Early Application Experience with the Grid Application Toolkit (GAT)

Stevens Le Blond, Ana-Maria Oprescu, Chen Zhang
Vrije Universiteit, Amsterdam, The Netherlands

{slblond,aoprescu,czhang}@few.vu.nl

Abstract

While diversity plays an important role in evolution and progress, it is also what makes writing Grid applications to be generally considered a challenging task. The Grid Application Toolkit (GAT) is promising to simplify this task through a plug-and-play design, which is meant to hide all diversity related application writing issues from a Grid-unaware application developer. In this paper, we report our experiences implementing and deploying a parallel MPEG encoder program as a task farming application using the GAT. While writing a real-world application following a programming model which is not Grid-specific, we identify those aspects that are made convenient by the GAT, and those that still remain cumbersome or difficult. Our main observation is that the GAT indeed simplifies application writing. However, additional services like resource brokerage are needed for similarly simplified application deployment. Though not acute, there are also some issues related to incomplete semantics.

1 Introduction

Writing Grid applications is generally considered a challenging task. The Grid Application Toolkit (GAT) is promising to simplify this task. In this paper, we report our experiences implementing and deploying a parallel MPEG encoder program as a task farming application using the GAT. We identify those aspects that are made convenient by the GAT, as compared to distributed technologies, such as RMI, and those that still remain cumbersome or difficult. Our main observation is that the GAT indeed simplifies application writing. However, additional services like resource brokerage transparent to application programmer are needed for similarly simplified application deployment. We also discuss several semantic issues of the current version of (Java-)GAT.

1.1 Paper organization

Section 2 presents shortly the real-life application we chose to implement as a Grid programming experience. It shortly introduces the reader to the history, development and critical issues of MPEG encoders. We add here a set of remarks from previous experience of implementing the MPEG encoder with RMI, and introduce GAT as a more appropriate candidate. We then continue in section 3 to a small preamble on JavaGAT, where we present the part of the JavaGAT API that has been of interest for our implementation. Section 4 details our MPEG encoder implementation as based on Joinc. A thorough description of our experience with JavaGAT is then given in section 5.

2 A Parallel MPEG Encoder

The Moving Picture Experts Group (MPEG) is a working group of ISO/IEC charged with the development of video and audio encoding standards [13]. MPEG has standardized a family of video and audio compression standards for multimedia applications [6], such as MPEG-1, MPEG-2 and MPEG-4. In fact, MPEG standardizes only the bitstream format and the decoder while giving out the liberty to encoder implementations as long as they produce bitstreams conforming to the specified bitstream format. MPEG files are much smaller for the same quality compared to other video and audio coding formats [15]. MPEG is frequently used for video/audio protocols, partly due to its ability to handle multimedia over varying bandwidth conditions [16].

As MPEG formats are being widely used, MPEG encoding however remains costly in terms of performance. The encoding process normally requires storage and computational power far beyond what traditional home computers can provide with satisfactory performance. To illustrate, in order to encode a raw DV file, a PC may require 10 GB or more disk space and may take hours to complete. There is obviously a huge gap between the end users delight and the poor performance of encoders on a single computer. This gap has led to considerable amount of research and software development for providing an effective way of doing high performance MPEG encoding. A prevalent solution to this goal is to have a parallelized MPEG encoder working over a pool of distributed multiprocessors, especially over Grid [5]. Currently, there are many different approaches in designing such a parallel MPEG encoder with either fine grain or coarse grain parallelism [8]. A typical design example is based on Master/Slave model, under which the master dispatches jobs to slaves for data processing and assembles results together in the end. Although the idea of parallel MPEG encoder is straightforward, a number of intricacies lie can be found in the design, implementation and deployment processes. Even for advanced developers it remains a very challenging task to implement a parallel MPEG encoder using distributed resources while using only common programming tools and environments.
Moreover, the trend of parallelizing computational expensive legacy software reaches far beyond the scope of parallel MPEG encoder only. We need a more common and friendly environment for parallel programming to alleviate complexity and promote ubiquity. GAT is one candidate for such an environment that provides much simplicity. Our experience in developing an MPEG4 encoder using GAT reveals that it makes parallel programming less challenging in Grid environment, and demonstrates the feasibility to simplify application of complex parallel programming techniques in developing daily user applications.

Another candidate environments we considered using is RMI. The following remarks about our RMI implementation efforts should not be seen as RMI shortcomings. RMI is a very powerful scheme for distributed programming and this has been the goal of its design team [19]. Our point is that Grid application programmers would benefit more from an abstraction layer, such as the one provided by JavaGAT. We tried to implement a customized task farming approach with RMI. The implementation design was quite simple, as explained in section 2, but the actual code was very cumbersome. We needed to explicitly write the pre-staging and post-staging of applications files, as well as to manage submission of tasks over a number of operators that are deployed outside of the application code on each working site.

Though discovery of deployed operators is possible through environment settings, the step of manually deploying each remote object that represents an operator must be explicitly dealt with. While scheduling the jobs is still a problem when using JavaGAT, the actual submission of a job and the setting of pre-staged and post-staged files for that particular

job are wrapped inside API calls. This is not the case with RMI, where explicit code needs to be written in order to copy the needed files at the working sites. Also related to task submission we would have needed to devise our own call-back mechanism to keep track of the state of submitted jobs.

Another issue related to RMI surfaces from the setup of policy files on all participating clusters, as RMI requires a Security-Manager to be explicitly set up in the remote objects JVM. This would be comparable to credentials needed by the ResourceBroker of JavaGAT, though the policy files are tied to a rather unfriendly syntax. One could also choose Policy objects rather than files, but then the problems are moved in the code writing section. On the other hand, using JavaGAT and simple certificate credentials, one simple script prepares the environment for the whole application to run during a user-specified amount of time.

3 The (Java-)GAT

3.1 Context

Computing Grids are getting more and more important. While this fame is pushing people to always come up with smarter functionalities, it also creates interoperability problems. This is maybe the reason why, today, only few complex Grid application are widely used.

Nowadays, applications written for a given Grid often have to be heavily modified or even rewritten to be ported on another one. One issue here is that Grid developers have to know the intricate details of all the resource manager systems (RMS) in order to write portable applications. A second is that the same application cannot run on different RMS spanning multiple clusters even if their administrations trust one another. It implies that one cannot simply, blindly, download a Grid application programmed for a given RMS and execute it on another – it seems that Grid applications are just like desktop applications 20 years back from now.

To circumvent the situation, some projects have worked on integration facilities for their own system [14, 9], while others tried to define more general interface on top of which applications can be written independent of the underlying RMS [3]. While being an important step forward, two issues still remain. First if multiple solutions are commonly adopted, even if alleviated, the interoperability problems will persist. Additionally, none of this solution intend to simplify life of the programmers and their APIs still remain complicated to learn and use.

This is where GAT (the Grid Application Toolkit) steps in. By providing a generalized and intuitive interface, GAT will allow the developers to do not even have to know which RMS is present on the underlying Grid. Even better, GAT will choose at runtime the correct way to access resources.

In this section we will briefly present the concepts behind GAT. We will first introduce the big picture of the GAT organization, finally we will briefly discuss the subset of the JavaGAT API we used to program our MPEG encoder.

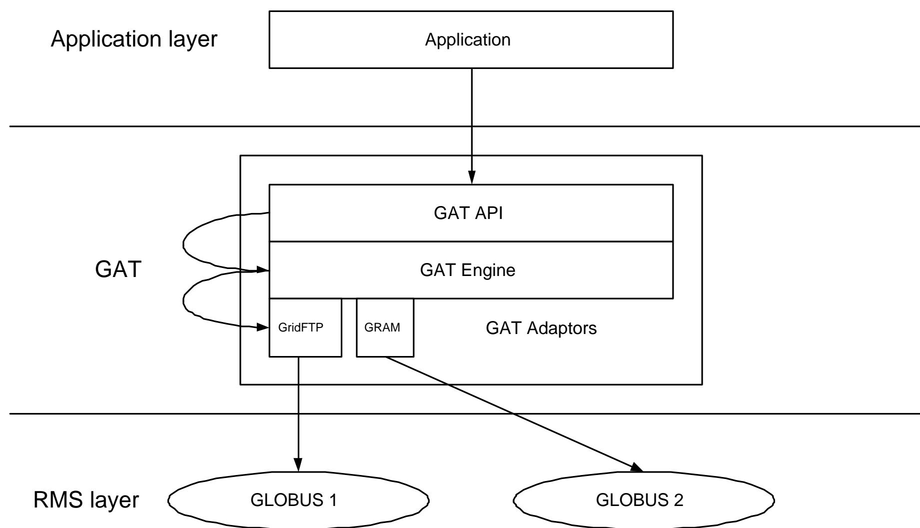

Figure 1: The GAT framework

3.2 GAT Organization

A programmer does not have to know the GATs internals to write an application on top of it. However, we firmly believe knowing a bit about GATs design helps to understand its usefulness. The interested reader is invited to refer to [1] for further details.

From figure 1 we can see that the GAT layer is split into 3 logical components. The API is the only piece of GAT the programmer has to know and to deal with when writing Grid applications. Its function is bound to the strict minimum – it provides the application with calls for essential Grid operations in a simple and stable way. The GAT engine will decouple applications from the always changing middleware by loading adaptors when needed. The engine is supposed to be very thin, transparently providing an efficient way to link the logic of Grid services with the GATs API.

Finally, the engine chooses adaptors on demand in order to satisfy the capabilities defined by the API and required by the application (job submission, file transfer etc.).

3.3 JavaGAT API – The Employed Subset

3.3.1 Files

In the context of task farming, we usually have to deal with pre-staged and post-staged files. The former is the set of files the task needs access to in order to run properly. Examples of such files would be required libraries, file to be read by the task during execution, or the tasks code. Post staged files are the set of files which the task produces, and which the master application needs to retrieve after the task finished.

In order to easily deal with both pre-staged and post-staged files, we used the `File` interface from the package `org.gridlab.gat.io`. Creating an `org.gridlab.gat.io.File` object is similar to the mechanisms known from `java.io.File`, but will provide the same services for a wider variety of file access protocols. For instance, the file could now be anywhere, as long as the location can be described by an `org.gridlab.gat.URI`, and at least one of the adaptors is able to access it. The Gat Engine will make sure the appropriate adaptors have been loaded and an instance of these adaptors is representing the file.

The code listed below shows how to obtain a `org.gridlab.gat.io.File` object for a file, given its name as a `java.lang.String`. It first builds an instance of the `GATContext` (`org.gridlab.gat.GATContext`), which is the context in which the application will run. Optionally, an instance of `org.gridlab.gat.Preferences` may be obtained, which would be responsible for specifying user preferences when selecting adaptors. If there are no specified preferences, or the `createFile` method is invoked with a `null` value for the `Preferences`, the default policy for selecting adaptors is used (note: specifying "any" in the scheme part of the Files URI will enable the Gat Engine to dynamically determine which adaptor(s) should be used).

```
String      fileName = "file";
URI         fileURI  = new URI ("any:///" + fileName);
GATContext  context  = new GATContext  ();
Preferences prefs    = new Preferences ();
File        file     = GAT.createFile  (context, prefs, fileURI);
```

3.3.2 Jobs

As described at the beginning of this subsection, for task farming we also need file retrieval mechanisms. Using the `org.gridlab.gat.resources.SoftwareDescription`, the pre-staging and post-staging may be expressed with only two lines of code, (assuming the `org.gridlab.gat.io.File` `preStagedFileList[]/postStagedFileList[]` have already been populated):

```
SoftwareDescription sd = new SoftwareDescription();
sd.setPreStaged  (preStagedFileList);
sd.setPostStaged (postStagedFileList);
```

If the applications tasks are designed to use a certain file as `standard input` and to redirect the `standard output` and `standard error` to some file(s), then one could use the next few lines of code:

```
sd.setStdin  (GAT.createFile (context, prefs, new URI (stdin )));
sd.setStdout (GAT.createFile (context, prefs, new URI (stdout)));
sd.setStderr (GAT.createFile (context, prefs, new URI (stderr)));
```

Finally and most important, the line of code which indicates the location of the tasks executable:

```
sd.setLocation (new URI (executableFilePath))
```

After this setup phase, the task, represented by the `SoftwareDescription` object, is ready to be submitted for execution (we refer the reader to section 5.3 for more details about job submission in JavaGAT).

4 Implementing the MPEG Encoder

4.1 GAT

As specified in section 2, we chose to implement the MPEG encoder using a task farming approach. Following the ideas provided in the previous section, we decided to use JavaGAT to make its programming and portability easier.

Inspired by the generalization of task farming applications like SETI@home [10] and Boinc [2], the JavaGAT developers decided to provide a standard similar interface so that the programmer doesn't have to know about the distribution process. That layer on top of JavaGAT is named Joinc.

4.1.1 Joinc

In the Joinc programming model, task farming applications are easily expressed by using two classes, *Master* and *Task*. The *Master* class is designed to be extended by the master object of the application, hence its abstract methods will be implemented by the application. *Task* exhaustively describes each job and its dependencies so that it can be properly executed on a remote machine.

The Master class contains a half dozen of abstract methods which will have to be implemented by the application programmer. A call to *getTask* should return a Task object with a unique task identifier, the file names of the stdin, stderr and stdout, the pre and post-staged files, the name of the class containing the main method of the task, and finally, the parameters the task accepts. A call to this method is done right before submitting a task.

totalTasks will return the number of tasks the application will produce and *maximumMachines* the number of machines Joinc can simultaneously use to run the tasks. These functions are called at the very beginning of the *start* method to initialized the behavior of the task distribution.

taskDone will notify the application of a task termination, and *idle* permits to the programmer to do what ever he wants when there is nothing better to do.

To conclude, the application calls the *start* method at the very beginning of the execution, then Joinc takes the control over the application details – the application doesn't have to deal with GAT at any moment. It is time to see how we can use this nice interface to write an application and, more precisely, a parallel MPEG encoder.

4.1.2 The Application

Using figure 2 we will now show step by step how Joinc can be used to parallelize a MPEG encoder. The remainder of this section will provide the reader with some insights about parallel MPEG encoding while illustrating how to use Joinc to write applications.

The very first thing the encoder does at step 1 is to execute the tool *avisplit* to divide the raw AVI files into smaller chunks which can be shipped to the workers. Note that *avisplit* will create independent chunks, that means that once the tasks are dispatched among the workers, they will compute without having to care about interleaving frames, or any other communication. The encoder will keep track of the generated chunks names and associate one chunk with one task.

Then the *start* method is called by the application to let Joinc take control. Joinc subsequently calls some initialization methods to learn about the encoder settings. At the end of step 2, Joinc knows the number of tasks it has to submit as well as the number of machines it is allowed to simultaneously submit tasks to.

Joinc then has to retrieve the next task object from the encoder through the *getTask* method call (step 3). This method will initialize the fields of the *Task* object, so that each

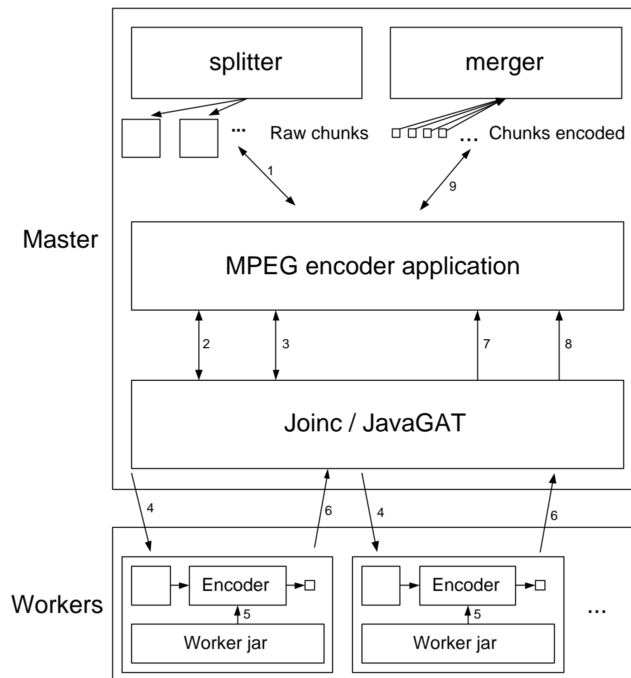

Figure 2: The MPEG encoder organization

worker encodes a part of the raw file. To do so, Joinc initializes a *SoftwareDescripton* per task so that GAT can transparently handle file transfers and job submission at step 4.

Step 5 corresponds to the execution of the worker code by the remote Java virtual machine. One advantage of Joinc over Boinc is its ability to send a worker jar over the network instead of having to install the binaries at all sites beforehand. Additionally to these flexibility issues it is also important to note that this constitutes a method to distribute any kind of computation in a portable way. In our example, the minimalist worker will just execute the tool *mencoder* [12], which will encode the chunk – this is the computing intensive task we want to parallelize. Once done, the now compressed movie chunk is shipped back to the master (step 6), and Joinc calls the *taskDone* method (step 7) for this task.

After all the tasks complete, Joinc will simply return control to the application (step 8). This is time to execute the tool *avimerge*, which merges all the chunks into the compressed movie (step 9).

5 The Good, The Bad And The Ugly

In this last section we will sum up what the experience of using JavaGAT taught us. While being a generally positive experience, we also encountered a number of problems or noticed a lack of middleware facilities when trying to implement more advanced features in our encoder. We first describe what the use of GAT bought us, and will list those parts we feel are general enough to be part of GAT, and we then will end with technical issues that should be improved to enhance the programming experience with GAT.

5.1 The Good

As we have seen in the examples presented in section 3.3, Grid programming becomes easier when assisted by the JavaGAT API. The details of different approaches to achieve a given operation on a given object are well hidden inside the adaptors implementing the respective object interfaces. The GAT engine will dynamically load a set of appropriate adaptors and delegate the requested operation to them. Though not required, the user is able to specify which adapter should be used for a given object, through the `preferences` passed to the GAT engine at instantiation time, as shown below.

```
Preferences prefs = new Preferences ();
preferences.put ("file.adaptor.name", "gridftp");
File file = GAT.createFile (context, prefs, fileURI);
```

There is a high degree of flexibility, without diminishing the highest achievable degree of transparency – more about the limitations on transparency can be found in section 5.3. The benefits of hiding details are best shown in the small code excerpt given below, with the use of `context` and `preferences` as discussed in section 3.3.1. The intricacies of creating an object that would communicate to a RMS are kept from the Grid developer, which only has to create an object with the `org.gridlab.gat.resources.ResourceBroker` interface, an action comparable in coding effort with the creation of a `org.gridlab.gat.io.File` object.

```
GATContext     context = new GATContext ();
Preferences    prefs   = null;
ResourceBroker broker = GAT.createResourceBroker (context, prefs);
```

Another good feature of the JavaGAT design is the availability of different levels of error messages. While for adaptor writers it is recommended to use the `gat.debug` system property when testing their programs, the rest of the users should consider the use of `gat.verbose` system property for high level debug purposes.

5.2 The Bad

To us, the functionality in GAT we missed most is its lack of brokerage awareness. Admitting the fact that Grids will get larger and larger, we think it is a mistake to let the programmer deal with the brokerage issues. The same remark applies to scheduling, where some very basic default scheduling policies, like FIFO, would often provide enough functionality for simple load balancing.

At the moment, the only way for the GAT user to have its application automatically scheduled on multiple clusters is to use the GRMS adaptor [18]. While GRMS provides an optimal transparency to the programmer, its not fully stable yet and still under development. The cost for this transparency will then come with poor performances to schedule jobs over an entire Grid – the issue is to know if using GRMS does not cost more than it buys by taking smart scheduling decisions.

We believe a tradeoff could lie in a distributed flavor of GRMS, possibly making decisions based on partial knowledge. We could for instance imagine to have a simple brokerage adaptor running at each GAT application and take decisions based on samples received from an external services like [11] – please note that this scheme would be quite similar to the ‘Smart Sources’ mechanism used in peer-to-peer systems to prevent the ‘best’ nodes to be flooded with requests when using a centralized information system component.

Finally, making GAT aware of its environment would improve its transparency, e.g. by letting it negotiate with clusters and determine the resource management system they are running. We saw in section 3 that we have to specify in advance which cluster is using which system. This substantially complicates the programming when having to submit jobs over heterogeneous resource management systems. Having GAT performing service discovery would, once again, help the programmer to write portable Grid applications.

5.3 The Ugly

As promised in section 5.1, we will try to list some of the limitations of the provided transparency we stumbled upon while developing our MPEG encoder application. One issue is related to credentials: As the application is Grid-empowered, the need of valid credentials is critical. Still, validating the credentials is bothersome action and automating the process would help. We tried to figure out where the problem actually lies, and it turns out it is not the responsibility of JavaGAT, nor of Globus credentials management system. In this case, the achievable degree of transparency is dependent on the Certificate Authority rules for certificate protection. In our case, that CA is run by DutchGrid [4], and does not allow a certificates pass-phrase to be empty. This is not the case with SSH authorization system, which is based on private/public key protocol and which allows for pass-phrases to be empty, hence allowing the login process to be fully automated. As a conclusion, the limitation on transparency where credentials are involved is strongly related to credentials specific protection mechanisms, which are under the strict control of the security component governing the given Grid.

Another issue is related to the main topic of section 5.2: In the current state of JavaGAT where ResourceBroker adaptors are concerned, the Grid developer which would like to have the application run on the real Grid and not on the local machine, needs to use the following code excerpt, which may not be very intuitive:

```
GATContext      context = new GATContext  ();
Preferences     prefs   = new Preferences ();
preferences.put ("ResourceBroker.adaptor.name", "globus");
preferences.put ("ResourceBroker.jobmanager",   "pbs");
ResourceBroker  broker  = GAT.createResourceBroker (context, prefs);
```

While it is true that the overload of dynamically deciding whether the application should run on the Grid or on the local machine is not worth the transparency gain, a compromise could still be made. The user should be able to indicate that the application needs to be gridified in a more abstract fashion, e.g. by using:

```
GATContext      context = new GATContext  ();
Preferences     prefs   = new Preferences ();
preferences.put ("ResourceBroker.type", "remote");
ResourceBroker  broker  = GAT.createResourceBroker (context, prefs);
```

In JavaGAT, in order to create the incarnation of a Joinc task (`Job`), the user has not only to provide the description of the software representing the task, but also a description of possible run-time environments for this task. This is achieved by specifying a `org.gridlab.gat.resources.ResourceDescription` (a set of software and hardware runtime requirements), or a `org.gridlab.gat.resources.Resource` (representing a specific resource). Once the `JobDescription` object has been created it can be used to obtain as many `Job`s for the task as the user would like to have. The `Job` object is returned by the `broker` after successfully submitting a job that meets the `JobDescription` specifications.

```
Map attribs = new Map ();
attribs.put ("machine.node", "fs0.das2.cs.vu.nl");
ResourceDescription rd = new HardwareResourceDescription (attribs);
Job j = broker.submitJob (new JobDescription (sd, rd));
```

Finally, a rather technical issue was raised by the `gridlab.gat.resources.Job.getJobID` method. Though the return value is a *"globally unique identifier for the physical job corresponding to this instance"* [7], it is no longer available after the `Job` object state changed to something different than `Running` or `Submitted`. When many jobs need to be monitored, as is the case of our MPEG encoder implementation, the globally unique identifier might prove a better way to select finished jobs from a list of all submitted jobs, rather than using the jobs object reference. Again, this is a technical issue, but following the definition of a globally unique identifier as given in [17], one might wonder why the job IDs cannot be retrieved from the job object in *any* given state of the job.

6 Conclusion and future work

In this contribution we discussed our experience writing a Grid-parallel MPEG encoder using the Java version of GAT. After having briefly introduced what an MPEG encoder is and in which context it can be distributed, we talked about the general issues GAT is supposed to solve, and further explained how GATs design team chose to address these issues. We finally came to the main point of this paper by showing to what extend GAT simplifies application development.

We then presented the approach we chose to parallelize and implement the MPEG encoder. By introducing an additional layer between the application and GAT we showed that it is possible to completely hide the distribution from the user, while at the same time making Grid programming portable and straightforward. As an example we gave some insights about a previous experience we had with using RMI for parallelizing the same application. However, distributed programming schemes have their own applicability and our point is not that RMI should be generally replaced by GAT, but merely that Grid programming without Grid-awareness is better served by GAT.

The overall conclusion after working with JavaGAT is optimistic. Though there are bad and ugly things to be considered, neither of them seems to be insurmountable. In our opinion, it is just a matter of further development and real-world user feed-back to make the JavaGAT a truly transparent and flexible Grid application development toolkit.

We are now planning to port our MPEG encoder application on the worlds first "virtual city supercomputer", the *Almere Grid* testbed in the Netherlands. While still only few information are available about the project, there would already be more than 2.000 machines connected through the 100 Mbit/s fiber network built across the city. This would be an incomparable environment to introduce one of the very first Grid application useful for users.

7 Acknowledgments

First, we would like to thank Prof. Henri Bal for having taught us parallel programming and the suggestion to parallelize a real world application – a MPEG encoder. We also want to express our gratitude to Jason Maassen for having been an impassioned Grid computing assistant through our first masters year and the initiator of Joinc. Special thanks go to Rob van Nieuwpoort for being the "back office" guy of JavaGAT. Finally, this paper would have never been written without the long lasting support of Thilo Kielmann and Andre Merzky.

References

[1] 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.

[2] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, 2004.

[3] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Lecture Notes in Computer Science*, volume 1459, page 62, Jan 1998.

[4] `http://www.dutchgrid.nl`.

[5] I. Foster. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 2nd edition edition, 2004.

[6] D. L. Gall. MPEG: A Video Compression Standard for Multimedia Applications. *Communications of the ACM*, 34(4), 1991.

[7] `http://www.cs.vu.nl/~robn/gatdocs/`.

[8] S. Y. I. Assayad, V. Bertin. Parallel Model Analysis and Implementation for MPEG-4 Encoder. In *Proceedings of Embedded Processors for Multimedia and Communications II*, San Jose, CA, USA, 2005.

[9] D. B. Jackson. *Grid Scheduling with Maui/Silver*, chapter 11. Kluwer, 2003.

[10] E. Korpela, D. Werthimer, D. Anderson, J. Cobb, and M. Leboisky. SETI@home-massively distributed computing for SETI. In *Computing in Science & Engineering* `[see also IEEE Computational Science and Engineering]`, volume 3, Jan 2001.

[11] J. Maassen, R. V. van Nieuwpoort, T. Kielmann, and K. Verstoep. Middleware Adaptation with the Delphoi Service. In *Workshop on Adaptive Grid Middleware (AGridM 2004)*, Juan-les-Pins, France, 2004.

[12] `http://www.mplayerhq.hu/homepage/design7/news.html`.

[13] `http://en.wikipedia.org/wiki/MPEG`.

[14] B. Nitzberg, J. M. Schopf, and J. P. Jones. *PBS Pro: Grid Computing and Scheduling Attributes*, chapter 13. Kluwer, 2003.

[15] R. Pulles and P. Sasno. A set top box combining MHP and MPEG-4 interactivity. In *Proceedings of the 2nd European Union symposium on Ambient intelligence*, Eindhoven, Netherlands, 2004.

[16] Rob Koenen, ed. Overview of the MPEG-4 Standard, March 2002. [Online]. Available: `http://www.chiariglione.org/mpeg/standards/mpeg-4/mpeg-4.htm`.

[17] A. Tanenbaum and M. van Steen. *Distributed Systems, Principles and Paradigms*. Prentice Hall, 2002.

[18] The GridLab Project. (2003) The GridLab Resource Management System. [Online]. Available: `http://www.gridlab.org/grms/`.

[19] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing, Nov 1994.