GRID superscalar User's Manual
Version 1.5.0

CEPBA-IBM Research Institute

March 24, 2004

1 Introduction

The aim of GRID superscalar is to reduce the development complexity of Grid applications to the minimum, in such a way that writing an application for a computational Grid may be as easy as writing a sequential application. Our assumption is that Grid applications would be in a lot of cases composed of tasks, most of them repetitive. The granularity of these tasks will be of the level of simulations or programs, and the data objects will be files. GRID superscalar allows application developers to write their application in a sequential fashion. The requirements to run that sequential application in a computational Grid are the specification of the interface of the tasks that should be run in the Grid, and, at some points, calls to the GRID superscalar interface functions and link with the run-time library. The rest of the code already written for your application doesn't have to change, because GRID superscalar has bindings to several programming languages.

Our tool provides an underlying run-time that is able to detect the inherent parallelism of the sequential application and performs concurrent task submission. In addition to a data-dependence analysis based on those input/output task parameters that are files, techniques such as file renaming and file locality are applied to increase the application performance. Current GRID superscalar prototype is based on Globus Toolkit 2.x.

GRID superscalar is a new programming paradigm for GRID enabling applications, composed of an interface and a run-time. With GRID superscalar a sequential application, composed of tasks of a certain granularity, is automatically converted into a parallel application where the tasks are executed in different servers of a computational GRID.

The behavior of the application when run with GRID superscalar is the following: for each task candidate to be run in the GRID, the GRID superscalar run-time inserts a node in a task graph. Then, the GRID superscalar run-time system seeks for data dependences between the different tasks of the graph. These data dependences are defined by the input/output of the tasks that are files. If a task does not have any dependence with previous tasks that have not been finished or which are still running (i.e., the task is not waiting for any data that has not been already generated), it can be submitted for execution to the GRID. If that occurs, the GRID superscalar run-time requests a GRID server to the broker and if a server is provided, it submits the task. Those tasks that do not have any data dependence between them can be run on parallel on the grid. This process is automatically controlled by the GRID superscalar run-time, without any additional effort for the user.

Figure 1-1 shows an overview of the behavior that we have described above.

The reason for only considering data dependences defined by parameter files is because we assume that the tasks of the applications which will take advantage of GRID superscalar will be simulations, finite element solvers, biology applications... In all such cases, the main parameters of these tasks are passed through files. In any case, we do not discard that future versions of the GRID superscalar will take into account all data dependencies.

GRID superscalar applications will be composed of a client binary, run on client host, and one server binary for each server host available in the computational GRID. However, this structure will be hidden to the application programmer when executing the program.

Application code

```
initialization();
for (i=0; i<N; i++){
    T1 (“file1.txt”, “file2.txt”);
    T2 (“file4.txt”, “file5.txt”);
    T3 (“file2.txt”, “file5.txt”, “file6.txt”);
    T4 (“file7.txt”, “file8.txt”);
    T5 (“file6.txt”, “file8.txt”, “file9.txt”);
}
```

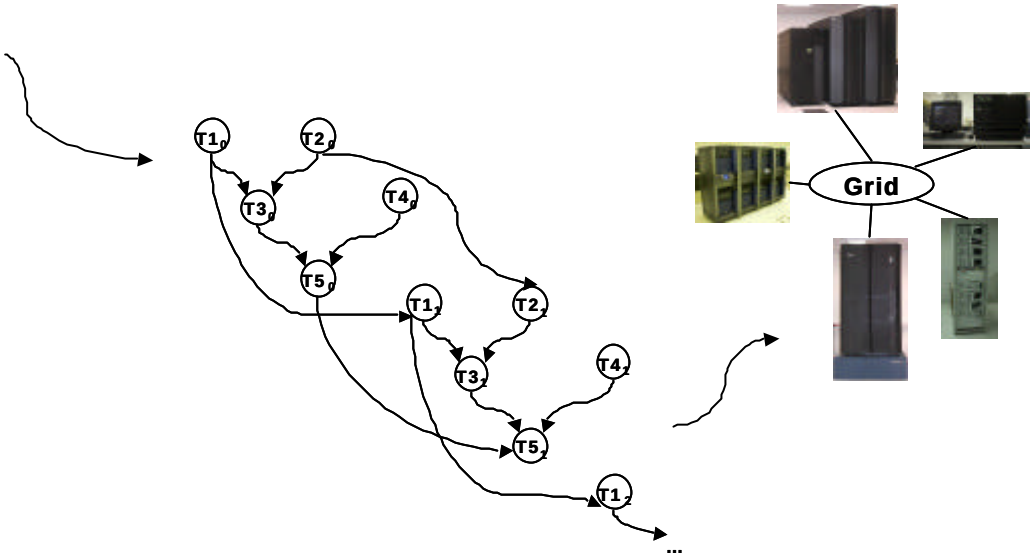

Figure 1-1

2 Developing your program with GRID superscalar

To develop an application in the GRID superscalar paradigm, a programmer must go through the following three stages:

1. Task definition: identify those subroutines/programs in the application that are going to be executed in the computational Grid.
2. Task parameters definition: identify which parameters are input/output files and which are input/output generic scalars.
3. Write the sequential program (main program and task code).

In the current prototype, stages 1 and 2 (task definition and task parameters definition) are performed by writing an interface definition file (IDL file). This interface definition file is based in the CORBA IDL language, which allows an elegant and easy way to write and understand syntax. We selected that language simply because it was the one that best fitted our needs, although GRID superscalar does not have any relation with CORBA. We are going to see all this in more detail into this chapter 2.

2.1 Quickstart

This section is intended to be as a reference of the steps that you have to follow when developing your program with GRID superscalar.

- Define an IDL file named <myapplication>.idl that contains the headers of the functions that are going to be run on the Grid. Write as parameters all files and scalars involved in the computation, trying to avoid out and inout scalars.
- Generate stubs and skeletons with gsstubgen <idl_file> in C/C++ case and gsstubgen -s -p <idl_file> in Perl case.
- Write / change your master code to call this new defined functions. Use GS_On() at the beginning, GS_Off(0) when the program ends correctly, GS_Off(-1) when you detect an error in the master and the file management primitives when working with files in the master (don't expect that the files have their original names). Avoid using GS_Barrier.
- Create a file at the workers named <myapplication>-functions.[c | pm] that contains the body of the functions defined in the IDL file. Use passed parameters instead of the expected names of the files. Call external binaries with GS_System and leave a possible error code at gs_result.

In next sections you will get more detailed information about each step. However, you can go to section 3.1 to see a quick guide about how to run your program.

2.2 Identifying functions that will be run on the GRID

In application programming, there are some options available in structuring the code. One really useful way is to program functions, instead of programming

everything in a big main function. This helps in two ways: it makes your code easier to understand, and allows you to repeat the same functionality in other stages of your application.

This basic programming technique will be the key to *gridify* your application. Your code may have some computation that you may want to be performed on the grid. This computation can be already in a function, called from the main program. If this is not the case, we recommend you to put your code into a local function, in order to ease even more the use of GRID superscalar.

Another important step is to define the header of the function properly. You have to put in this header the files needed (input files, output files, or input/output files) and scalar parameters needed (input or output) (i.e. you could need a scalar value to start a simulation). If you need to return a file, or a scalar, write it in the header parameters as an output parameter. This way you can return more than one value or file. Current prototype doesn't allow the functions to have a return value, so you have to return this value in the header.

This whole process will be seen really clear in our matrix multiplication example. One typical operation done between matrixes is the multiplication. When the matrixes grow in size, it grows also the complexity of the algorithm. Then we search a way to speed up this computation trying to parallelize our code. A first step is to divide matrixes in blocks, so we get several advantages from a version without doing this division. We don't need a full row or column to do some calculation, because we can operate between blocks. Another advantage is that you don't need to have all matrixes in memory, because we just need the blocks that are going to be operated. This is known as an out-of-core implementation.

This example is included in the GRID superscalar distribution, so you can follow this explanation while looking at the source code. We see that in our main code (matmul.cc) there are three local functions: PutBlock, GetBlock and matmul. The file named block.cc contains the definition of a block, and some useful operations. We want to put the matrix multiplication running on the Grid, so we must pay attention to matmul function. We see that the definition is right, because it has an input block named f1, another input block named f2, and an input/output block named f3.

```
void matmul(char *f1, char *f2, char *f3)
```

Each block is stored into a different file. We can suppose a less favorable situation, like this one:

```
double matmul(char *f1, char *f2, char *f3)
```

Imagine that you have a returning double, which has the mean value between all the elements of the block. We recommend you to add this double to the header, and remove it from the return value, so next steps will be even easier.

2.3 Defining the IDL file

The IDL file describes the headers of the functions that will be executed on the GRID. If you have this functions already defined with a function structure in your main code, this step will be really simple. You just have to write your function headers in our IDL form into a file called <myapplication>.idl (we will assume from now that is named app.idl). In order to learn how the syntax works, we present a generic example:

```
interface MYAPPL {
        void myfunction1(in File file1, in scalar_type scalar1, out
File file2);
        void myfunction2(in File file1, in File file2, out scalar_type
scalar1);
        void myfunction3(inout scalar_type scalar1, inout File file1);
};
```

As you can see there is one requirement needed in this interface: all functions must begin with **void**. If you have to return a parameter, you have to specify it as an output parameter. Files are a special type of parameters, since they define the tasks' data dependences. For that reason, a special type **File** has been defined. This type is also needed to differ a file from a string that could be needed in your function as an input (i.e. when passing modifiers to a simulator call, -v –f ...). Let's detail what combinations are possible for each parameter:

```
in File, out File, inout File, in scalar_type, out scalar_type, inout
scalar_type
```

The scalar_type can be one of these: char, wchar, string, wstring, short, int, long, float, double and boolean.

Another important thing to have in consideration is that we do not recommend the use of output scalar parameters, because they will have a little influence in the parallelism extracted from your code (it can be reduced). This only happens with output, or inout scalar parameters, not with input scalars. So, if you don't really need a scalar value to go on with your algorithm (i.e. when you need this value to take a decision), don't put it as an out scalar_type.

We can see all this now in the matrix example. We are going to create a file named matmul.idl. This file is going to have this content:

```
interface MATMUL {
        void matmul(in File f1, in File f2, inout File f3);
};
```

So we have two input files, and an input/output file (where the multiplication is going to be stored). Remember that we don't have to add GetBlock and PutBlock functions to this IDL file, because they are just functions to support our implementation (they don't have any computation).

If you don't have your code structured in functions, this and next steps will be not so easy, but won't be difficult at all. You have to think what parts of your code are needed to be run on the GRID, and write a line in your IDL file for each of these parts. There is also mandatory to see what files and parameters will be needed as inputs of this part of the code, and what files and parameters are considered as results or outputs. You just have to write it following the syntax described above.

2.4 Generating stubs and skeletons

From the interface definition that we have done in the previous step, some code is automatically generated by **gsstubgen**, a tool provided with the GRID superscalar distribution. This automatically generated code is mainly two files: the function stubs and the skeleton for the code that will be run on the servers. If you are not familiar with this two terms (stubs and skeleton) we can say as a summary that stubs are wrappers on the client (or master) side, and skeletons are wrappers on the server (or

worker) side. Some specifications are needed in order to communicate a client and a server, and this wrappers are the key point to do it (they can code or decode parameters, pass some information between them, ...).

From the user's point of view, he just has to call in C/C++ case:

```
gsstubgen app.idl
```

Or in Perl case:

```
gsstubgen –s –p app.idl
```

You can also execute gsstubgen without any parameters to see a list of possible modifiers to choose an option. At current version of GRID superscalar we have bindings for C/C++ and Perl. We are going to see in more detail what happens in each case. These bindings have in common that the name chosen for the IDL file will determine the name of the generated files.

2.4.1 C/C++ Binding

For C and C++, the files generated by gsstubgen are: app-stubs.c, app-worker.c and app.h.

We can see now how will be this files in our matrix example. Figure 2-1 shows the stubs file that will be generated for the IDL file (defined in previous section 2.3) when the C/C++ interface is used. For each function in the IDL file, a wrapper function is defined. In the wrapper function, the parameters of the function that are strings and filenames are encoded using base64 format. Then, the **Execute** function is called. The Execute function is the main primitive of the GRID superscalar interface. It's the entrance point to the run-time.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <gs_base64.h>
#include <GS_master.h>
#include "matmul.h"

int gs_result;

void matmul(file f1, file f2, file f3)
{
   /* Marshalling/Demarshalling buffers */

   /* Allocate buffers */

   /* Parameter marshalling */

   Execute(matmulOp, 3, 0, 1, 0, f1, f2, f3, f3);

   /* Deallocate buffers */

}
```

Figure 2-1

The other file automatically generated by gsstubgen is shown in Figure 2-2. This is the main program of the code executed in the servers. Inside this program,

calls to the original user functions are performed. Before calling the user functions, the parameters are decoded.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <gs_base64.h>
#include <GS_worker.h>
#include "matmul.h"


int main(int argc, char **argv)
{
   enum operationCode opCod = (enum operationCode)atoi(argv[2]);

   IniWorker(argc, argv);

   switch(opCod)
   {
      case matmulOp:
         {



            matmul(argv[3], argv[4], argv[6]);



         }
         break;
   }

   EndWorker(gs_result, argc, argv);
   return 0;
}
```

Figure 2-2

2.4.2 Perl Binding

Also three files are generated for the Perl binding: app-stubs.c, app-worker.pl and app.i. You must call gsstubgen with flags –s –p in order to generate these files. File app-stubs.c is exactly the same file as the generated for the C/C++ case (see Figure 2-1). File app-worker.pl is the main program of the code executed in the servers (similar to app-worker.c for the C/C++ case (Figure 2-2) but in Perl).

The file app.i is an interface file that will be used by SWIG. SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages, primarily common scripting languages such as Perl, Python, Tcl/Tk and Ruby. Basically, the app.i file it is a translation from the IDL syntax to the interface syntax required by SWIG of the application functions interface.

From file app.i, SWIG generates two files: app-wrap.c and app.pm. File app-wrap.c is linked with file app-stubs.c and with the GRID superscalar library and a dynamic library is generated (app.so). File app.pm indicates the Perl interpret to dynamically load the library (app.so) when the application functions specified by the IDL file are called from the client program. These all files are required at the master side.

2.5 Writing the Master

The main program that the user writes for a GRID superscalar application is basically identical to the one that would be written for a sequential version of the application. Maybe you will have to modify a little your functions (that is, the header), because you have to call now the functions described in your IDL file. If your program was not written in functions, you will have to extract the code you have identified to be run into the GRID from your main program, and call the primitives that you have described in your IDL (each primitive corresponds to a part of your code). This is like putting the code from a part of your program into a function, but the functions won't be written here. You can save the code into another file or leave it here by now (outside the main source of your program).

Other differences would be that at some points of the code, some primitives of the GRID superscalar must be called. For example, GS_On() and GS_Off() are called at the beginning and at the end of the application respectively (even if this end is caused by a premature exit). As these functions are defined in the GS_master.h file (given with the GRID superscalar distribution), it is necessary to include this file. Also you have to include app.h file (generated with gsstubgen), because it contains the headers of your new GRID superscalar functions (defined in app.idl).

In Perl case you must include the GSMaster module and the app module (remember that the syntax is: "use GSMaster;"). The On and Off functions are called as GSMaster::on(), and GSMaster::off(). And now your local functions are in an external module, so you must call them beginning with app::your_function() (with all parameters, of course).

Another change would be necessary on those parts of the main program where files are read or written. Since the files are the objects that define the data dependences, the run-time needs to be aware of any operation performed on a file. Let's see all those primitives. We have detailed Perl syntax in parentheses. Remember to put the name of the module before the call (GSMaster::).

2.5.1 Special primitives

- **GS_On() (on()):** Tells the GRID superscalar run-time that the program is beginning. The best place of putting this is at the beginning of your main code, but you can put it later, always considering that you cannot call any GRID superscalar primitive or function if you have not called first GS_On.
- **GS_Off(code) (off(code)):** This call will wait for all remote tasks to end, and will tell to the GRID superscalar run-time to stop. In order to indicate an error situation (i.e. when your program has to end prematurely because you detect an error) you have to set code to -1. Take into account that this GS_Off(-1) will exit your main program. You can put this also at the end of your code with GS_Off(0) (indicating that there has been no error). GS_Off(0) won't exit your main program, but remember that you won't be able to call any GRID superscalar primitive or function from this point and till the end of your program.
- **GS_Barrier() (barrier()):** In some special cases you may need this advanced feature. Sometimes you might need all the submitted tasks to finish, in order to take a decision and start working again. GS_Barrier() allows you to do that kind of synchronization, as GS_Off does, but it allows you to call more GRID superscalar functions later. Don't use this

function unless you don't have any other choice, because it can severely slow the parallelization of your code.

- **GS_Open(filename, mode)** and **GS_FOpen(filename, mode):** As explained in previous section 2.5, GRID superscalar needs to have full control over the files. These primitives will allow you to work with files while keeping GRID superscalar in control. They both return a descriptor that **MUST** be used in order to work with these files (i.e. to call read/write functions). These descriptors correspond to the ones returned by your C library (when using open and fopen), so you won't have to change following C library calls that work with these file descriptors. Modes currently supported are: R (reading), W (writing) and A (append). Perl case is special, because several functions are defined: open_r(*file_handle, file_name), open_w(*file_handle, file_name), open_a(*file_handle, file_name).

Because some file renaming techniques are used to avoid data-dependencies in your code (and so achieve more parallelism), you have to use this file descriptor returned in order to work with the file. There is no guarantee that the files will be available with the expected name.

- **GS_Close(file_des)** and **GS_FClose(file_des):** You have to call this primitives to close the file you opened before. The previous file descriptor explained as the return of the GS_Open and GS_FOpen primitives must be used here as a parameter. You cannot forget this step, or your GRID superscalar execution will never end. Working with Perl you have to call close(*file_handle).

You just have to replace your calls for opening and closing files by GRID superscalar primitives to do this. There is no need to change your read/write calls.

Another important point is that you can't rename a file in your main code, because this can disturb GRID superscalar run-time. If this renaming is unavoidable, you can copy that file giving to the new copy the name you desire, but always using the GRID superscalar file primitives.

The current set of specific GRID superscalar primitives is relatively small, and we do not discard the possibility that more primitives could be included in future versions. However, what is more probable is that these functions will be hidden to the programmer by writing wrapper functions that will replace the system functions.

In the matrix multiply example, our master will be:

```
#include <time.h>
#include <stdio.h>
#include <errno.h>
#include "GS_master.h"
#include "matmul.h"

int main(int argc, char **argv)
{
        long int t = time(NULL);
        char f1[15], f2[15], f3[15], file[15];
        FILE *fp;

        GS_On();
......
        for(int i = 0; i < MSIZE; i++)
```

```
                for(int j = 0; j < MSIZE; j++)
                        for(int k = 0; k < MSIZE; k++)
                        {
                                sprintf(f1, "A.%d.%d", i, k);
                                sprintf(f2, "B.%d.%d", k, j);
                                sprintf(f3, "C.%d.%d", i, j);

                                //f3 = f3 + (f1 * f2)
                                matmul(f1, f2, f3);
                        }

        GS_Off(0);

        printf("Total time:\n");
        t = time(NULL) - t;
        printf("%d Hours, %d Minutes, %d Seconds\n", t/3600,
(t%3600)/60, (t%3600)%60);

        return 0;
```

In this particular case, we just have to add GS_On() and GS_Off(), because matmul is defined exactly with the same parameters that in our sequential version. We have decided to remove all the local functions that we don't need, and leave them in another file.

2.6 Writing the Workers

Additionally, the user provides the code of the functions that have been selected to run on the GRID. The code of those functions does not differ from the code of the functions for a sequential application. The only current requirement is that they should be provided in a separate file from the main program. This file must be called app-functions.[c | pm] (remember that app is the name we gave to the IDL file). Moreover, there are some basic rules to build it: you have to include GS_worker.h file (given with GRID superscalar distribution) and app.h (generated by gsstubgen). This file will have as many functions as defined in your IDL file, so you have to copy the code of your functions here, or, if your code was not structured in functions, the parts of the code according to the ones defined in the IDL file. You can find (and copy) generated headers for your functions at app.h file. In Perl case you have to write your app-functions.pm file also copying your functions, and you should look at the IDL file and shift the parameters into variables in the same order. Remember to "use GSWorker" module.

Again you must remember that renaming techniques could have been applied to files, so you cannot refer to a file with the name you think it has. You have to use the input/output parameters defined in the function header. By the way, you are allowed to create a temporary file, with the name you prefer (but ALWAYS referring to current working directory), and do whatever is required. So you can't create temporary files with absolute or relative paths. All temporary files will be destroyed at the end of the task.

As an example, Figure 2-3 shows the code for the matrix multiply function.

```
#include <time.h>
#include <stdio.h>
#include <errno.h>
#include "block.cc"
#include "GS_worker.h"
#include "matmul.h"

#define BSIZE 2 //Blocks size in elements
```

```
block<double> * GetBlock(char *file, int rows, int cols)
{…}

void PutBlock(block<double> *A, char *file)
{…}

void matmul(char *f1, char *f2, char *f3)
{
        block<double> *A;
        block<double> *B;
        block<double> *C;

        A = GetBlock(f1, BSIZE, BSIZE);
        B = GetBlock(f2, BSIZE, BSIZE);
        C = GetBlock(f3, BSIZE, BSIZE);

        A->mul(A, B, C);

        PutBlock(C, f3); //A and B are sources
        delete A;
        delete B;
        delete C;
}
```

Figure 2-3

We can see that our matmul-functions.c file needs to include the definition of the block (block.cc), and define the size of the block. Also PutBlock and GetBlock functions are required to get the blocks from disk to memory, and then proceed with the multiplication. These functions could have been also defined in a separated file and then included in matmul-functions.c.

There are some special variables and primitives that must be called when creating the worker code. We will give more details in the following subsection.

2.6.1 Special primitives

- **GS_System(command_line):** When you need to call an external executable file (i.e. a simulator), you have to use GS_System, passing as a parameter the command line to be executed. You can suppose that your current working directory is the one that you have defined for that machine in broker.cfg file (its working directory) (see section 3.4 "How broker.cfg works"), so you can use an absolute path or a relative path to call that program.
- **gs_result:** This is not a primitive. This is a special variable that can be used to pass an error code to the master, so the master can stop the execution. If you don't use it, gs_result defaults to 0 (that means no error is detected in the task). If you detect an error, you can put an error code in this variable. This code must be higher than 0, because 0 is used to say that everything is ok, and negative values are reserved for the GRID superscalar run-time. You can even build your own error code mapping to detect what is happening in the worker by giving each number a meaning.

And now we have all the programming work done, so we are ready for running our application.

2.7 Hints to achieve a good performance

When programming your application, you can take into account several indications in order to achieve a better performance than if you don't. This is not a mandatory thing to do, because you can have already your code programmed and you don't want to severely modify the sources. So you can run your application without knowing anything about this section, but we recommend you to follow reading because maybe with some little changes you can really increase the performance of your application.

The first restriction we find when trying to run some tasks in parallel is when a true data-dependence is found. This happens when a task wants to read a file (input File) that is generated at a previous task (output File). If the input file is not really necessary (i.e. it could be some debug information, not needed data) we recommend that you do not include this file as an input file in the task definition at the IDL file.

You could also think about other data-dependencies, when a task needs to write in the same file that a previous one, and when a task needs to write into a file that has first to be read by another task. You don't have to worry about these dependencies, because GRID superscalar will eliminate them.

The next indication is about out scalars. In section 2.3, we have described that you can define a parameter as an output scalar, but we also point that when you define this kind of parameter, the performance could be worst than if you don't. That is because you can be using this parameter immediately after calling to this IDL defined function. Then, the GRID superscalar runtime doesn't have any other possibility than to wait for this task to complete, so this output scalar will be available. This wait can be hidden if GRID superscalar has enough tasks available to be run in parallel, and when the task with the output scalar is early scheduled for execution. If we don't meet this conditions, performance will diminish.

Another thing to avoid when trying to get a better performance is the call to GS_Barrier. We have presented it as an advanced feature in section 2.5.1 because in most of cases you will never use this call. In other cases, you may need it. When you call to GS_Barrier you tell the GRID superscalar run-time to continue to run previous generated tasks, but wait for all of them to finish. This waiting means that no new tasks are going to be generated from this point (main code is not going to continue) till all previous tasks are done. This synchronism point makes you to loose potential parallelism. So we recommend that you don't use this call unless there is no other option.

The last thing you can consider is to turn GS_SOCKETS to 1 in order to allow communications by sockets. In current prototype this is only allowed when working with the C/C++ bindings (in Perl is not supported). GRID superscalar works with files to achieve communication between the master and the workers. But, when all involved machines have external connectivity, you can set this communication to be done by sockets. This way of sending messages is faster, because no information is written to disk, and it is sent directly to the destination. We recommend that you take benefit from this advanced feature if your machines accomplish the requirements.

2.8 Known restrictions

You have to remember always that GRID superscalar run-time considers files as the main operands of each function (they define the data-dependencies and they have the main information required to execute a task, and to store results). In order to achieve better performance when executing your application, GRID superscalar uses

renaming techniques in your files. This way more parallelism can be extracted from your algorithm. But, that feature has several implications regarding file names when programming with GRID superscalar. Here is the list of restrictions.

- You have to use GRID superscalar special primitives to open and close files (section 2.5.1) in the master. And you must use the file descriptors returned by this functions to work with the files. You can never suppose that a file has its original name.
- You cannot rename files at the master side in your program. If this renaming is unavoidable, you have to copy the file to a new one with the new name (but remember to use GRID superscalar special primitives to handle files while doing this copy).
- You cannot remove files that are used as input or output parameters in your IDL defined functions before calling to GS_Off, because you cannot do it in a safe way.
- In the worker side, you cannot call an external application in your functions code by calling "system" (provided by the C library). You must use GS_System (section 2.6.1). But you can use a relative or absolute path when calling to this external application.
- Inside worker functions it is not allowed to refer to a file by its original name when this file is passed as a parameter from the function. You must use the parameters defined in the function. However, you can create a temporary file in current working directory, and refer to it by its name.
- You cannot define the same working directory between a master and a worker at broker.cfg (section 3.4).
- It is not available to define output files that belong to a shared disk. This feature is provided to share source files (section 3.5).
- Perl binding doesn't allow you to set GS_SOCKETS to 1.

You can see that not all this restrictions are because of the file renaming done by GRID superscalar. But you must consider them all.

2.9 The gsbuild tool

Since version 1.5.0, GRID superscalar distribution contains a tool named gsbuild, that automates the steps needed to obtain, from the sources, the files you need in order to run you developed application. If you execute it without parameters, the help will appear:

```
Usage: gsbuild <action> <component> <appname>

  Available actions:
    copy         Setup a compilation environment for the component for
customization.
    build        Build the selected component.
    clean        Remove generated binaries.

  Available components:
    master       Build or copy the master part.
    worker       Build or copy the worker part.
    all          Build or copy the master and workers parts.
```

```
  <appname> corresponds to the name of the application used for source
files and IDL files.
```

The files needed to run this tool are: <appname>.c, <appname>.idl and <appname>-functions.c. You also need to have previously installed in your system the tools automake, autoconf and the library named libxml2 version 2.6.x.

There are some things to take into account before using the gsbuild tool. You just can choose the build option when your code is done in C (not C++). If your main code is done in C++ and named <appname>.cc (or other) you will have to use the copy option, that will create an environment to configure your compilation options.

When this configuration environment is ready, you just have to call configure:

```
./configure --with-gs-prefix=$GS_HOME
```

Here $GS_HOME means the path were you have installed GRID superscalar. You can see other available options using the --help modifier, but the typical option is the one used in the given example.

3 Running the developed program

In order to run our developed application, we have to prepare the binary files in every machine that will be used for running our program, other configuration files and environment variables. This constraint applies in current version of GRID superscalar. Nevertheless, we are already working in a deployment environment, which will automate all this steps. Anyway, it's always good to know how the internals work, so we encourage you to follow reading.

This section will explain how to copy and compile your code, how to define environment variables, how to make your configuration files, and finally some basic Globus commands needed to run your program. We will suppose from now that you have an installation of GRID superscalar under $GS_HOME directory.

3.1 Quickstart

These are the main steps that you have to follow when running you GRID superscalar enabled application.

- Install Globus 2.2 or 2.4 (not 2.0 or 3.x) and GRID superscalar libraries.
- Copy in the corresponding machines the files that each of them needs. In C/C++ case you need app.h, app-stubs.c and app.c at the master, and app.h, app-worker.c and app-functions.c at the workers. When working with Perl involved files are app.pl, app.so and app.pm at the master, and app-worker.pl, app-functions.pm at the workers (section 3.2). Compile when needed.
- Consider modifying environment variables to change their default values. You can set the run-time to write debug information, leave logs at workers, pass messages with sockets instead of files, define which port uses your gsiftp servers, length of your parameters, length of paths and URL's, length of messages and length of the RSL string that describes each job. If you change a value, do the same at the worker side if this variable applies also at the worker side. Also define LD_LIBRARY_PATH when needed (section 3.3).
- Define broker.cfg that describes the machines involved in the computation (section 3.4), diskmaps.cfg that specifies shared disks and working directories (section 3.5), and finally estimations.cfg with the expected duration time for each function in each machine (section 3.6).
- Start your Globus proxy with grid-proxy-init (if it wasn't already started).
- Check that no file named .tasks.chk exists if you want to start the computation from the beginning.

The final step is running your application by simply executing the binary that contains your main code.

3.2 Copying and compiling your code

It's essential to know that some files are going to be at the master side, and some files are going to be at the worker side. But first of all we can talk about installation

requirements. Current version of GRID superscalar uses Globus Toolkit 2.2 or 2.4 (2.0 is not compatible). You need at least a client installation in the master machine, and a server installation in each machine that is going to be a worker. You also need to have the gsiftp service running in every machine involved in the computation (included in Globus Toolkit distribution), so transfer of files between machines can be done. From GRID superscalar you will need the GS-master library at master machine, and the GS-worker library at worker machines. You will need also the gsstubgen tool at the master side, and the library includes (GS-master.h at the master, GS-worker.h at the workers, and gs_base64.h at every machine). All these files are included in the GRID superscalar distribution.

The GRID superscalar also includes a tool called moved-libtool.sh. This tool repairs the library files if you decide to move them to a new directory. You have to use it like this:

```
moved-libtool.sh new_path_for_the_libraries $GS_HOME/lib/*.la
```

Regarding developed files location, where each file must be placed will be clearer with a graphical description. There are differences between C/C++ binding and Perl binding.

As Figure 3-1 shows, at the master we need app.c (that contains the main application), app.h (with previously defined function headers) and app-stubs.c (generated with gsstubgen). In the case of Perl (Figure 3-2) we need app.pl (with the main application), app.so (dynamic library generated to call GRID superscalar run-time) and app.pm (the Perl module that will be called from app.pl).

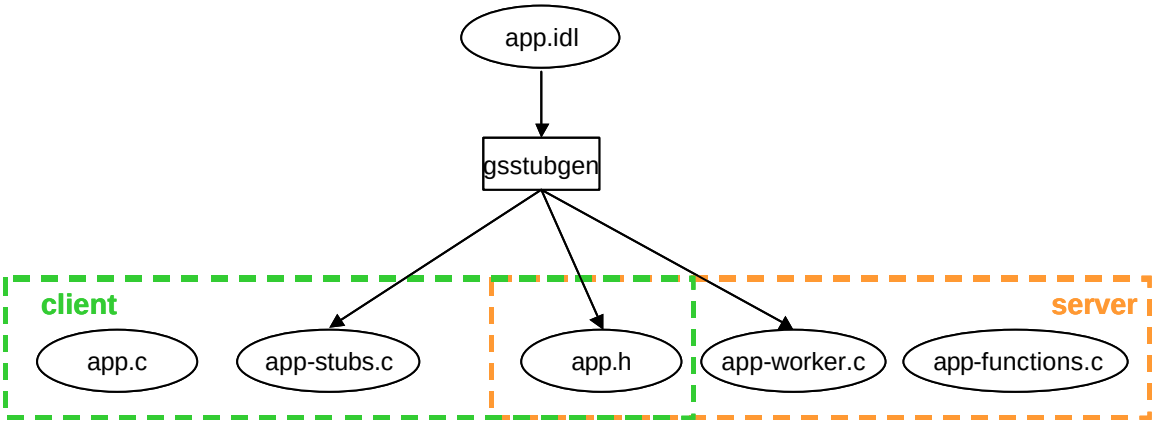

Figure 3-1

In the worker side we need app.h, app-worker.c (generated automatically) and app-functions.c (where we wrote all our functions). Looking at Perl again, we see that we need app-worker.pl (the generated skeleton) and app-functions.pm (with the functionality). In both versions we need an additional file called workerGS.sh that will define all environment variables at the worker side. We will talk about this in the next section.

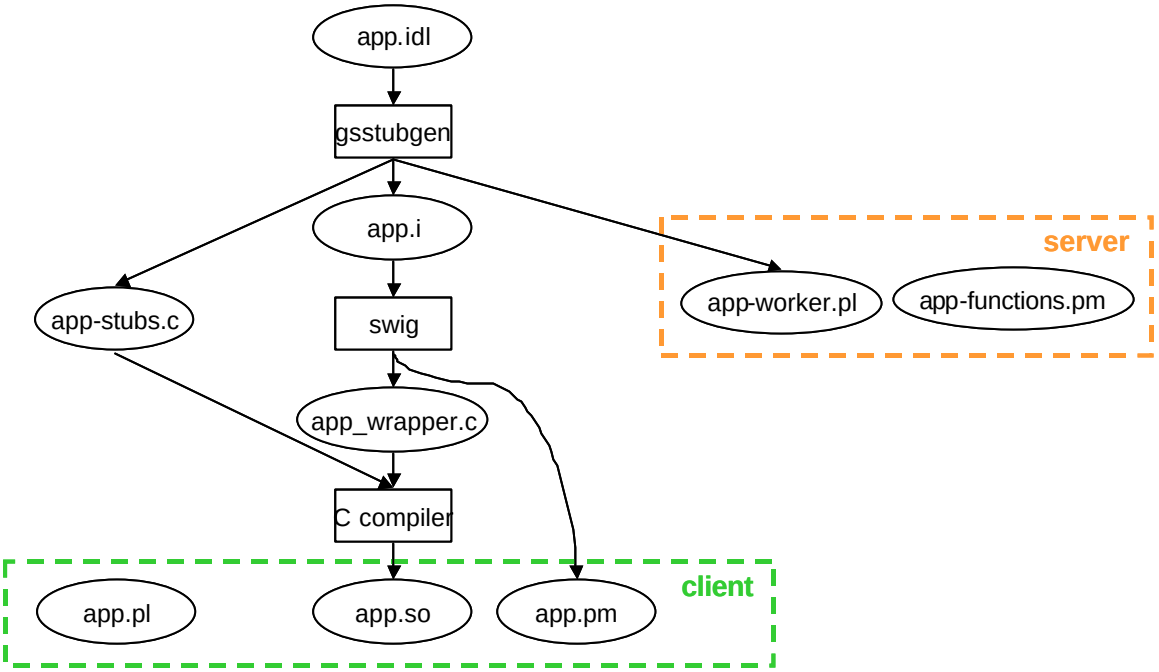

Figure 3-2

Final step in C/C++ binding is to compile all parts. You can take our example as the base to make your own Makefile:

```
CC=gcc
CFLAGS=-g -Wall –I$(GS_HOME)/include
CXX=g++
CXXFLAGS=-g -Wall –I$(GS_HOME)/include

all: matmul

matmul-stubs.c matmul.h: matmul.idl
        gsstubgen matmul.idl

matmul.o: matmul.cc matmul.h

matmul: matmul.o matmul-stubs.o
        g++ -Wall -g matmul.o matmul-stubs.o –L$(GS_HOME)/lib -o matmul
-lGS-master

clean:
        rm -f matmul *.o core* *~
```

As this master Makefile rules describe, our matmul.o must be linked with matmul-stubs object and the GS-master library. Remember that your app.c code must include GS_master.h (given with GRID superscalar distribution) and app.h in order to compile correctly.

At the worker we could have this Makefile:

```
CC=g++
CFLAGS=-g -Wall -I$(GS_HOME)/include

all: matmul-worker

matmul-worker: matmul-worker.o matmul-functions.o
        g++ -Wall matmul-worker.o matmul-functions.o -o matmul-worker
-L$(GS_HOME)/lib -lGS-worker

clean:
```

```
rm -f core *.o matmul-worker
```

Note that we are compiling with C++ because the block type included in matmul-functions.c is defined in C++. Here we have to link matmul-worker with matmul-functions object and with GS-worker library. The resulting executable will be named matmul-worker.

Remember that each part must be compiled in the machine where it is going to be run (in the C/C++ case), so we can avoid architecture incompatibilities and GRID superscalar library location differences.

3.3 Defining environment variables

Some environment variables are required to get your program running. These environment variables allow you to change some behavior of the GRID superscalar run-time without having to recompile neither your program nor the GRID superscalar library. You don't have to define them if you don't want, because they have a default value, but we recommend you to check if the default value satisfies your requirements. This part is concerning the master (or client):

- **GS_DEBUG:** You can set this variable to receive more or less debug information. When it's 20, the master will write at its standard output lots of useful information in order to determine potential problems. When set to 10, you will receive less information than before, but enough to follow the execution of your tasks. If you set this variable to 0 it means that we don't want debug information. Default value is 0.
- **GS_LOGS:** Set to 1 tells the master to leave execution logs of all tasks executed in all server machines. These logs will be named OutTaskXX.log and ErrTaskXX.log, according to the standard output and standard error messages given at that task (where XX is the number of the task). Each time you call to a function defined in the IDL file, a new task number is generated. This way you can know to which call corresponds the log file. If set to 0, this logs won't be left at workers. Default value is 0.
- **GS_SOCKETS:** Currently GRID superscalar allows two ways of master and worker communication in the C/C++ binding: sockets or files. The former means that the worker machine has external connectivity and can talk to the master with a direct connection. The latter means that the worker doesn't have direct external connectivity (i.e. a node of a cluster) and has to communicate with the master through files. To choose socket communication we have to set this variable to 1. Otherwise, if we want to use file communication we must set it to 0. Default value is 0. Note that in Perl version you cannot set it to 1. This is also explained in section 2.7.
- **GS_MIN_PORT:** This variable only applies when working with GS_SOCKETS set to 1. Some machines have constraints in connectivity, regarding to opened ports. For this reason you have to tell to GRID superscalar an available range of ports to be used to open a reply port when working with the sockets version. Default value is 20340.
- **GS_MAX_PORT:** The upper threshold. It is considered only when GS_SOCKETS is set to 1.

- **GS_SHORTCUTS:** Allow or not (1 or 0) shortcut mechanism between workers. This mechanism allows you to resolve faster more data-dependencies between tasks. Currently this feature is not supported so you won't be able to change its value to 1.
- **GS_FTPORT:** This integer tells us where the gsiftp port is located for transferring files. If you don't know which port is the gsiftp using, you can ask your system administrator. Default port is 2811.
- **GS_NAMELENGTH:** Maximum length of the names of the files involved in the computation. This means the files used when calling your new GRID superscalar functions defined with our IDL. Default value is 255.
- **GS_GENLENGTH:** Maximum length of scalar variables involved in the computation (i.e. maximum digits of a number). This value doesn't determine the precision when representing the scalar in the computer arquitecture. Default value is 255.
- **GS_MAXPATH:** Maximum length of a given path in your application. Must be 10 or more characters. Default value is 255.
- **GS_MAXURL:** Maximum URL size from your program (i.e. machine name plus invoked service and port). You can approximate this value by adding 40 characters to the maximum length of a machine name in your system. Default value is 255.
- **GS_MAXMSGSIZE:** Size of the messages that will be sent between the master and the worker. This could grow if you use lots of output files, or output scalars. Default value is 1000 (it's the lower limit).
- **GS_MAXRSL:** This variable is related to Globus. In order to run a Globus job a string that describes it must be constructed. This is done with a language called Resource Specification Language. In addition, you can receive a message from the master recommending you to raise this value, or telling that the value is not big enough. Default value is 5000 (the lower limit is set to 1000).
- **GS_ENVIRONMENT:** This variable is considered an advanced feature. Some extra environment variables could be needed to be passed when executing your jobs with Globus (i.e. when your jobs are parallel). These variables can be passed with this parameter. Your GS_ENVIRONMENT string can be as long as pointed by GS_MAXPATH. Each variable must be in parentheses: (VARIABLE1 value1)(VARIABLE2 value2) … Take into account that the content of GS_ENVIRONMENT will be sent to each worker machine.

Note that you can use "setenv" or "export" as the command to define an environment variable. This can change depending on the shell your system has.

Your main program is going to load the GRID superscalar shared library, so you have to put its path into an environment variable called LD_LIBRARY_PATH. You have to avoid erasing other previous defined library paths when defining the new one (check it with "env" command). An example follows:

```
setenv LD_LIBRARY_PATH $GS_HOME/lib:$LD_LIBRARY_PATH
```

Don’t do this if the variable doesn’t exist previously. This step is not needed when GRID superscalar libraries are installed in a standard location. You may ask your system administrator about this.

At the worker side there is a file named workerGS.sh. If there is not, you must create one. This file MUST have execute permission, because the master will invoke it. It’s content must be similar to this:

```
#!/bin/sh

export GS_MIN_PORT=20341
export GS_MAX_PORT=20459
export LD_LIBRARY_PATH=$GS_HOME/lib
../app-worker "$@"
```

Take into account now that we used “export” to define the environment variables, because we are now using a shell that supports this command. This file will set the environment variables in the worker side. You can suppose that no previous environment variables are defined, and set them here if needed (i.e. when running an external simulator). GS_MIN_PORT and GS_MAX_PORT are only required when working with GS_SOCKETS set to 1, and when we want to modify the default values. Also LD_LIBRARY_PATH must be set (if needed) considering the local machine, not the master. If you are familiar with scripting languages, you could think that you can add an exec before last line, so the new process will replace the current one. Don’t do this because if someone kills your worker, you won’t get any information about that.

3.4 How broker.cfg works

This will be the description of the first configuration file needed by your program developed with GRID superscalar. In this file we have to tell what machines will be our workers, what machine will be the master, and some characteristics of this machines. To understand the syntax, we can see a generic example:

```
Machine LimitOfJobs Queue WorkingDirectory (Master doesn't have
LimitOfJobs or Queue)

worker1 limit1 queue1 path1
worker2 limit2 queue2 path2
master path
```

The first line acts as a help to define all your machines. Then you have to specify the name of the worker, the limit of jobs that can be running in that machine at the same time (1 is the minimum value), the queue that has to be used when submitting to that machine (*none* if you don’t have to use a queue system), and the working directory for this machine (this means that you can find there the workerGS.sh file, and the app-worker file).

Concerning the master you just have to write the master hostname (that’s because the operating system doesn’t have always the right information) and the path where the master executable file is.

We can see an example of our matrix multiplication case:

```
Machine LimitOfJobs Queue WorkingDirectory (Master doesn't have
LimitOfJobs or Queue)

khafre.cepba.upc.es 4 none /home/at/khafre/MatMul
```

```
kadesh.cepba.upc.es 16 short /home/at/kadesh/MatMul
kadesh8.cepba.upc.es 8 mbench /home/at/kadesh8/MatMul
kandake.cepba.upc.es /home/at/kandake/MatMul
```

In this example kandake is going to be the master, and khafre, kadesh and kadesh8 are going to be the workers. We can submit 4, 16 and 8 jobs to each machine at the same time and kadesh jobs will go to a queue named short, and kadesh8 jobs will go to a queue named mbench. Each machine has also defined its working directory.

3.5 How diskmaps.cfg works

In current working systems, it's usual to have disks shared between various machines. With this configuration file we can tell the master when a shared disk exists, so there is no need of transferring the file between these machines, because both can reach the file.

Let's have a look at a generic example to understand the syntax:

```
Machine NameOfSharedDisk PathnameInTheMachine

worker1 disk0 path1
worker2 disk0 path2
master disk0 path3

worker1 disklocal1 path4
worker2 disklocal2 path5
master disklocal3 path6
```

This file is divided in two parts. The first part describes the shared disks between machines. You can name the disk as you want, but remember to use the same name for the same physical disk. You can read the information like this: "worker1 can see disk0 using path1", "worker2 can see disk0 using path2", and so on. This means that worker1 and worker2 can access to disk0, but it has a different path in each machine (it can be mounted in a different way). The important thing is that when we pass as a parameter of a function a file beginning with path3, the GRID superscalar run-time will detect that is placed in disk0, and will avoid the transfer when possible. This is also useful when we have mirrors of a database in several machines. If we are going to use this database as inputs, we can tell also where are this copies located in the server machines, and in the client machine.

The second part specifies the name of the disk where the working directory is placed (the working directory is defined in broker.cfg). Usually each machine will have its own disk name and the same path given into broker.cfg file. But sometimes you can be interested in some workers sharing a working directory (i.e. into a cluster). So some workers can have the same disk name, to warn GRID superscalar about this sharing.

An example will clarify things:

```
Machine NameOfSharedDisk PathnameInTheMachine

kandake.cepba.upc.es Disk0 /scratch/at/kandake/mat_files/
khafre.cepba.upc.es Disk0 /scratch/at/khafre/mat_files/
kadesh.cepba.upc.es Disk0 /scratch/at/kadesh/mat_files/
kadesh8.cepba.upc.es Disk0 /scratch/at/kadesh8/mat_files/

kandake.cepba.upc.es DiskLocal0 /home/at/kandake/MatMul
khafre.cepba.upc.es DiskLocal1 /home/at/khafre/MatMul
```

```
kadesh.cepba.upc.es DiskLocal2 /home/at/kadesh/MatMul
kadesh8.cepba.upc.es DiskLocal2 /home/at/kadesh8/MatMul
```

With this configuration all machines have access to matrix files stored into /scratch/at/kandake/mat_files/ (from the master point of view). Workers have to access this matrix files using another path. So these files won't be transferred between machines. In this real case, Disk0 is not the same physical disk in all these machines, but a replica. The working directories of each machine are also defined, but we can see that kadesh and kadesh8 use the same working directory (they belong to a cluster). This means that files will be just transferred one time to that zone, and both machines will be able to use them. Note that this diskmaps.cfg doesn't match with the one provided in GRID superscalar distribution into the matrix multiplication example. That cfg file doesn't have any shared disks.

3.6 How estimations.cfg works

With the objective of doing a better scheduling when giving tasks to the workers, a file with the estimations of the execution of the functions must be provided. This can be done by running an isolated execution of the function on the required machine, or with a simulator. Also the bandwidth of each machine (specified in bytes per second) must be given. This will help the runtime in deciding where is the best place to execute our task (depending on transfers of files needed and computation power of that worker). The syntax can be extracted from this generic example:

```
NumberOfOperations (new line) BandwidthBytesPerSecond
OpId0SimulatedTime OpId1SimulatedTime ... (and MasterBandwidth)

N
worker1_linkspeed time_op1 time_op2 ... time_opN
worker2_linkspeed time_op1 time_op2 ... time_opN
master_linkspeed
```

N has to be the number of operations defined at app.idl. Then the workers have its own link speed in bytes per second, and for each operation defined at appl.idl file, a measure of the time (in seconds) that the operation is supposed to last in that machine. The end of the file contains the master link speed also specified in bytes per second.

Let's see all this in a real case. We are going to suppose a different example from matmul, to add a little more size to the file.

```
NumberOfOperations (new line) BandwidthBytesPerSecond
OpId0SimulatedTime OpId1SimulatedTime ... (and MasterBandwidth)

4
1310720 0.1 3.7 0.2 0.001
1310720 0.2 5.8 0.3 0.001
1310720 0.17 5.6 0.25 0.001
1310720
```

This file will be for an IDL with 4 defined functions. When executed in the first worker defined at broker.cfg, first operation is expected to last 0.1 seconds. Second operation 3.7, and so on. We can see that the first worker is the fastest, and the second is the slowest one. The more accurate these values are, the better our scheduling policy will apply (avoiding file transfers).

We can also see our configuration file in our matrix multiplication example:

```
NumberOfOperations (new line) BandwidthBytesPerSecond
OpId0SimulatedTime OpId1SimulatedTime ... (and BandwidthSpeed)

1
1310720 0.001
1310720 0.002
1310720 0.0017
1310720
```

So, just one operation, that is really fast executed in server machines.

3.7 Am I ready to run?

Not yet. Before you run something that uses Globus (and GRID superscalar does) you have to start a user proxy. This proxy will authenticate the current user in all the machines that are going to be the workers, so you don't have to type your password every time you access to a machine. The command is:

```
grid-proxy-init
```

There is a useful flag (-valid) that allows you to make it last more than 12 hours. You can see this and more flags with -help. You can also use grid-proxy-info, to see if your proxy is already running, or grid-proxy-destroy, to stop your proxy.

If you don't have this command in the path, you better ask your system administrator about how to initialize your Globus environment.

Another important thing to consider is at the worker side. You can copy all the code in the worker side in whatever machines that are going to be workers from our execution. But you must remember to change in workerGS.sh the line that points to the LD_LIBRARY_PATH (it must contain the right path, regarding that machine). And also remember to change GRID superscalar library location from Makefile (if needed).

Yes! You are now ready! If you want to be really sure about this, you can do again some checks. Be sure to have all files mentioned in previous sections in the master side and in the worker machines. Remember to have the same values for environment variables at both sides, to have all code compiled and ready to run in all machines, and to define correctly all the cfg files at the master side. Check also that your workerGS.sh files have execute permission.

You can check also if have all GRID superscalar environment variables defined at the master side. Take a look at LD_LIBRARY_PATH and confirm that the GS-master library path is defined there (if needed). And just run your main code:

```
./app
```

3.8 Recovering from a checkpoint file

GRID superscalar has a feature that automatically checkpoints your tasks. This means that previously executed tasks won't have to be repeated when we detect an error in a task. When restarting from a checkpoint file, GRID superscalar will warn you with this message at the master:

```
FOUND CHECKPOINT FILE
```

This file is named ".tasks.chk" and is in you master's working directory. Sometimes you won't desire that GRID superscalar restarts from this checkpoint. If this is the case, you can simply delete this file from your file system and GRID superscalar will start execution from the beginning.

Do not try to build your own checkpoint file, because it can be really dangerous. This file is not the only one that stores information in order to recover your previous executed tasks.

4 Debugging your GRID superscalar program

4.1 Monitoring your execution

GRID superscalar run-time doesn't have by now a specific monitoring system. This means that if you want to see how your jobs are going you have to use standard operating system methods of monitoring processes. These commonly are: ps and top.

This section is not intended to be an operating systems tutorial, but we can give you some hints and examples of what you can, and cannot see.

So, when you run your master program, you can see several threads belonging to it (in particular you can see 3 for example in Linux, see Figure 4-1). This is normal, because the master creates a thread to listen to messages, and this thread needs a thread master in Linux. So don't be worried if the name of your master process appears more than once.

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
25273 username  19  10  3012 3012  2384 S N  94.8  2.3   0:00 in.ftpd
25272 username  19  10  3012 3012  2384 S N  88.7  2.3   0:01 in.ftpd
25266 username   9   0  6172 6172  2656 S     0.0  4.8   0:00 matmul
25267 username   9   0  6172 6172  2656 S     0.0  4.8   0:00 matmul
25268 username   9   0  6172 6172  2656 S     0.0  4.8   0:00 matmul
```

Figure 4-1

At the master side you can also see a process called in.ftpd and owned by root consuming CPU (Figure 4-1). This means that a file is being transferred, so the gsiftp service is being used.

If you want to see what processes are running in a worker you don't have any other way (by now) than to log into that machine and see it for yourself. You can see several processes that can tell you what is happening in that machine (all owned by your username). The most common is globus-jobmanager, started by globus-job-manager-script.pl (so you can see sometimes both, Figure 4-2). This process (globus-jobmanager) will handle the execution of your remote job (copy files, start the binary, ...). When files are being copied to that machine you can see the corresponding globus-url-copy process running (Figure 4-2). And when the worker binary is running, you can see a workerGS.sh and a ../matmul-worker processes (Figure 4-3). When the worker binary ends, you can see still the remaining globus-jobmanager. The most typical case is to have as many globus-jobmanager processes as the limit of jobs defined in that machine in broker.cfg file. If there is a queue system in that machine, some more processes can be there (depending on your queue system), but the basic ones described before will also appear.

```
username  7312 22.0  0.1  5356 3956 ?        S    12:06   0:00
/usr/bin/perl /aplic/GLOBUS/2.2/libexec/globus-job-manager-script.pl
username  7235  2.7  0.1  5208 3244 ?        S    12:06   0:00 globus-
job-manager -conf /aplic/GLOBUS/2.2/etc/globus-job-manager.con
username  7319  0.0  0.0  5216 2376 ?        S    12:06   0:00
/aplic/GLOBUS/2.2/bin/globus-url-copy gsiftp://kandake.cepba.upc.es:2
```

Figure 4-2

And when the transfers end:

```
username  8035  1.0  0.0  2152 1016 ?        S    12:06   0:00 /bin/sh
/home/at/khafre/MatMul/workerGS.sh 0 0 A.0.0 B.0.0 C.0.0 C.0.0
```

```
username  8036  1.0  0.0  1804  580 ?        S    12:06   0:00
../matmul-worker 0 0 A.0.0 B.0.0 C.0.0 C.0.0
```

Figure 4-3

Another hint of how many jobs are being executed in a worker machine at the same time is in the file system. You can see several sub-directories named "gram_scratch_<random_name>". Each of this directory is created to let the user work with it's own temporary files. So the directory where the worker is really being executed is the weirdly named one.

You can see also other files in master and worker named .RENXX (where XX is an integer number). Do not mess with these files, because they are the result of applying renaming techniques to your main code. They will be correctly removed during the execution and at the end of the main master program.

If your master seems stopped and no process owned by your username is in any of the worker machines, you may have a problem because the execution won't go on. In order to solve this situation, you have to see all information first, as the master debug information.

4.2 Master debug information

In previous section 3.3 (Defining environment variables) we have seen that we can set GS_DEBUG to 10 or 20 so the master is going to give us more information on how the execution is going. It is useful to redirect all this standard output to a file, so you can examine it with more patience.

The most important information you have to consider is the one that is given about the queues that are defined inside the GRID superscalar run-time. You can see prints about running, waiting, pending and ready tasks. Waiting means that they are stopped waiting for a file to be transferred, but this transfer has been started from another task. Pending means that the task still has data-dependencies to be resolved and ready means that it can be submitted at any time.

You can see also the decision that GRID superscalar takes when choosing which is going to be the next running task. When the sentence "ESTIMATION BEGINS" appears, this means that the run-time is deciding where to run the job. There is an estimation of transfer files and execution time for each task in the ready queue against each worker. Estimation comes like this:

```
ESTIMATION: 0.200000 Task: 14 Machine: 0 Size: 0.000000
ESTIMATION: 0.203664 Task: 14 Machine: 1 Size: 4802.000000
ESTIMATION: 0.203664 Task: 14 Machine: 2 Size: 4802.000000
```

You can see here that the task 14 is going to last 0.2 seconds in worker 0, and more than 0.20 in worker 1 and worker 2. That is because the files needed are not in these two workers, in contrast with worker 0, that already has the files (as pointed by the Size value, that means how many file bytes do we have to transfer to that machine). So worker 0 will be chosen as shown in sentence:

```
<------------------ MARKED MACHINE: 0 ------------------>
<---------------- SUBMITTED : 1 ---------------->
```

This also tells us how many jobs are submitted to that worker at the same time.

If you are familiar with Globus RSL language and its callbacks mechanism, you can find this also as printed information. If you are not familiar, the information is

really self-explaining. Remember that all this parameters refer to the worker where the job is going to be run.

There is also another important thing to remember about returning values of tasks. You can see at some point this debug information:

```
TASK 11 JUST EXTRACTED!!!
```
(and some lines later)
```
ERROR Code: 0
```

In this case the returning value of the task is 0, so everything is ok. But if something different from 0 is returned, a worker has detected an error, so the master is going to stop its execution. An error code different from 0 will be shown in the master even when you have GS_DEBUG set to 0. As described in section 2.6 "Writing the Workers", you can detect errors by setting gs_result to a value different from 0. So here we will know at the master when a worker fails. If you receive a negative error code this means that there is an operating system problem (your code can have an invalid memory reference, someone could have killed your process, ...). Probably you want to see what happens into the worker, and look at the worker log files.

4.3 Worker log files

As shown in section 3.3, we can tell GRID superscalar run-time to leave standard output and standard error information in the worker that has executed a task. This information can be really useful when trying to determine why our program doesn't run. You can print information from inside your app-functions.[c | pm] file to standard output and standard error that can give you a hint of what is happening there. Each call to an IDL function from your master main program generates a new task, so a new number to name it is also generated. First task will be named task 0, next will be 1, and so on. This will help you to determine which IDL call has generated a log file. Some default information is printed from the run-time at OutTaskXX.log:

```
Task: 0. SCode: 0
Getting stats of: TMP.0.cfg
We are sending this: 1 0 78.194940442219376564 3121 0
MasterName is: kandake.cepba.upc.es. ReplyPort is: 20342
```

This log is for Task 0 (so it must be named OutTask0.log). The SCode refers to the shortcut mechanism, but we don't have to worry about that, as explained when defining GS_SHORTCUTS environment variable. When the worker gets stats of a file, this means that this file is an output of this task. Beyond the "We are sending this" sentence we have a message that is going to be sent to the master. First integer refers again to shortcut stuff, so we don't worry, next we have the task number, all its output scalars, all the size of output files, and the last integer refers to the value of gs_result. This can be 0, positive or negative. A 0 value would mean that there is no error, a positive value would mean an error detected by the programmer, and a negative value would mean that a signal has been received. Several signals can be received, so this could tell you that your program has an invalid memory reference (typically a –11 error code), been terminated (almost always a –15 error code), aborted (–6) , ... Signal number 9 (kill) cannot be reprogrammed, so you will never receive a –9 error code (You will have to look at the worker logs to see if a worker has been killed). Not all signal numbers are standard, so, if you are not familiar with this operating system features, you can ask your system administrator about this.

4.4 Cleaning temporary files

There are several hidden files that you can find in your master and in your workers when running your application developed with GRID superscalar. These files, that are needed to implement techniques as renaming (to improve parallelism, and so performance of your application) or checkpointing (to avoid repeating computation that has been already done), are automatically erased during the execution of your program and when the application finishes. For some strange reasons, the application cannot finish correctly (i.e. when the master crashes) and some of these files can remain in their locations. There is no real need to deal with these files, because they will be overwritten if you execute again your application. However, we can see what are their names and which is their purpose, so, if you find yourself in trouble, you can decide if you want to delete them or not.

- **.RENXX:** These files are used for renaming techniques. They are different versions of a file during the original file lifetime. They can appear at the master and at the workers.
- **.GS_fileXX:** Some extra information must be saved when checkpointing local tasks in your main program. This information is stored in those files. They are created at the master side.
- **.tasks.chk:** This file is only in the master. It allows you to restart from a task your execution, without having to repeat previously done computations. If you delete it, the master will restart all the computations from the beginning.
- **OutTaskXX.log / ErrTaskXX.log:** Standard output and standard error from the task with number XX at the worker side. They won't be generated when GS_LOGS is set to 0.
- **destGen.XX:** They appear at the master and at the workers. This name identifies the files that are messages from a task between the master and the workers. When GS_SOCKETS is set to 1, these files don't have to appear. If the master is stopped, you can delete them without any danger.

Some files transferred as sources to tasks can remain in the working directory of the workers. You can also delete them with no danger if everything is stopped. However, remember that if you are planning to execute your program again, you don't have to worry, because these files will also be overwritten.

In addition, you can add to your Makefile some basic rules to erase all this files. So we have at the master side:

```
delete:
        rm –f .REN* .GS_file* .tasks.chk destGen.*
```

And now the worker side:

```
delete:
        rm –f .REN* destGen.*
```

So you can "make delete" anytime you want to clean all those files. You will seldom need to do this, but it could be useful if you find a bug in your master code, or even in GRID superscalar (although we hope you don't!).

5 Frequently Asked Questions (FAQ)

Here there are some typical questions that may arise when working with GRID superscalar. We recommend you to look in the table of contents of this manual to find faster what you are looking for.

5.1 Globus

5.1.1 What is Globus? Why do I need it? Can you give me some useful commands?

Globus provides services for running your jobs remotely, transferring files, and more. It is needed to access other machines outside your administration domain. There are some useful commands that you can test: grid-proxy-info (to see the status of your proxy), grid-proxy-init (to start your proxy), grid-proxy-destroy (to end your proxy), globus-job-run (to run remote jobs), globus-url-copy (to copy files between machines).

5.1.2 I have several log files In my workers' home directory. They are named gram_job_mgr_<number>.log

Usually, when a Globus job fails it leaves information in a log called gram_job_mgr_<number>.log. If you don't need the information inside, you can erase them safely. Depending on your Globus installation they can appear always, when errors rise, or never. You can contact your system administrator to know that.

5.2 GRID superscalar tools

5.2.1 When I use gsstubgen I get this output: "Warning: renaming file 'app-stubs.c' to 'app-stubs.c~'. Warning: renaming file 'app-worker.c' to 'app-worker.c~'. Warning: renaming file 'app.h' to 'app.h~'.". What is this for?

In this case gsstubgen has done backups for your old generated files from your IDL definition. This backups end with the '~' character. You can remove them by hand. Next time, if you don't want to generate backups, use –n flag.

5.3 The master

5.3.1 When I set GS_DEBUG to 10 or 20, the output of my main program seems to appear in really weird places. What is happening?

If you print something to the standard output the system has a buffer to print more information from one call. So it's normal that sometimes appears in weird places.

5.3.2 When I redirect all output given from the master to a file, sometimes at the end some information is missing. Why?

Again buffering of the operating system is cheating you. You can also see that the order of some prints also change when printing by screen or when printing to the file. But that's normal. You can repeat the execution and see how it ends printing by screen.

5.3.3 I get a message like this when trying to run the master: "ERROR activating Globus modules. Check that you have started your user proxy with grid-proxy-info"

You forgot to start your Globus proxy or its lifetime has expired. Try the Globus command grid-proxy-info to see if you have started it. If you have not, remember to use grid-proxy-init. If it has expired, you can run grid-proxy-destroy and grid-proxy-init again.

5.3.4 The master ends with this message (or similar): "./app: error while loading shared libraries: libGS-master.so.0: cannot open shared object file: No such file or directory"

You have to add to your environment variable LD_LIBRARY_PATH your GRID superscalar library location.

5.3.5 When I set GS_SHORTCUTS to 1 I get this message "ERROR: Check environment variables values". Why?

That is because you haven't read this manual! We said that you won't be able to turn this to 1, because file forwarding mechanism is no more supported. We don't discard to recover this feature in the future, so that's the reason because this variable still remains.

5.3.6 I get this message: "ERROR: Check environment variables values". But I have all variables defined and GS_SHORTCUTS is set to 0

Your environment variables are wrong or too small. You cannot set GS_SOCKETS to a value different from 0 or 1, for example. We have set some lower limits in order to run your master correctly. See chapter 3.3 "Defining environment variables".

5.3.7 When working with GS_SOCKETS set to 1 I get a segmentation fault at the master. More precisely, this happens when a previous execution ends (prematurely or not) and I try to launch the master immediately

The problem is that some previous jobmanagers stay running at worker machines, because socket verision of the run-time doesn't wait for them to finish (to be faster than file version). Before executing again be sure that no globus process remains in the workers, or simply wait 30 seconds (the higher time the running jobmanagers will stay when the worker ends).

5.3.8 I get this message: "******** ERROR AT TASK 0 !!! ********* ******** MACHINE khafre.cepba.upc.es ********* the job manager could not stage in a file

The cause can be that your gsiftp service is not reachable or is not started in your master. Be sure to have an opened port for it. You can telnet to that port (default is 2811).

```
localhost> telnet localhost 2811
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
220 localhost GridFTP Server 1.5 GSSAPI type Globus/GSI wu-2.6.2
(gcc32dbg, 1032298778-28) ready.
```

If you don't get this output (or a similar one), contact your system administrator and tell him that the gsiftp service is not working.

5.3.9 I get this message: "ERROR: Submitting a job to hostname. Globus error: the connection to the server failed (check host and port)"

One of your workers cannot run Globus jobs because the service called "gatekeeper" is not started or its port is closed by a firewall. You can do this to check it:

```
localhost> telnet hostname 2119
Trying 147.83.42.31...
Connected to hostname.
Escape character is '^]'.
```

Where hostname is the worker that we suspect is failing. The connection has to remain till you write 'quit'. If you get a "Connection refused" message, tell your system administrator that Globus is not working properly because the gatekeeper is not started or is unreachable.

5.3.10 When the master is going to end I get this message: "ERROR: REMOTE DELETION OF FILES IN MACHINE hostname HAS FAILED. Globus error: (error from system). Checkpoint file erased for safety reasons". What happened?

When the master ends it recovers all result files and erases temporary files in all the workers involved in the computation. If this final process fails, the master reaches a non consistent state. In this situation it cannot recover from the checkpoint file. You

can get your results by hand, and erase temporary files, or start your execution again from the beginning. The main reason that makes this error appear is when you don't have enough quota in the master to receive the result files, but check the "Globus error" sentence to know this more precisely.

5.3.11 I get an error like this when trying to run the master: "License Manager Error: Your license expired on 23/02/2004 Please contact Rosa M. Badia (rosab@ac.upc.es)." What is all this stuff about licenses? I haven't acquired any

We use to generate GRID superscalar distributions with expiration date, so you can take benefit from our new versions of GRID superscalar with new features and fixed bugs. It is not good that you remain with the same old (and possibly not bug-free) version forever.

5.4 The Workers

5.4.1 The first task executing returns an error of this kind "******** ERROR AT TASK 0 !!! **********". When I see log files at the worker side I find this at the ErrTask0.log: "../app-worker: error while loading shared libraries: libGS-worker.so.0: cannot open shared object file: No such file or directory"

You, probably with good intentions, deleted at workerGS.sh a line that defines the LD_LIBRARY_PATH environment variable to load the GS-worker library. You cannot remove it if your GRID superscalar library is not installed into a standard location. Just put it back.

5.4.2 I get this message when I try to execute a remote task: "******** ERROR AT TASK 0 !!! ********** ******** MACHINE hostname ********** the executable file permissions do not allow execution"

You must check that the workerGS.sh file in the worker named hostname has execute permission. To change permissions you can run "chmod ugo+x workerGS.sh".

5.4.3 The firs task ends with an error, but now when I look into the worker I find in ErrTask0.log: "workerGS.sh: ../app-worker: No such file or directory"

You have not compiled the worker in this machine.

5.4.4 Once more my first task fails but my log files are empty. That's crazy!

Be sure that your paths for finding the worker executable are correctly defined in broker.cfg, and that nobody has deleted last line from workerGS.sh. It has to contain this: "../app-worker "$@""

5.4.5 I always get errors when trying to run a task into a worker. Is it Globus fault? Is it GRID superscalar fault? Is it my fault?

The first thing you can do when the remote executions fail is to run a single test to check that Globus can run jobs. You can do:

```
globus-job-run worker1 /bin/date
```

And see if this returns the current date and time. If this fails, you can contact your system administrator and tell him that you cannot use Globus for running your jobs.

5.4.6 I receive this message at the master: "ERROR: Submitting a job to hostname. Globus error: the cache file could not be opened in order to relocate the user proxy"

Check if you have available disk space in that worker machine. This error can leave some .gram_scratch_<random_name> subdirectories in the involved worker.

5.4.7 I receive this message at the master: "ERROR: Submitting a job to hostname. Globus error: the job manager failed to create the temporary stdout filename"

This can be also a problem with quota in hostname.

5.4.8 I get this message: "ERROR: Submitting a job to hostname. Globus error: data transfer to the server failed"

The reason could be that you don't have enough quota on the worker machine to transfer your input files. Check this with the "quota" command.

5.4.9 After having a quota problem in a worker, I see some temporary files remaining. How can I manage to erase them correctly?

You can erase all subdirectories that are named .gram_scratch_<random_name>. Some input files can remain also (their names will be familiar for you). The rest of temporary files are described in section 4.4.

5.5 Other questions

5.5.1 I love GRID superscalar! It has saved me lots of work hours!

We will appreciate comments and suggestions about our tool. You can reach the authors at grid-superscalar@ac.upc.es.

5.5.2 I hate your run-time. It's giving me lots of problems.

Don't give up. If you really think you are in a situation that you cannot solve, we can try to see what could be happening in your particular case. Contact us at GRID superscalar mailing list (grid-superscalar@ac.upc.es).