Towards better optimization of resource allocation

Zoltán Horváth, Tamás Kozsik, Attila Ulbert
Department of General Computer Science
Eötvös Loránd University, Budapest, Hungary
hz@inf.elte.hu, kto@elte.hu, mormota@elte.hu

January 7, 2003

1 Introduction

During our visit in CERN we have designed some extensions to the existing grid middleware and we have prepared prototypes to some of our ideas, which will be used as an initial step in a more complete implementation. The aim of our visit was to get acquinted to the state-of-the-art of the EDG project and find those research areas where collaboration between the EDG team and our research group would be beneficial to both. Thanks to the valuable help of and the continueous discussions with a few members of the EDG team these goals were fulfilled. This report summarizes the results of our visit. Section 2 introduces the problems that we would like to solve and gives some motivation. Section 3 describes the design and the prototype implementation of our solutions to the afore-mentioned problems. Section 4 explains what should be done in the near future to obtain a powerful extension to the existing grid middleware. Finally, Section 5 draws the conclusions.

2 Problem specification

Applications on the grid need different types of resources. Such resources are CPU, memory, network bandwidth and secondary storage. The grid middleware tries to allocate resources to applications in a way that results can be computed as fast and as efficiently as possible.

Our main focus is on the secondary storage, on data access. Many applications read and write large files. Often these files are not available on the computing element where the application runs. An optimal strategy for data access may depend on the kind of the application. An application may perform heavy computations spending long periods of time between successive access of data. On the other hand data-intensive applications may need to access huge files frequently. Data transfer may be optimal or not in parallel to the computation. A small puffer may be enough for storing a local copy of the data needed in the next step of computation or a complete copy of a huge file may be neccessary in advance. The granularity of the read/write operations may be very diffrent, reading/writing small or big amount of data in one single operation. If an application uses several input and output files, then the access method my vary. An optimal data access may require that some files should be copied as a whole, while other files are accessed in parts using a novel version of gridfs (developed by A. Nagy).

We introduce the concept of *data access pattern*: which files are accessed by the application, by read/write/append operations, amount of data needed in one step, frequency of data access operations, accessing data in a regular (e.g. sequential) or irregular order.

An optimal data access strategy can be determined only if the Resource Broker and the Replica Manager gets sufficient amount and quality of information about the validated data access pattern of the application.

3 Design and prototype implementations

Defining the data access patterns for an application can be used to optimize resource allocation by providing valuable information to the Resource Broker and the Replica Manager. The definition of data access patterns can be acomplished in different ways. They can either be specified by the programmer or be derived from data collected by a monitoring tool. In the next subsections we will address both approaches.

3.1 Towards a JDL extension

In order to better optimize job scheduling and file access the JDL must be extended. The extended JDL descripion should be able to specify the amount of computing and secondary storage resources required by a job. Moreover, this description should enable the scheduling service to calculate the required network bandwidth. The descripion should be able to specify the secondary storage access method as well, allowing the Replica Manager (or the future EDG Grid FS implementation) to optimize file access parallel to the job execution.

The JDL extension is based on our simplified job model. According to this model, the jobs consist of processing steps. During a processing step, the job opens its input files, reads the input data, completes operations on the input data, and writes the result of processing operations. The JDL extension will describe the resource (computing and secondary storage) consumption of the processing steps.

3.1.1 Input files

The input file access description is block-based, i.e. it describes the blocks that will be read by the job, and specifies the read method:

```
input "<file name>" -> (offset, length, stride, ratio, method)
```

The description requires the following parameters to be set:

input identifies the input file access descriptions

<file name> the name of the input file

offset the beginning of the first block to be read

length the length of the blocks

stride the distance between two blocks

ratio the ratio of the actual amount of data that will be read and the block length ($\in (0, \inf)$)

method the way the job will read the input blocks, specified by three keywords: `seq` in case of sequential processing, `step(n)` in case of sequential processing having `n` bytes gap between two processed data, `rand` in case of undefined file access.

An example description:

```
input "measurements.dat" -> (0, dom("measurements.dat"), 0, 1, seq)
```

The example description specifies a simple input file access: the job sequentially reads the whole `"measurements.dat"` file. (The `dom` function returns the length of the file in bytes.)

Analysing this description, the Replica Manager and the Grid FS can decide on not to replicate the whole input file on the CE that runs the job, but pre-fetch the beginning of the file and read the rest into a relatively small buffer parallel with the job execution. This strategy can minimize the time from job submission to job termination.

3.1.2 Output files

The output file access description is also block-based. However, different write methods require different block parameters:

```
output "<file name>" -> (offset, length, stride, ratio, modify)
output "<file name>" -> (length, append)
```

The output file access descriptions are identified by the `output` keyword, <file name> specifies the name of the output file to be written.

In case of `modify` write method, the `offset`, `length`, `stride` and `ratio` parameters have the same semantics as in case of input descriptions.

In case of `append` method, only the `length` has to be supplied, which specifies the amount of data to be written in bytes.
An example description:

```
output "sum.dat" -> (4, append)
```

According to the example description, the results will be appended to the output file `"sum.dat"`. The approximated amount of data to be written is 4 bytes in this case.

3.1.3 Operations

The processing operations are described by their fixed and floating point operations:

```
operation -> (fixed, floating, file1:amount1, file2:amount2, ...)
```

The description requires the following parameters to be set:

operation identifies an operation description

fixed is the number of fixed point basic operations completed by the operation in one processing step, e.g. fixed point `+`

floating is the number of floating point basic operations completed by the operation in one processing step, e.g. $\sqrt{x}$

file:amount the input `file` and the `amount` of data that will be processed by the operation in one step (multiple `file:amount` descriptions are allowed)

An example description:

```
operation -> (10, 0, "measurements.dat":40)
```

According to the example description, the operation reads the `"measurements.dat"` input file. In each processing step 10 fixed point basic operations are completed, and 40 bytes are processed.

3.1.4 A simple and a more complex example

So far we have presented the parts of the following example:

```
input "measurements.dat" -> (0, dom("measurements.dat"), 0, 1, seq)
output "sum.dat" -> (4, append)
operation -> (10, 0, "measurements.dat":40)
```

The example job description specifies a job that reads `"measurements.dat"` sequentially into 40 bytes long buffers (10 times 4 bytes long integer), and each processing step calculates the sum of the 10 integers stored in the buffer. The final sum is written into `"sum.dat"`.

Unfortunately, this description requires the name of input and output files, which are usually unknown, or only specified as actual program parameters. Therefore, the extended job description should also be able to refer to the actual program parameters:

```
for (i in 1..arg[2])
  input arg[1]$i -> (0, dom(arg[1]$i), 0, 1, seq)
rof

output "union" -> (sum(i in 1..arg[2], dom(arg[1]$i))/2, append)
output "intersection" -> (arg[1]1/arg[2], append)

operation -> (2, 0, for (i in 1..arg[2]) arg[1]$i:dom(arg[1]$i))
```

The job description above contains file access and operation specifications using parameter variable array `arg`. Variable `arg[1]` refers to the first actual parameter which is an input file name prefix, `arg[2]` is the second actual parameter specifying the number of input files. An input file name is the concatenation of input file name prefix and input file index, e.g. `arg[1]$i` where $i \in [1, arg[2]]$.

Using the `for` description operation multiple resource access specifications can be written by "instantiating" the operation body with the actual variable. In our example we instantiate specification `input arg[1]$i -> (0, dom(arg[1]$i), 0, 1, seq)`, where $i \in [1, arg[2]]$.

The `sum(i in 1..arg[2], dom(arg[1]$i))` expression is equivalent with the following: $\sum_{i=1}^{arg[2]} dom(arg[1]i)$.

3.1.5 Implementation methods

The two major goals of our proposals concerning application development for Grid:

- make the work of specialists (physicist, biologists, etc.) developing applications for Grid easier, and
- enable Grid services to apply further optimization techniques in order to achieve shorter job execution times.

According to the basic idea behind our proposals, each Grid application has to solve two very different problems. First of all, the application has to provide the results required by the scientists. Secondly, the application has to be able to efficiently use and interact with the Grid services. Today, these aspects of Grid application aren't separated. For example, the `MPI2` communication codes are mixed with the "real" application code computing the required scientific results. Moreover, scientists (but not computer scientists) have to deal

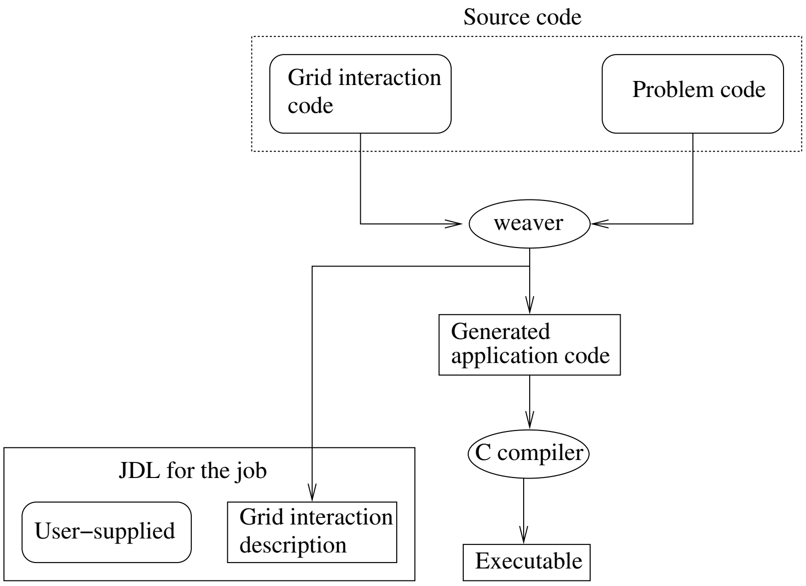

Figure 1: Aspect-based approach for application development in Grid

with the problems of parallel and distributed computing, and the communication between distributed processes.

We think that the aspect of Grid interaction has to be separated from the aspect of scientific calculations (see Figure 1). The scientists should be allowed to concentrate on the implementation of scientific calculations, and use (pre-implemented) Grid interaction aspects. Hence, a future Grid application will have two major modules expressing the two major aspects. These modules will have to be woven together generating the "final" application-code, which can be compiled by a C or C++ compiler.

The Grid interaction aspects may form an extensible library. Consequently scientists will not have to deal with this aspect of their applications but chose the appropriate aspect to be woven with their application code.

The Grid interaction aspect of a given application may offer the data required for better optimization. The information extracted from this module will be used to generate an extended JDL description for the job. Thus, each future JDL description will have a user-supplied and an automatically generated part.

3.2 Monitoring data access

In many cases data access patterns for an application will not be specified. One possible reason is that the author of the application is not able to specify them. Another possible reason is that the exact behaviour and source code of the application are not known to the scientist who wants to execute the application on the grid. In such cases monitoring can still be used to define data access patterns.

Many applications are executed not only once but several times, namely, for different input values. Parameter sweep applications are a good example. Jobs originating from the same application are likely to share data access patterns. If we can identify such jobs, monitoring the first few executions of the application can provide information to the grid middleware to optimize resource allocation for any subsequent executions. Middleware products supporting parameter sweep applications already exist for the grid. Using such a middleware can make the identification of jobs originating from the same application transparent to the scientists.

To enable monitoring of jobs, code instrumenting techniques can be used. There are a number of possibilities to merge monitoring code with the code of an application. In the following we will discuss some of these possibilities, focusing on the programming language C. (Other languages like FORTRAN should be handled in a similar way.) All of the discussed possibilities are based on monitoring standard file handling operations such as file opening (`fopen`), closing (`fclose`), reading (e.g. `fread`) and writing (e.g. `fwrite`) coming from the `stdio` library.

Data collected by monitoring the job must be further processed and appropriate information should be extracted and made available for the Grid middleware components performing optimization. For processing collected data existing monitoring tools can be used. The extracted information can be made available through a suitable extension to the JDL language.

3.2.1 Altering the source code

A possibility to extend an application with monitoring code is to ask the author of the application to modify the original source code. Obviously this modification should not be subtle, otherwise a scientist will be reluctant to perform it. Fortunately, with an appropriate set of C macros much of the necessary code transformations can be performed. Using such a set of macros and a new, monitored io library the required modification would be to replace an include statement with another one.

In many cases this little modification of the source code is affordable – though, of course, in many other cases it is not. For example, if the scientist who wants to run the application on the Grid has limited programming skills, he might not be able to perform even such a simple modification. Furthermore, this approach requires that the source code of the original application be available, which is also a major drawback. Finally, the expressive power of C macros might not be sufficient for inserting certain kinds of monitoring code into the code of the application.

A prototype has been developped as a proof-of-the-idea. In this prototype a monitoring version of some `stdio` functions has been collected in a library (performing merely logging of function invocations and actual parameters). A header file containing the appropriate macros to transfer calls to `stdio` functions to the monitoring functions has been provided. This header file should be included in the application instead of `stdio`, and the application should be linked to the monitoring library.

3.2.2 Altering the compiler

Another possibility is to leave the source code untouched, but intervene during the compilation process. The source code of the application can be altered by a special phase during the execution of the compiler.

This kind of code transformation probably has more expressive power than the previous approach. (This should be investigated further.) Another advantage is that it does not require the scientist to make any changes to the source code of the application. To obtain a monitored version of his application he only needs to compile the application with an appropriate compiler. It is the task of the site administrator, a person with suitable skills, to provide the necessary compiler or makefile settings.

The drawback of this approach is that the source code of the application should still be available and a recompilation on a suitably configured computer (e.g. a user interface machine) is required.

Many tools exist that enable parsing and transforming C source code. We have chosen a wide-spread tool, javacc for our prototype. The functionality of this prototype is the same as that of the approach presented in 3.2.1.

3.2.3 Altering the run-time system

The third possibility is to intervene during run-time, when shared libraries are linked to the application. It is possible to hack the run-time system on a Computing Element node in a way that certain stdio function calls are not dispatched to stdio, but to a custom shared library which performs monitoring as well as executing the actual stdio functions.

This is a very flexible solution. Only an environment variable has to be modified in the run-time system from non-monitored to monitored applications. Another advantage is that the source code of the original application is not needed, since no (re)compilation is necessary. A special shared library must be available on the Computing Element nodes, and an environment variable must be set according to e.g. the job description, whether the data accesses of the job should be monitored or not. Monitoring is completely hidden in the Grid middleware, and is transparent to the scientist who executes the application.

A disadvantage of this approach is that it is platform-dependent. According to our knowledge the techniques required by this approach can be used on Linux and Solaris machines, but other operating systems require further investigation.

In our prototype we use the same technique as the "fakeroot" tool of Debian Linux systems. A shared library (called `monio`) containing the monitoring versions of the file manipulation functions is "preloaded" during the execution (more precisely, during the run-time linking) of the Grid job on a Computing Element node. This means that application calls to standard `stdio` functions will be dispathed to this `monio` library, which in turn will dynamically load and execute the appropriate `stdio` functions using the same API as used by `ld.so`, the dynamic linker.

3.2.4 Altering at the operating system level

Changing the operating system slightly also offers a solution. All the techniques discussed so far assume the presence of a grid filesystem, which is a part of the operating system of the Computing Element nodes. A prototype of such a grid filesystem has been written by one of our students earlier. It might be possible to extend this grid filesystem with the possibility of monitoring file accesses. A drawback of this approach is that only the file accesses performed through the grid filesystem (namely remote file accesses) could be monitored this way.

4 Future work

First we should investigate what kind of information about data access patterns would be best to use for the optimization by the grid middleware. We could measure the usefulness of different kinds of information through simulations. (For example, OptorSim is a simulation package for Optor, the optimizer component of the Reptor replica manager.) According to the results we should define an appropriate extension to JDL. Then monitoring libraries should be developped that collect the necessary information to incrementally build JDL files for parameter-sweep applications.

Finally, there are a few more ideas that should be further investigated in the near future. They are summarized in the following subsections.

4.1 Resource access annotations

The program designer may help to determine the data/resource access pattern of an application by annotations. The JDL specification can be produced using these annotations.

4.2 Verification of resource consumption specifications

A long term goal to verify the annotations given by the programmer. Site administrators are allowed to determine a policy which is compared to verified resource access properties of an application. If the application requires too much resource, site administrators may refuse to schedule the job.

4.3 Filtering agents

If the application accesses a huge remote file in an irregular way, then a small agent may assist. The filtering agent moves near to the physical location of the file and filters the records needed by the application. Only the filtered records has to be transfered via the net.

5 Conclusion

The grid middleware (the resource broker and the replica manager) should ensure that applications are executed on the grid as efficiently as possible. Optimizing data access is an important aspect in this respect.

We introduce the concept of data access patterns. The description of the data access pattern of an application should provide sufficient amount and quality of information for the resource broker and the replica manager to determine an optimal data access strategy for the application.

We have examined the kinds of information that can make up data access patterns and proposed a resource consumption specification language that can form the basis of an extension to JDL. We have also proposed a framework for separating pure computational and distributed resource access aspects of an application. In this framework an extensible library of grid interaction patterns can be developped.

We suggest data access patterns be either specified by a programmer or, in case of parameter-sweep applications, determined by a monitoring grid middleware tool. We have investigated several ways to add monitoring of data access to an application.

References

[Pac02] Pacini, F.: JDL Attributes. DataGrid-01-NOT-0101-0_6-Note, 2002.

[GJMB] Gagliardi, F., Jones, B., Reale, M., Burke S.: European DataGrid Project: Experiences of deploying a large scale Testbed for e-Science applications.

[Sto01] Stockinger, H. et al.: File and Object Replication in Data Grids. HPDC 2001.

[Sto02] Stockinger, H. et al: Grid Data Mirroring Package (DGMP), 2002.

[Guy02] Guy, L. at al: Replica Management in Data Grids, 2002.

[Optor] Data Grid: Optor Home.
http://grid-data-management.web.cern.ch/grid-data-management/optimisation/optor/

[Rud01] Ruda, M.: Integrating Grid Tools to build a computing resource broker: activities of DataGrid WP1. Conference in Computing in High Energy Phisics (CHEP01), Beijing, September 3–7, 2001.

[Bal00] Balaton Z., Kacsuk P., Podhorszki N.: From Cluster Monitoring to Grid Monitoring Based on GRM and PROVE. Report of the Laboratory of Parallel and Distributed Systems, LPDS – 1/2000.

[Abr00] Abramson, D., Giddy, J., and Kotler, L., High Performance Parametric Modeling with Nimrod/G: Killer Application for the Global Grid?, IPDPS'2000, Mexico, IEEE CS Press, USA, 2000. http://citeseer.nj.nec.com/abramson00high.html

[Fakeroot] fakeroot. Debian GNU/Linux manual.
http://www.clc.unibe.ch/cgi-bin/man2html?fakeroot+1