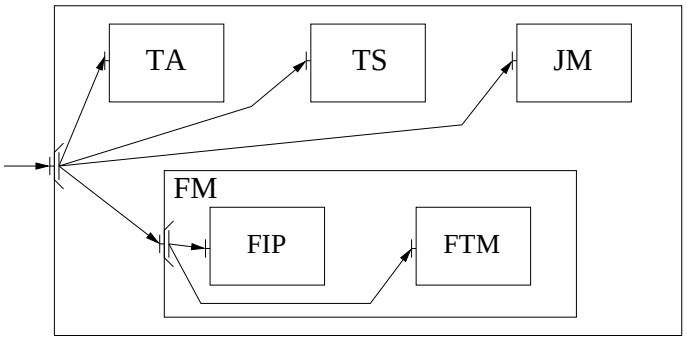

Figure 4. Initialization of the Integrated Toolkit

TA is the component which receives task execution requests from the application. In the case of Sum, a total of 4 tasks will be issued (2 per iteration).

The TA registers the file accesses of a task with the help of the FIP, which keeps track of the file versions that are eventually created: whenever a task writes a file it creates a new version of that file, and this new version is assigned a renaming. Then, the TA discovers the dependencies between the task and all previous ones, thanks to a structure where it stores the last writer task for each file. The current Integrated Toolkit only considers file dependencies, while in future versions other kinds of data (scalars, arrays, etc.) will be taken into account.

The task dependency graph for Sum is the one depicted in Figure 5. The dependencies represented with dashed lines are automatically removed by means of the renaming technique, so that only RaW ones remain.

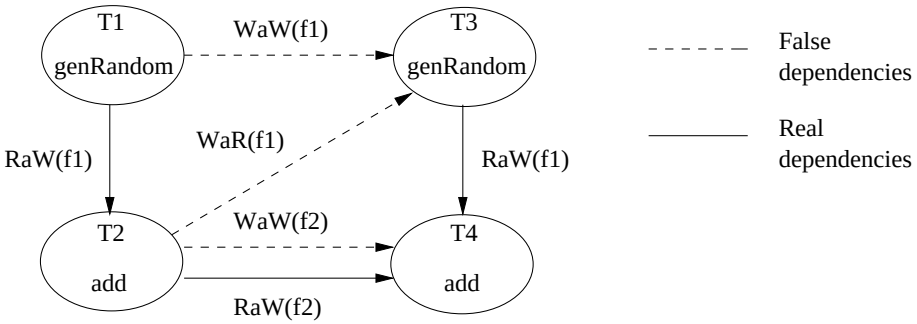

Figure 5. Task dependency graph of the Sum application

Tasks with no dependencies pass to the next step: the scheduling. According to the graph of Sum, the first suitable tasks are T1 and T3: they can be run in parallel on the Grid. When receiving these tasks from the TA, the TS decides, following a certain scheduling algorithm, the destination hosts where they will

be executed. Currently only a FIFO algorithm is implemented, though more complex ones will be added in the future to try, for instance, to reduce the execution and transfer times.

Once the scheduling is done, the TS communicates its decision to the JM. For a given task, the latter requests all the necessary file transfers to the FTM, which invokes the JavaGAT API to actually perform them. When all the input files of the task are in the destination host, the JM transforms the task into a GAT job, submits it to the Grid and subscribes to its state change notifications.

3.3.3 Task completion. Whenever a callback which informs about the end of a job is received, the JM notifies the TS of that fact. At its turn, the TS forwards the notification for the corresponding task to the TA. If the task has finished successfully, the TA removes the edges to all its successors from the graph and searches for newly dependency-free tasks to send for scheduling; otherwise, an error is thrown (see Section 3.3.6 for further details about error situations). For instance, in the case of Sum, T2 sees how all its dependencies are solved after T1 ends.

3.3.4 Opening a file. The Integrated Toolkit interface also offers a method to work with a file on the user's local machine. A call to `openFile` in a given point of the application makes the API perform the following actions: first, it registers the file access by invoking the FIP; second, if the open call is for reading or appending, it requests to the TA to be notified when the last writer task of the file ends; third, also for read and append modes, it makes the last version of the file be transferred to the local host of the user by contacting the FTM; finally, it returns the file name of this version, so that the application can open the necessary I/O streams.

3.3.5 End of the application. When the application reaches a `stopIT` call, the API makes it block until three events take place: first, the completion of all the tasks created until that moment; second, the transfer to the user's local host of all the result files, that is, the final version of each of the files accessed by the application; third, the deletion of all intermediate file versions, which will not be used anymore. The first event is notified by the TA, and the two last ones are triggered by the FTM.

Furthermore, the `stopIT` method allows to specify whether the Integrated Toolkit must finish definitely or not. In the first case, all the subcomponents are stopped, cleaned and killed, while in the second one they are just stopped, so that they can be restarted later and accept new task execution requests.

3.3.6 Error handling. During the execution of the application, the Integrated Toolkit runtime can experience errors of different kinds: a job submission

that has failed, a problem with a file transfer, an exception in some point of the code, etc. Unfortunately, managing an error produced inside the Integrated Toolkit while it is working is not a trivial issue. Since all its subcomponents are interconnected and communicate constantly, a failure in one of them could impede the overall system to work properly. The general response to such a situation should be to stop the components as quickly as possible; however, there are a couple of points that must be considered when facing an error.

On the one hand, the components that form the Integrated Toolkit cannot be stopped in any arbitrary order because they have data dependencies. A dependency between two components A and B appears when A invokes a synchronous method on B and waits for its result. The problem arises if B is stopped before it can serve the request from A; in that case, A would remain blocked waiting for the result of the call and it could never serve the stop control request[1].

If one invokes the stop method of the Integrated Toolkit life-cycle controller (`stopFc`, see [2]) the call is forwarded to all the hierarchy of components in an *a priori* unknown order. Nevertheless, the synchronous calls between subcomponents lead to the dependencies shown in Figure 6, and such dependencies impose a stop order that must be respected; otherwise, we could experience a deadlock. One solution could be to redefine the Integrated Toolkit life-cycle controller to ensure that the subcomponents are stopped in an adequate order, specifically the following one: TA, TS, JM, FTM, FIP.

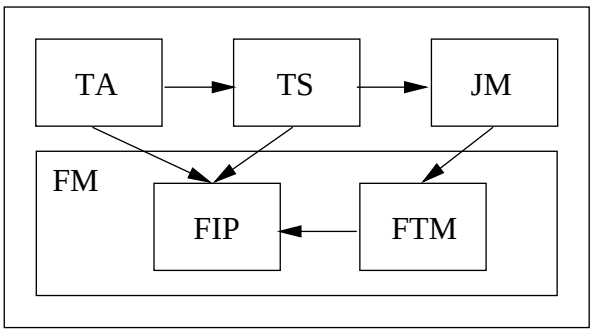

Figure 6. Data dependencies between Integrated Toolkit subcomponents

On the other hand, it is not enough to stop the subcomponents when an error appears, because that suspends their internal communications but does not finish the ongoing operations on the Grid. Consequently, a cleaning process must begin after stopping the subcomponents, and it includes two main actions: first, canceling the submitted jobs and unregistering for their state notifications;

[1] This theoretical behaviour has only been checked with ProActive components, therefore it might differ for other implementations of the Fractal/GCM model.

second, avoiding the beginning of new transfers (the JavaGAT API does not allow to cancel a transfer in progress[2]). This postprocessing can be done by means of a custom controller, contacted by the life-cycle controller when the component is stopped.

The methodology to face an error that has been explained above is probably the best one, but currently it cannot be applied efficiently in practice. To stop the components as fast as possible, the `stopFc` invocation should be placed in the head of each subcomponent request queue, so that it could be served next. However, ProActive does not allow to give higher priority to control requests for the time being; it is certainly possible to examine the whole queue each time a request is going to be served (searching for an eventual stop call), but that would clearly lead to a poor performance. Therefore, trying to stop the Integrated Toolkit while in operation would take a while, and during this time the system is doing useless work and could behave unpredictably.

The currently adopted solution aims to minimize the time between the error and the finalization of the system, while informing the user of what has exactly happened. Whenever an abnormal situation occurs, the component that detects it tells the Integrated Toolkit API about it. Then, the API invokes a multicast immediate service[3] that is forwarded to every component and that performs the necessary cleanup. After that, without actually stopping the components, a kill method is invoked on the Integrated Toolkit in order to destroy all the component structure. Lastly, the error message is returned to the user.

4. Preliminary tests

Our prototype is still in the test phase. Thus far, some preliminary tests have been performed. The objective of these tests was not to obtain performance measures, but to show that applications which benefit from the Integrated Toolkit maintain the same behaviour as if they were executed locally and sequentially. Concerning the GAT adaptors, we used the local, Globus Pre-WS and SSH ones for job submission, and the local, GridFTP and SSH ones for file transfer.

Following subsections describe some of the chosen applications and which functionalities of the Integrated Toolkit we wanted to test in each one.

4.1 Matrix multiplication

The *Matmul* application multiplies two matrices. It takes as input the matrices divided in blocks, which are themselves smaller matrices of doubles. Tasks

[2]The JavaGAT API methods to perform file transfers are synchronous and block the thread that invokes them until the transfer is finished.

[3]ProActive immediate services permit to run a method of a component server interface without having to wait in the request queue. The execution takes place immediately and in parallel with the normal services of the component.

work with blocks, which are stored in files. In our tests, we varied both the number of blocks of the input matrices and the number of elements in each block. More blocks implies more tasks, and larger blocks means tasks which are more coarse-grained.

We began with Matmul to perform a general and simple test of the Integrated Toolkit. The results showed that its main functionalities were performing well. The following points were checked: component deployment, start and stop; task creation, analysis and scheduling; file version management and transfer; job submission and monitoring.

4.2 Cholesky decomposition

The *Cholesky* application decomposes a symmetric positive-definite matrix (A) into a lower triangular matrix (L) and its transpose (U). As Matmul, all matrices are divided in blocks, which are taken by tasks as their unit of work.

With Cholesky we wanted to take the task analysis and scheduling tests a step further. Concerning the analysis, Cholesky generates a highly connected dependency graph, which represents a much more challenging test for the TA. Regarding the scheduling, there are five types of task (that is, five different methods to execute remotely) on which to impose particular constraints, and that allows to check if they are actually scheduled on the resource/s whose capabilities match their constraints.

The results were satisfactory, demonstrating that the Integrated Toolkit is able to manage applications with complex dependencies and to schedule their tasks respecting the required constraints.

4.3 Counter increment

The Counter application performs several increments on the integer value contained in a file. Some of the increments are spawned as remote tasks, and some of them are executed locally.

The objective of this application was to test the `openFile` method, since it must be called before a local increment in order to get the right file version. The method was invoked alternatively with the write-only access mode to replace the value of the counter and with the read-write one to increment it; in the case of the latter, the file needs to be transferred to the user's local host, while with the former it is not necessary thanks to the renaming technique.

The results showed that the counter was properly incremented both in the local and remote ways, and its final value was the expected one.

5. Related work

Other approaches that enable the programming of parallel applications for computational Grids are Satin, HOCs and ASSIST. Satin [6] is a Java based

programming model for the Grid which allows to explicitly express divide-and-conquer parallelism. It uses marker interfaces to indicate that certain method invocations need to be considered for potentially parallel (spawned) execution. Moreover, synchronization is also explicitly marked to wait for the results of an invocation. HOCs [7] is a component-oriented approach based on a master-worker schema. Higher-Order Components (HOCs) express recurring patterns of parallelism that are provided to the user as program building blocks, pre-packaged with distributed implementations. ASSIST [8] is a programming environment aimed at providing parallel programmers with user-friendly, efficient, portable, fast ways of implementing parallel applications. It includes a skeleton based parallel programming language and a set of compiling tools and runtime libraries.

Besides, there exist several systems that permit workflow definition and execution on Grids, for instance P-GRADE and SEGL. P-GRADE [9] is a general purpose, workflow-oriented computational Grid portal. It offers a high-level, graphical workflow development system and an execution environment for various Grids. SEGL [10] allows to define complex workflows which can be executed in a Grid environment, and supports the dynamic generation of parameter sets. It also makes possible the execution of sets of independent tasks of interdependent jobs which can turn either synchronously or asynchronously on heterogeneous systems.

6. Conclusions and future work

We have proposed a componentised design of the Integrated Toolkit, a framework which facilitates the development of Grid-unaware applications and which can also provide Grid-aware ones with some functionalities. After that, we have presented a first implementation of the Integrated Toolkit through an example, explaining how to write a simple application that uses the Integrated Toolkit and which internal processes are triggered in order to execute it. Finally, we have shown that the Integrated Toolkit prototype has been able to run several sample applications on the Grid.

Furthermore, thanks to the componentised nature of the design presented in Section 2, we believe that the Integrated Toolkit could also offer an alternative to develop Grid-aware applications, which could use the runtime as a whole or deploy solely specific subcomponents. For instance, a programmer interested in adding a scheduling functionality to an application could choose to deploy only the TS subcomponent, binding its interfaces to the ones of the application components.

Forthcoming phases of this project will:

- Extend and improve the functionalities of the Integrated Toolkit subcomponents. Some of the envisaged features are: fault-tolerance mechanisms

for job submission and file transfer, new scheduling algorithms, checkpointing of tasks to avoid resuming the application from scratch in case of failure, dependency analysis which takes into account different data types (not only files but also scalars and arrays), identification of the critical path in the task dependency graph, and so on.

- Study the possible bottlenecks in our component design. Synchronous calls cause waiting times that could be partly avoided if the data dependencies are minimized. Moreover, too frequent communications between components could also degrade the performance of the system.
- Implement controllers to steer the behaviour of the Integrated Toolkit. These controllers could serve to modify certain parameters (such as the scheduling algorithm used), change the overall structure (add/remove/ bind/unbind components), manage the persistence of the application (in relation to the checkpointing and fault recovery mechanism), etc.

References

[1] Proposals for a Grid Component Model, CoreGRID Deliverable D.PM.02, 2006.

[2] Fractal specification, http://fractal.objectweb.org/specification/index.html

[3] R. M. Badia, Jesús Labarta, Raúl Sirvent, Josep M. Pérez, José M. Cela and Rogeli Grima. *Programming Grid Applications with GRID superscalar*. Journal of GRID Computing, Vol. 1 Issue 2. Pages: 151-170, June 2003.

[4] ProActive, http://www-sop.inria.fr/oasis/proactive/

[5] Grid Application Toolkit, http://www.gridlab.org/gat/

[6] Rob van Nieuwpoort, Jason Maassen, Thilo Kielmann and Henri E. Bal. *Satin: Simple and Efficient Java-based Grid Programming*. Scalable Computing: Practice and Experience, 6(3):19-32, September 2005.

[7] Sergei Gorlatch and Jan Dunnweber. *From Grid Middleware to Grid Applications: Bridging the Gap with HOCs*. In Future Generation Grids, Springer Verlag, 2005.

[8] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Marco Vanneschi and Corrado Zoccolo. *ASSIST as a Research Framework for High-performance Grid Programming Environments*. In Jose C. Cunha and Omer F. Rana, editors, Grid Computing: Software environments and Tools. Springer-Verlag, 2004.

[9] P-GRADE portal, http://www.lpds.sztaki.hu/pgportal/

[10] Natalia Currle-Linde, Uwe Kuester, Michael M. Resch, Benedetto Risio. *Science Experimental Grid Laboratory (SEGL) Dynamical Parameter Study in Distributed Systems*. In proceedings of the 2005 International Conference on Parallel Computing (ParCo 2005), pp 49-56, Malaga, Spain.

[11] Java annotations, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[12] Design of the Integrated Toolkit with Supporting Mediator Components, CoreGRID Deliverable D.STE.05, 2006.