Specification Document and Installation Guide of the Supercomputing API for the Grid (DRAFT)

Károly Bósa Wolfgang Schreiner

March 15, 2010

Abstract

In the frame of the Austrian Grid Phase 2, we have designed and implemented an API for grid computing that can be used for developing grid-distributed parallel programs without leaving the level of the language in which the core application is written. Our software framework is able to utilize the information about heterogeneous grid environments in order to adapt the algorithmic structure of parallel programs to the particular situation. Since our solution hides low-level grid-related execution details from the application by providing an abstract execution model, it is able to eliminate some algorithmic challenges of nowadays grid programming. In this paper, we present on the first feature-complete prototype of our topology-aware software system.

1 Introduction

In this paper, we present on a completed work whose goal was to design and develop distributed programming software framework and API for grid computing [8]. This software system is able to utilize the information about the grid environment in order to adapt their algorithmic structures to the particular situation.

Our solution is an advanced topology-aware programming tool which takes into account not only the topology of the available grid resources but also the point-to-point communication structure of parallel programs. In our approach, a pre-defined *schema* is assigned to each given parallel program that specifies preferred communication patterns of the program in heterogeneous network environments. The execution engine first adapts and maps this schema to the currently available grid resources and then starts according to this mapping the processes on the grid. Our API contains function calls which are able to query all the details of the mapping information which contains both the adapted communication structure of the program and the topological information of the allocated grid resources.

Regard an example where a user intends to execute a tree-like multi-level parallel application on the grid. She specifies in advance that the given application shall consist of 20 processes organized into a 3-levels tree structure. On the lowest level leaves belonging to the same parent process shall form groups such that each group contains at least 5 processes scheduled to the same local network environment. For this specification, our software framework is able to determinate a suitable partition of processes on the currently available grid resources and to start the processes according to this scheduling. The partition is based on some heuristics, e.g.: our framework prefers such tree structures where the sizes of the groups formed by the leaf processes belonging to the same parents are maximal; consequently the processes of each such group can be scheduled to a cluster. Furthermore, our API maps at runtime the predefined roles of processes in the specified logical hierarchy (global manager, local manager and workers) to the allocated pool of grid nodes such that the execution time is minimized.

2 Software Architecture

In our approach, the user assigns to each given parallel program a pre-defined *schema* that specifies a preferred communication pattern of the program in heterogeneous network environments. In our system the following kinds of communication schemas are currently employed [6]: the schema *singleton* specifies a number of processes which should be scheduled to the same local network environment; the schema *groups* specifies the number processes and either the accurate size of the *local groups* (the number of processes in the same local network environment) or a minimum size for the local groups and some restriction for the number of the local groups; the schema *graph* is similar, but it additionally defines edges/links between the local groups such that they describe a communication pattern; the schema *tree* specifies a tree-like multilevel parallelism with the given number of processes and the given number of tree levels, such that the size of the local groups located on the lowest level (the level of leaves) are not determined but some restriction are given for it; last but not least the schema *ring* is similar to the schema graph, but the local groups always compose a ring. For more detailed description of the communication schemas, see Section 3.

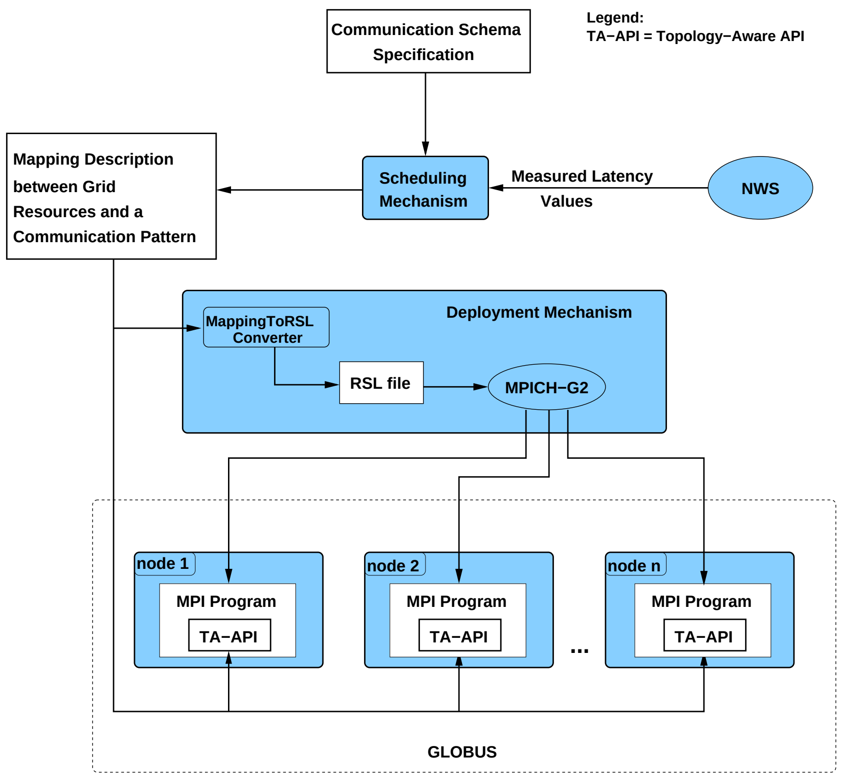


Figure 1: Overview on the Software Framework

We have finished the implementation of a first feature-complete prototype version of our software framework [8] called *"Topology-Aware API for the Grid"* (*TAAG*). The system is based on the pre-Web Service architecture of the Globus Toolkit [2] and on MPICH-G2 [12] and it consists of three major components (see Figure 1):

Scheduling Mechanism This component depends on the *Network Weather Service (NWS)* [14], which is a performance prediction tool that has become a de facto standard in the grid community. Since the NWS provides all necessary information concerning the utilizable grid resources, the user needs not know any detail of the grid architecture. In addition to these performance characteristics the scheduling algorithm needs a communication schema of a particular application specified in an XML format.

Before each execution of a parallel program on the grid, the scheduling mechanism adapts and maps a preferred communication pattern of the program to the available grid resources such that it heuristically minimizes the assessed execution time (for more details see Section 4). The output of the algorithm is an XML-based *mapping file* which describes a mapping between the grid resources and the given communication pattern.

Deployment Mechanism This mechanism is based on the job starting mechanism of the grid-enabled MPI implementation MPICH-G2 [12]. It expects a mapping file generated by the scheduling mechanism as input which contains among others the name and various locations of the executable, the designated grid resources and the partition of processes. It then starts in two steps the processes of an application on the grid according to the content of the mapping file:

- First, it distributes via gridFTP the mapping file into the directory `/tmp` on all designated grid machines.
- Then it generates a RSL expression from the mapping file; with the help of this RSL expression, it starts the application on the grid via MPICH-G2.

Topology-Aware API This API is an addition to the MPI interface. Its purpose is to query mapping files and inform parallel programs how their processes are assigned to physical grid resources and which are the designated roles for these processes. It provides information such as in which local group a particular process resides or which are the characteristics of local groups, graphs, trees or rings.

For representing the versatility of our API, we have developed some simple distributed example applications [7, 9] (e.g.: tree-like multilevel parallelism on the grid).

All the three major components of our software framework have completely been implemented and the entire system has already been tested successfully on the sites `altix1.uibk.ac.at` (SGI Altix 350), `lilli.edvz.uni-linz.ac.at` (SGI Altix 4700) and `alex.jku.austriangrid.at` (SGI Altix ICE 8200) of the Austrian Grid.

3 Specifications for Heterogeneous Communication Structures

In this section, we present some schemas used for specifying heterogeneous point-to-point communication structures for parallel programs. The parallel programs are classified into these schemas on the basis of the roles of their processes and the qualification of the used point-to-point channels among them (often used and rarely used channels). Common features of these schemas:

- They never include the grid client from where the programs are submitted by the user.
- They arrange processes into some groups, where each group is supposed to execute on a local network environment (cluster or LAN). The point-to-point channels among the processes of such a group are never specified by the schemas.
- For each parallel program going to be executed via our software system on the grid, we must define such a schema (only exception is the *singleton*, which is also used for scheduling pure MPI codes, see the next section).

3.1 Single Group

This schema *singleton* is used for scheduling programs on the grid which were designed for homogeneous network environments:

$$SINGLETON\{nrOfProcs, strictRestriction\}$$

with the arguments, we can specify the number of processes used by the program and a condition whether all processes must be schedule to the same local network environment. If it is not possible to find a cluster or LAN with the given number of available CPUs and the second parameter is true, then the scheduling will be unsuccessful. But if the second parameter is false, the scheduling mechanism always returns a possible distribution of the processes on some grid resources (which may belong to different local networks).

In case of an existing MPI program which does not comprise our API, the schema singleton will be used for finding an appropriate local environment for running the given number of processes.

3.2 Set of Groups

The schema *groups* is for specifying a condition how to organize a given number of concurrent processes into as few local groups as possible on an available grid environment (as was mentioned before the point-to-point structure within a group is not interesting for us at the moment):

$$GROUPS\{nrOfProcs, minSizeOfGroups, divisorOfGroupNr\}$$

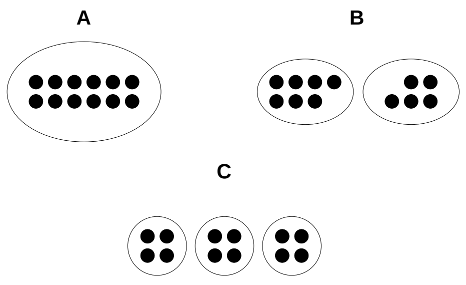


Figure 2: Some Distributions satisfying Specification $GROUPS\{12, 4, 1\}$

The first argument is the number of processes and second is the minimum number of processes in a local group. In the third argument it can be specified a restriction for the final number of the local groups such that the given number must be a divisor of the number of the local groups (e.g.: we can specify that the number of the group must be an even number).

Example Let us regard the following specification,

$$GROUPS\{12, 4, 1\}$$

which requires to schedule 12 concurrent processes into some local groups mapped to a heterogeneous grid environment, where each local group consists of 4 processes at least. The third argument is 1, thus we do not specify any restriction for the final number of the local groups.

There are many possible distributions which fulfill this requirements (some candidates are depicted in Figure 2). The scheduling mechanism attempts to find an available local network environment (cluster or LAN) first, where all the processes can be executed (see Figure 2A). If it is not possible, the scheduling mechanism attempts to find a distribution which takes into account the minimum number of groups and fits to the current physical grid architecture (see Figure 2B and C).

3.3 Graph

The schema *graph* is a similar structure to the schema groups, but here we can define the number of groups (first argument) and the precise number of the processes in each group respectively; furthermore links between any two groups (as edges of a graph) can be specified.

$$GRAPH\{nrOfGroups, nrOfLinks, listOfGroupSizes, listOfLinks\}$$

In the case of predefined links, the scheduling mechanism takes care of the placing of the groups compared to each other, such that the groups supposed to be connected to each other are scheduled on the physical grid architecture close to each other (in terms of latency and bandwidth) Of course, this notation does not mean that only those groups can interact each others which are bound to each other by such predefined links (it is just for adjusting some additional preferences for the scheduling mechanism).

Example a: In the following specification, we intend to schedule 18 processes, but we do not define any link

$$GRAPH\{3, 0, [5, 6, 7], [[]]\}$$

The maximum number of groups specified by the first argument is 3 in this case. First, the scheduling mechanism attempts to schedule all processes into same local network environment. If it is not feasible, it tries to organize the processes either into two groups or into three groups, where the size of groups is determined by the second, third and fourth arguments (in case of two groups the size of the one group is correspond to the sum of any two of these arguments).

Example b: The next specification is similar to the previous one, but this time we define some links among some processes

$$GRAPH\{4, 3, [5, 6, 7, 6], [[0, 1], [0, 2], [0, 3]]\}$$

The scheduling mechanism attempts to place the first group such that it can interact all the other groups as optimal as possible (in terms of latency).

3.4 Multi Level Parallelism – Tree

For specifying a multi-level manager-worker structure, in which there are some local managers connected one or more global managers (e.g.: because of some scalability issue), the schema *tree* is going to be used:

$$TREE\{nrOfProcs, depth, minSizeOfLeafGroups\}$$

The arguments are the following from left to right: number of processes (both workers and managers), the expected depth of the tree and the minimum number of worker processes in a local group.

We do not give directly a maximum number of the worker processes in one group, but we can specify precisely depth of the tree (e.g.: in order to

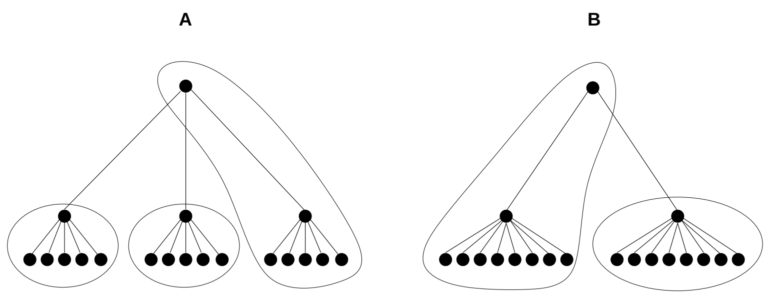


Figure 3: Some Distributions satisfying the Specification $TREE\{19, 3, 5\}$

control the scalability). For instance, in the case of depth 2, each worker will executed under the direction of one (local) manager in one local network environment (if such a distribution of the processes is not applicable on a particular grid architecture, then scheduling is unsuccessful). In the case of depth 3, the scheduler mechanism attempts to divide the workers into 2 groups at least (one local manager is included to each group additionally), in the case of depth 4 minimum number of worker groups will be 4, etc (if the depth is equal to 1, then we are in the same situation as in case of schema singleton — one local group without any manager process).

Example If we take the following specification

$$TREE\{19, 3, 5\}$$

then we can say that similarly to the schema groups we have more than one possible tree distribution which fulfill the given requirements (see Figure 3). The scheduling mechanism tries to distribute this tree structure of processes as optimal as possible (first into one local network environment and if it is not possible then int two or three ones (as it is showed by the Figure 3A and B). In addition, it attempts to place the global managers close (in terms of latency and bandwidth) to its children processes.

3.5 An Addition: Ring

This schema is very similar to the schema groups presented in Section 3.2, but this time the groups compose a ring (each group has two neighbors):

$$RING\{nrOfProcs, minSizeOfGroups, divisorOfGroupNr\}$$

In the case of the schema *ring*, the scheduling mechanism takes care of the placing of the groups compared to each other, such that the groups supposed to be neighbors in the ring are scheduled on the physical grid architecture close to each other (in terms of latency and bandwidth).

4 The Scheduling Algorithm

The task of the scheduling mechanism is to find a partition of processes based on the given schema which can be mapped to the available hardware resources such that the assessed use of any slow communication channel is minimized.

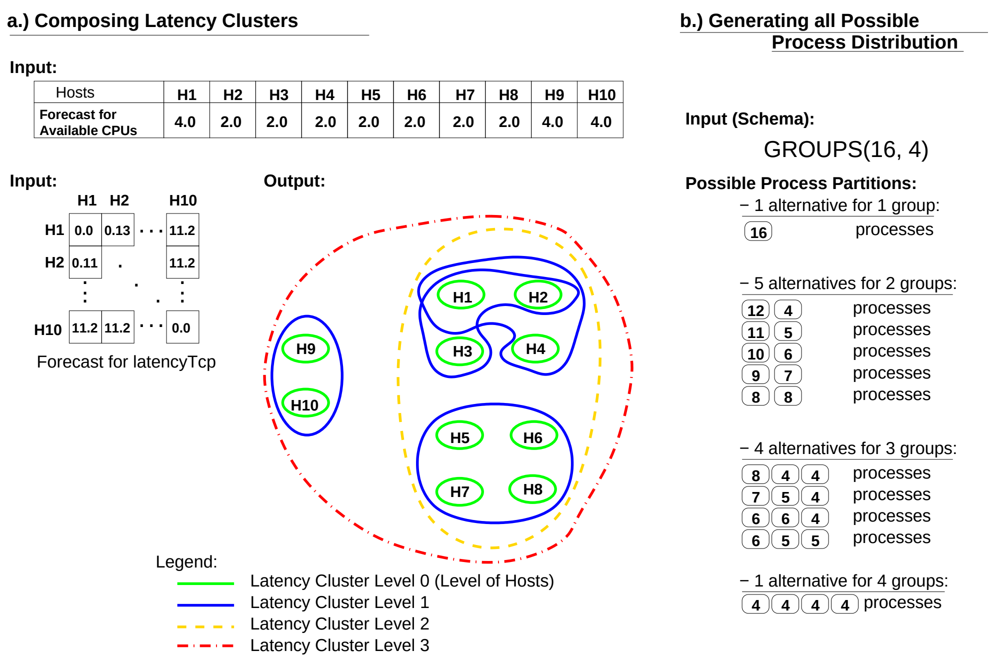
This kind of communication-aware mapping is an NP-complete problem which can be only efficiently solved by some kind of heuristic search algorithm. Similar problems have already arisen three decades ago in the mapping of processes to parallel hardware architectures (e.g.: hypercube) [10]. Nowadays the technique of communication-aware mappings is recalled in connection with heterogeneous multi-cluster and grid environments [13]. In this Section we discuss our solution for the problem of the communication-aware mapping in detail whose implementation was completely finished in the last project phase.

4.1 The Scheduling Algorithm in the Case of the Schema “Groups”

In this section, we describe how the algorithm applied by the scheduling mechanism works in the case of the schema “Groups”. The algorithm expects as input the list of the available hosts, a forecast for the available CPU fractions on these hosts and a forecast for the latency values in milliseconds are predicted for each pair of hosts, and finally a communication schema which specifies the preferred heterogeneous communication patterns of a program. The first three groups of data are provided by the NWS [14] while the schema is given by the user. The algorithm works roughly as follows:

1. First we classify all the links between each pair of hosts according to the order of magnitude of latencies. For the generated classes we assign an ascending sequence of integer numbers (*latency levels*). To the class which comprises the fastest links we assign the level 1, to the next one we assign the level 2 and so forth.

2. We compose some not necessarily disjoint clusters (let us call them *latency clusters*) from all the given hosts such that the latency levels of the links between any two member hosts of such a cluster cannot exceed a certain value (some of these latency clusters may comprise


a.) Composing Latency Clusters

Input:

Hosts	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Forecast for Available CPUs	4.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	4.0	4.0

Input:

	H1	H2	⋯	H10
H1	0.0	0.13	⋯	11.2
H2	0.11	.		11.2
⋮	⋮		.	⋮
H10	11.2	11.2	⋯	0.0

Forecast for latencyTcp

b.) Generating all Possible Process Distribution

Input (Schema):

GROUPS(16, 4)

Possible Process Partitions:

− 1 alternative for 1 group:
16 processes

− 5 alternatives for 2 groups:
12 4 processes
11 5 processes
10 6 processes
9 7 processes
8 8 processes

− 4 alternatives for 3 groups:
8 4 4 processes
7 5 4 processes
6 6 4 processes
6 5 5 processes

− 1 alternative for 4 groups:
4 4 4 4 processes

Figure 4: Composing Latency Clusters and Process Partitions

some others with less maximum latency level), see Figure 4a. Furthermore each host itself is regarded as a latency cluster with the latency level 0. Each latency cluster has a capacity feature which determines how many processes can be assigned to it at most. This capacity is calculated from the number of CPUs in the latency cluster multiplied with an integer coefficient. The default value of the coefficient is 1, but one can specify a higher value via a command line interface. The generated latency clusters are stored in a list which is sorted according to their maximal latency levels in ascending order (and on the same level according to their capacities in descending order).

3. We generate all those *partitioning* of processes (in which processes are organized into various local groups) which fulfil the given preferred communication pattern of a program, see Figure 4b.

4. Finally we map the generated process partitions to some latency clusters according to some compound heuristic (which helps to avoid the combinatorial explosion of possibilities) which roughly works as follows:

   - The process partitions are pre-evaluated. If there exist a latency cluster whose capacity is greater than or equal to the minimal

size of groups (given by the schema) and less than or equal to the maximal size of groups (calculated from the schema) then:
    - only those partitions are kept for the mapping which either contains only one group or
    - they have at least one group whose size is equal to the capacity of one of the latency clusters (independently from the latency values the optimal mappings always contain at least one group which fits exactly into a latency cluster).
- For the mapping the latency clusters are stored in a list in which they are arranged according to their latency level in an ascending order; and on the same latency levels according their capacity in a descending order in the list. A process partition is mapped to some latency clusters group by group (greater groups are assigned earlier). Each group is assigned to a latency cluster whose latency level is minimal and available capacity is large enough for the group. According to some additional low level heuristics a group can be assigned more than one latency cluster if their latency levels are the same (this can result an alternative mapping for a particular partition).

To find a reasonably efficient scheduling for the program in the space of solutions, we associate a cost function to each mapping (between a process partition and some latency clusters). This cost function takes into consideration the following characteristics of the mappings:

- the maximum latency level within the local groups,
- the maximum and the average latency values of all possible links among the local groups.

The algorithm always returns the mapping whose associated cost function is minimal.

4.2 The Scheduling Algorithm in the Case of the Schema "Graph"

In the case of the schema "Graph" the algorithm is slightly different because the number and sizes of local groups are fixed by the given schema. So we count only with the given process partition and we can therefore skip the third step of the algorithm above.

Additionally since the schema "Graph" specifies links among the local groups, in the cost function (in step 4) we apply average and maximum latency values of the pre-defined connections instead of all connections among the groups.

4.3 The Scheduling Algorithm in the Case of the "Tree" and the "Ring" Schemas

Although the number and the sizes of the local groups are not specified in the cases of the "Tree" and the "Ring" schema, but each possible partition contains pre-defined connections among its local groups. Hence, we apply in the cost function (in step 4) the average and maximum latency values of the pre-defined connections instead of all connections among the groups.

Furthermore, in the case of the schema "Tree" we take into account that every local manager process shall be scheduled together with the corresponding leaf group (they are mapped to the same latency cluster).

4.4 Disadvantage of the Algorithm

The algorithm assumes that on each host of a grid architecture an NWS sensor runs and the latency is measured among all sensors pairwise. This all-to-all network sensor communication would consume a considerable amount of resources (both on the individual host machines and on the interconnection network). For instance, the most common way to measure the end-to-end performances in a grid architecture comprising 15 hosts is to periodically conduct the $15^2 - 15 = 210$ network probes required to match all possible sensor pairs [14]. This problem can be overcome with a careful, network topology dependent configuration of the Network Weather Service (by establishing a corresponding clique hierarchy).

5 Installation Guide

5.1 Pre-requisites

The current version of our distributed programming tool requires the following installed softwares and settings on each participating grid machines:

- the user must own a user certificate issued by a corresponding CA;
- the grid services belonging to Globus Toolkit pre-Web service architecture [2] must be installed;
- the XML C parser library of Gnome called *libxml2* [5] must be installed;

- the MPICH-G2 [12, 11], which is a grid-enabled implementation of the MPI standard, must be installed;
- the library `bin/` of MPICH-G2 (with its full path) must be given in the system environment variable `$PATH`; and
- on that grid machine where the user intends to type MPICH-G2's mpirun (the local grid machine where the user logged in e.g.: via SSH), she must do one of the following at least once before running her application(s):
  - `source $GLOBUS_LOCATION/etc/globus-user-env.csh` or
  - `. $GLOBUS_LOCATION/etc/globus-user-env.sh`

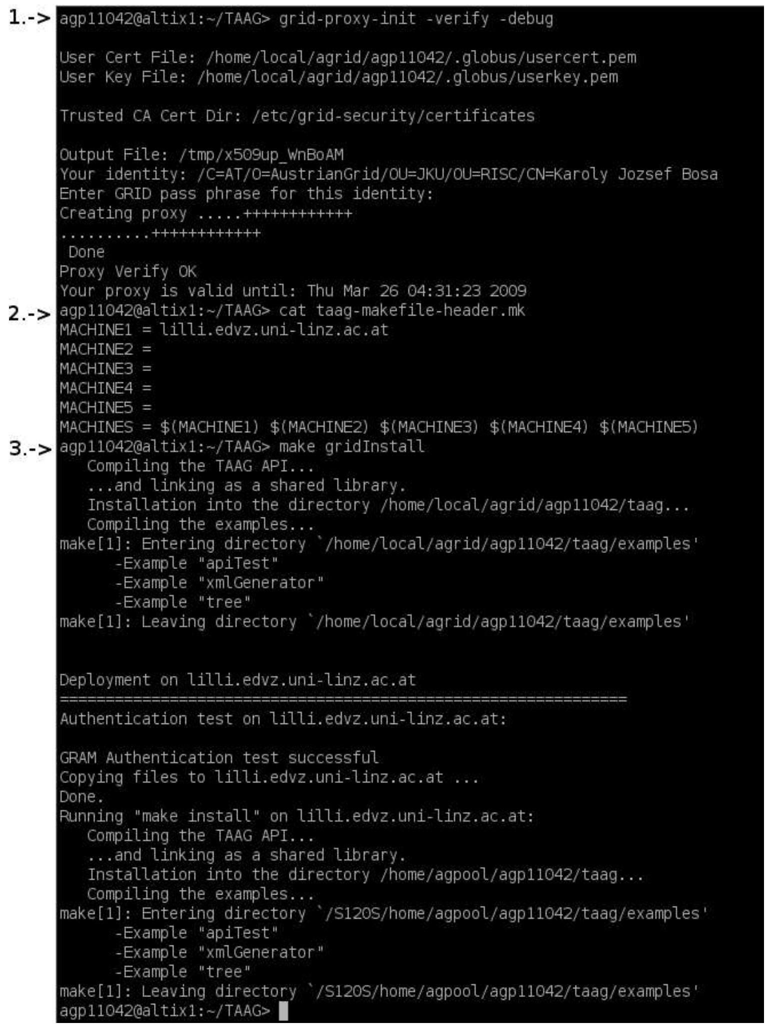
5.2 "Easy to Use" Deployment Mechanism for Evaluation and Testing

If one cannot or does not want to deploy our programming framework permanently on several grid nodes, but she would like to use it (or try it out at least), we provide for this purpose an "Easy to Use" deployment procedure. By this procedure a user can compile and install our distributed programming framework on the local and several remote grid sites at once. This "Easy to Use" deployment procedure requires only a list about those machines which fulfill the conditions described in Section 5.1.

5.2.1 Deployment Steps

After a user logged in to a machine where the Globus Toolkit is installed and she uploaded and unpacked the tarball of our TAAG software framework into a directory on this machine (*TAAG* is the abbreviation of the term *"**T**opology-**A**ware **A**PI for the **G**rid"*), she can apply our "Easy to Use" deployment procedure which facilitates the installation of our software on several grid machines. This deployment procedure consists of the following three steps (see Figure 5):

1. The user must generate her user proxy certificate with the globus command `grid-proxy-init` (if she has not done it before).
2. The user must enumerate some grid sites with fully specified host names on which she intends to execute her applications (except the local machine). For this, she must enter into the directory to where our software



```
agp11042@altix1:~/TAAG> grid-proxy-init -verify -debug

User Cert File: /home/local/agrid/agp11042/.globus/usercert.pem
User Key File: /home/local/agrid/agp11042/.globus/userkey.pem

Trusted CA Cert Dir: /etc/grid-security/certificates

Output File: /tmp/x509up_WnBoAM
Your identity: /C=AT/O=AustrianGrid/OU=JKU/OU=RISC/CN=Karoly Jozsef Bosa
Enter GRID pass phrase for this identity:
Creating proxy .....++++++++++++
..........++++++++++++
 Done
Proxy Verify OK
Your proxy is valid until: Thu Mar 26 04:31:23 2009
agp11042@altix1:~/TAAG> cat taag-makefile-header.mk
MACHINE1 = lilli.edvz.uni-linz.ac.at
MACHINE2 =
MACHINE3 =
MACHINE4 =
MACHINE5 =
MACHINES = $(MACHINE1) $(MACHINE2) $(MACHINE3) $(MACHINE4) $(MACHINE5)
agp11042@altix1:~/TAAG> make gridInstall
   Compiling the TAAG API...
   ...and linking as a shared library.
   Installation into the directory /home/local/agrid/agp11042/taag...
   Compiling the examples...
make[1]: Entering directory `/home/local/agrid/agp11042/taag/examples'
       -Example "apiTest"
       -Example "xmlGenerator"
       -Example "tree"
make[1]: Leaving directory `/home/local/agrid/agp11042/taag/examples'


Deployment on lilli.edvz.uni-linz.ac.at
==============================================================
Authentication test on lilli.edvz.uni-linz.ac.at:

GRAM Authentication test successful
Copying files to lilli.edvz.uni-linz.ac.at ...
Done.
Running "make install" on lilli.edvz.uni-linz.ac.at:
   Compiling the TAAG API...
   ...and linking as a shared library.
   Installation into the directory /home/agpool/agp11042/taag...
   Compiling the examples...
make[1]: Entering directory `/S120S/home/agpool/agp11042/taag/examples'
       -Example "apiTest"
       -Example "xmlGenerator"
       -Example "tree"
make[1]: Leaving directory `/S120S/home/agpool/agp11042/taag/examples'
agp11042@altix1:~/TAAG> █
```

Figure 5: The "Easy to Use" Deployment in Action

framework was unpacked and open the header file `taag-makefile-header.mk`. The list of the grid nodes can be given in the variable `MACHINES` in the opened file, e.g.:

```
MACHINE1 = host.name.1
MACHINE2 = host.name.2
...
MACHINE_N = host.name.n

MACHINES = $(MACHINE1) $(MACHINE2) ... $(MACHINE_N)
```

Note: If the user would like to deploy her own application(s) together with the TAAG software framework to the given grid machines, she should place her application(s) in the directory `applications` located directly under the directory to where our software was unpacked. The directory `applications` must directly contain a `makefile`, in which the "make target" `all` is responsible for the compilation of the corresponding user application(s) (the specified user application(s) will be deployed and compiled on each grid machine during the next step into a directory called `taagApps` located directly under the user's home).

3. Finally the user should issue the command `make gridInstall` in the same directory where the file `taag-makefile-header.mk` mentioned above resides. Then it can be followed on the screen how the TAAG software framework is deployed first to the local machine, then to all given remote grid machines. On each machine (including the local one), the software is deployed into a directory `taag` located directly under the user's home. After this step the user is able to use our software framework in a real grid environment and to execute the example programs or her application(s) (based on the TAAG programming framework), see Section 5.3 and Section 5.5.

After the user finished her work and would like to clean up the grid resources, she can issue the command `make gridClean`. This command removes every installed instance of our software from each grid machine.

5.3 Execution Framework "*taagrun*"

The software tool "*taagrun*" is the implementation of the Deployment Mechanism described in Section 2 and it is based on the starting mechanism of the grid-enabled MPI implementation MPICH-G2 [12]. It is located under the directory `bin` in the tree hierarchy of the software package. It expects an XML-based mapping file as an argument and starts an application on the grid according to the content of the mapping file (executable name, location, grid resources, distribution of processes, etc) in two steps:

- First, it distributes the mapping file into the directory `/tmp` on all the specified grid machines,
- Then, it generates an RSL script from the mapping file and with the help of this script it starts the application on the grid via MPICH-G2.

If the mapping file does not specify any grid resources within the XML tag `topology`, the program attempts to execute the given application on the `localhost`. Additionally, the user can apply the argument `-dumprsl`:

```
        taagrun -dumprsl <mapping_file.xml>
```

In this case the program only generates a RSL script from the given mapping file and prints it out on the standard output.

5.4 The Scheduler Implementation "*taag*"

The program requires a running NWS service on the grid. ...

```
    taag [options...] <SCHEMA file> [program arguments...]

    Options:
     -help              This help.
     -run               Run executable on the grid according
                        to the generated mapping.
     -dumprsl           Displays only the generated RSL file
                        on the standard output.
     -ppc n             Specify the maximum number of
                        scheduled "Processes Per CPU".
                        The default is 1.
     -log file          Create a logfile about the mapping.
     -nwsns hostname    NWS nameserver. The default is
                        localhost.
     -nwsport port      The port of the NWS nameserver.
                        The default is 8090.

[program arguments...] take effect iff the option "-run" is
given.
```

Some examples for the XML-based schema files can be found in the directory `examples`. ...

5.5 How to Execute the Example Programs

The current distribution of our software framework comprises three example programs which are located under directory `examples` (the "Easy to Use" deployment procedure [7] deploys and compiles these sources, too):

apiTest It simply presents the usage of the statements of our API in succession (the output depends on the given mapping file). The program can be started by the command "`../../bin/taagrun test.xml`" from its directory.

broadcastExample It sends broadcast messages round in a ring between neighbor groups in two steps (the ring is always composed from some groups of processes and its structure is described in the given mapping file). In the first step the root of every even group (local groups with even group rank) sends broadcast to all elements of its right neighbor group. In the second step the root of every odd group (local groups with odd group rank) sends broadcast to all elements of its right neighbor group. The program can be started by the command "`../../bin/taagrun ringWith6Groups.xml`" from its directory. The complete source code of this example is described together with some additional information and comments in Appendix E.

tree This program establishes a tree structure of processes where numerous tasks are distributed by the root process (via the non-leaf processes), elaborated by the leaf processes and finally the results of task are collected by the root (via the non-leaf processes again). The program works in case of various tree structures with 2, 3, 4, ... any levels depending what kind of tree structure is described in the mapping file. The program can be started by the command "`../../bin/taagrun treeWith3levelsOnlocalhost.xml`" from its directory. The complete source code of this example is described together with some additional information and comments in Appendix C.

Of course every example can be executed with different mapping files. The XML-based mapping files should be written directly by the user at the moment. In a later project phase, these XML files will be automatically generated by the Scheduling Mechanism (see Section 2).

6 Conclusions

Summarizing the achievements of the existing topology-aware programming tools (e.g.:MPICH-G2), we can say that they make available the given topology information on the level of their programming API and they optimize (only) the collective communication operations (e.g.: broadcast) with the help of the topology information such that they minimize the usage of the slow communication channels. But they are still not able to adapt the point-to-point communication pattern of a parallel programs to network topologies such that they achieve a nearly optimal execution time on the grid.

Compared to these existing topology-aware programming tools, the major advantages of our solution are the following:

- It takes into consideration the point-to-point communication pattern of a MPI parallel program and tries to fit it to a heterogeneous grid network architecture,
- It preserves the achievements of the already existing topology-aware programming tools. This means the topology-aware collective operations of MPICH-G2 are still available, since MPICH-G2 serves as a basis for our software framework.
- Since our system hides low-level grid-related execution details from the application by providing an abstract execution model, it eliminates some algorithmic challenges of the high-performance programming on the dynamic and heterogeneous grid environments. Programmers need to deal only with the particular problems which they are going to solve (like in a homogeneous cluster environments).
- The distribution of the processes is always conformed to the loading of the network resources.

A drawback of our solution is that the applicable communication patterns cannot be retrieved from the programs. If some schema is not enclosed to a distributed application, its effective scheduling may not be possible at the moment. We propose to overcome this issue in a subsequent version of our software system where the programmer will be forced by the API library to specify a recommended schema (with defined flexibility) via some function calls in the source of the programs. According to our conception, the scheduling mechanism will be able to query this built-in information from the compiled application.

As the next step, we intend to replace the MPICH-G2 in our software framework with its successor called MPIg [4]. By this substitution, our TAAG system will be able to submit and execute parallel programs via the Web-Service architecture of the Globus Toolkit, too (MPIg had only one internal release at the end of 2007, which is freely available for testing and development purposes). Besides, we also plan to develop on the basis of our TAAG programming framework some grid-distributed parallel applications (e.g.: a distributed n-body simulation based on Barnes-Hut algorithm and some other programs in the fields of the hierarchical distributed genetic algorithms) in cooperation with other research groups.

Acknowledgement

The work described in this paper is partially supported by the Austrian Grid Project [1], funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ 4003/2-VI/4c/2004.

Appendix

A The Finalized Topology-Aware API

The implemented API is an addition to the MPICH [3] programming library based on the MPI standard and its purpose is to inform a parallel program

- how its processes assigned to some physical grid resources and to certain virtual hierarchies (e.g.:groups, tree, etc.) and
- which are the designated roles for these processes.

All these are performed according to the XML-based execution plan file (which was generated by the scheduling mechanism). A detailed description of the refined API is presented below. All calls described in this chapter are **completely implemented** and tested.

A.1 Header File

taag.h header file is required for all programs/routines which intend to use any calls of our API (*TAAG* is the abbreviation of the term *"**T**opology-**A**ware **A**PI for the **G**rid"*).

A.2 Format of the API Calls

int rc = TAAG_Xxxxx (parameter, ...) is the general format of the calls defined our API. All of them return an integer error code. If the call was successful, the return value is equal to the constant `TAAG_SUCCESS (= 0)`. The follow error codes are defined in the API at present:

- `TAAG_ERR_INIT (= 1)` TAAG library was not initialized.
- `TAAG_ERR_MPI (= 2)` MPI environment was not initialized.
- `TAAG_ERR_FILE (= 3)` Invalid file name.
- `TAAG_ERR_ARG (= 4)` Invalid argument.

- `TAAG_ERR_SCHEMA (= 5)` Invalid schema type.
- `TAAG_ERR_PRANK (= 6)` Invalid process rank.
- `TAAG_ERR_GRANK (= 7)` Invalid group rank.
- `TAAG_ERR_COUNT (= 9)` Invalid argument count.
- `TAAG_ERR_HWTOPOLOGY (= 10)` The hardware topology is not given in the XML mapping file.
- `TAAG_ERR_MEMORY (= 14)` Unsuccessful memory allocation.
- `TAAG_ERR_XML (= 15)` Incorrect XML mapping file.
- `TAAG_ERR_UNKNOWN (= 20)` Unknown error.

A.3 Calls wrt. Initialization and Termination

TAAG_Init (char *exec_plan_file) allocates and initializes the corresponding data structures according to the generated execution plan file given in the argument. If the argument is null then the latest available execution plan is taken. If there is none, then the call returns an error code which is different as `TAAG_SUCCESS`. This function must be called in every program, must be called before any other TAAG functions and must be called only once in a program.

TAAG_Initialized (int *flag) indicates whether TAAG_Init has been called. It returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`).

TAAG_Free () deallocates the data structures used by the API library.

A.4 Calls wrt. the Program Structure Group

Program structure called group is not correspond to object MPI_Group. **The group always consists of some processes which are located on the same local physical infrastructure (same host or same LAN)**. Each group has a unique rank assigned by our library when the group is initialized. A *group rank* (similarly to the process ranks) is an integer number and it can take any value from the domain which runs from 0 to the number of groups minus one. Processes within a group have a predefined order.

TAAG_Group_number (int *nr) returns the number of the groups.

TAAG_Group_rank (int rank, int *grpRank) requires a process rank as input and returns the rank of the group which belongs to this process.

TAAG_Group_size (int grpRank, int *nr) requires a group rank as input and returns the number of member processes of the group.

TAAG_Group_members (int grpRank, int nr, int *members) requires a group rank and the maximum size of the vector given in the third argument as input and returns the ranks of all member processes of the group.

TAAG_Group_MPIGroup (int nrProcs, int *ranks, int nrGroups, int *grpRanks, MPI_ Group *grp) requires the number and the enumeration of some process ranks; furthermore the number and the enumeration of some group ranks as input and it composes a MPI_Group structure from all given processes (included the processes of the given groups as well). This call is useful if some MPI collective operations are going to be used within among the given processes and groups. The order of processes in the MPI_Group object is: given processes first (in the given order) then the members of the given groups (in the given order of groups).

A.4.1 Convenient Calls Derived from the Call `TAAG_Group_graph`

TAAG_Group_degree (int grpRank, int order, int *nr) requires a group rank and the order (distance) of the queried neighbors as input. If the execution plan defines predefined links among the groups (which specify the groups that are planned to interact each other) then this call returns the number of the n-th order "neighbor" groups of the given one. For instance, if the second argument is equal to:

- **-1 :** this call returns the number of groups which are unreachable from the given one;
- **0 :** this call returns with 1 (the given group itself is the only one whose distance is 0 - no reason for this call);
- **1 :** this call returns the number of groups which are the (1st order) neighbors of the given one;

- **2 :** this call returns the number of groups which are the 2nd order neighbors of the given one;
- **...** and so on and forth.

TAAG_Groups_neighbours (int grpRank, int order, int nr, int *list) requires a group rank, the order (distance) of the queried neighbors (see the description of the call TAAG_Group_degree above) and the maximum size of the vector given in the fourth argument and as input. If the execution plan defines predefined links among the groups (which specify the groups that are planned to interact each other) then this call returns a list of the ranks of the n-th order "neighbor" groups of the given one.

TAAG_Group_distance (int grpRank1, int grpRank2, int *nr) returns the number of edges (the shortest distance) between the two groups in the predefined graph. If there is now a predefined way between the two groups in the graph the third argument returns the value -1.

TAAG_Group_way (int grpRank1, int grpRank2, int nr, int *list) requires ranks of two groups and the maximum size of the vector given in the forth argument as input and returns the list of the ranks of the intermediate groups (along the shortest path) included grpRank2 at the end. If there is now a predefined way between the two groups in the graph the forth one returns a NULL.

A.4.2 Convenient Calls Derived from the Call `TAAG_Group_Members`

TAAG_Group_element `(int grpRank, int index, int *rank)` requires a group rank as input and integer n number (where *index* can be a value taken from an integer domain runs from 0 to the number of processes in the group minus one) and returns the rank of the *index*–th process in the given group.

A.5 Calls wrt. Program Structure Tree

The following section presents some calls which are related to the program structure tree. All these calls are convenient calls which can be implemented by application of the calls presented in Section A.4.

In a tree, the rank of the root process (and the rank of the root group as well) is always `TAAG_ROOT (= 0)`. The leaves which belong to the same parent compose a group. Furthermore, each non-leaf process is wrapped into a one element group (but its group rank is not necessarily corresponds to the rank of the non-leaf process). Hence, every call presented in Section A.4 above can be applied in a tree. Each execution plan can describe one tree at most.

TAAG_Tree_isTree (int *flag) indicates whether given execution plan (file) describes a tree structure. It returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`).

TAAG_Tree_depth (int *levels) returns the depth (number of the levels) of the tree.

TAAG_Tree_level (int rank int *level) requires a process rank and returns on which level the given process is located on the tree. The level of the root is 0 and the level of the leaves is $depth - 1$.

TAAG_Tree_width (int rank, int level, int *nr) requires process rank which specifies root of a subtree, a level in the specified subtree (or degree of children) and it returns the width of the subtree on the given level in terms of processes (or 0 if the given level number is not applicable on the given subtree).

Comment: the tree structure described by the execution plan may be unbalanced, it can therefore be useful to know e.g.: the number of the available workers on particular subtree).

TAAG_Tree_children (int rank, int level, int nr, int *procs) requires process rank which specifies root of a subtree, a level in the specified subtree (or degree of children) and the maximum size of the vector given in the forth argument (the maximum width of the tree on the given level) and it returns a list of the ranks of the successor processes located on the given level in the specified subtree.

TAAG_Tree_parent (int rank, int *parent) requires a process rank and returns the rank of the parent of the given process.

A.5.1 Further Convenient Calls

The following three calls can be derived from the calls `TAAG_Tree_level` and `TAAG_Get_Tree_depth`.

TAAG_Tree_root (int *rank) returns the rank of the root process.

TAAG_Tree_isLeaf (int rank, int *flag) requires a process rank and indicates whether the given process is located on the level on *depth* $- 1$ in the tree. The call returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`).

TAAG_Tree_isLocalManager (int rank, int *flag) requires a process rank and indicates whether the given process is located on the level on *depth*$-$ 2 in the tree. The call returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`). The local managers are specials in the sense that they are always located on the same local physical infrastructure (Host or LAN) as their children.

A.6 Calls wrt. Program Structure Ring

This section contains some convenient calls wrt. program structure ring of groups. These calls can be derived from the calls presented in Section A.4

TAAG_Ring_isRing (int *flag) indicates whether given execution plan (file) describes a ring structure. It returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`).

TAAG_Ring_left (int grpRank1, int *grpRank) requires a group rank and it returns rank of the left neighbor group of the given group.

TAAG_Ring_right (int grpRank1, int *grpRank2) requires a group rank as input and it returns rank of the right neighbor group of the given group.

A.7 Calls wrt. Topology Structure

TAAG_Topology_given (int *flag) indicates whether given execution plan (file) describes a network topology. It returns a flag as either logical true (`TAAG_TRUE = 1`) or false (`TAAG_FALSE = 0`).

TAAG_CPU_number (int *nr) returns the number of the allocated CPUs.

TAAG_Host_number (int *nr) returns the number of the allocated hosts.

TAAG_Host_properties (int rank, int *nrCPUs, int *nrProcs) requires a process rank as input and returns the number of CPUs and the number of processes on the host where the given process is located. NULL pointer can be applied for the second and third arguments.

TAAG_Host_processes (int rank, int nr, int *ranks) requires a process rank and the maximum number of the vector given in the third argument as input and it returns the ranks of all the processes residing on the same host.

TAAG_Host_latency (int rank1, int rank2, double *latency) requires the ranks of two processes as input and returns the average latency value between two hosts on which the given processes reside. If the given two processes are located on the same host, then the call returns with 0.

TAAG_Host_address (int rank, char *address) requires a process rank as input and returns the address of the host where the process resides. The size of the given char vector should be equal to the constant `TAAG_MAX_HOSTNAME_STRING`.

TAAG_LAN_number (int *nr) returns the number of LANs whose resources are used in the current session.

TAAG_LAN_properties (int rank, int *nrHosts, int *nrCPUs, int *nrProcs) requires a process rank as input and returns the number of hosts, the number of CPUs and the number of processes in the LAN where the given process is located. NULL pointer can be applied for the last three arguments.

TAAG_LAN_processes (int rank, int nr, int *ranks) requires a process rank and the maximum number of the vector given in the third argument as input and it returns the ranks of those processes which are executed on the same LAN.

TAAG_Comm_level (int rank1, int rank2, int *commlevel) requires two process ranks as input and returns on which network level they can communicate with each other.

- If the call returns with `TAAG_WAN_LEVEL (= 0)` then the two processes can interact each other only via WAN,

- but if the call returns with `TAAG_LAN_LEVEL (= 1)` they are located in the same LAN network and
- if the call returns with `TAAG_HOST_LEVEL (= 2)` they nest on the same host.

B The XML-based Execution Plan

The scheduling mechanism performs the mapping between the generalized communication structure of program and the topology of a physical grid architecture and provides an XML-based execution plan as an output. This execution plan is required for the deployment mechanism and the Topology-Aware API.

The XML-based mapping language composed for describing execution plans consists of the following XML tags:

<mapping> is the root element of the XML-based description.

<executable> is a child element of the element <`mapping`> and it contains the file name of the executable of the corresponding application.

<applicationId> is a child element of the element <`mapping`> and it specifies an unique identifier for the mapping description. This identifier should be the same if an application is executed more than once on the same grid environment with the same parameters within a certain time interval.

<timeStamp> is a child element of the element <`mapping`> and it defines validity this particular mapping description.

<graph> is a child element of the element <`mapping`> and it describes how the processes are organized into a particular logical (undirected graph) structure.

<type> is a child element of the element <`graph`> and it specifies the type of the given structure (in the current version of the program, the type can be a "tree", a "ring" or a "graph").

<group> is a child element of the element <`graph`> and it defines a local group of processes (a local group is always correspond to vertex of the given logical structure of processes). Its attribute `''id''` defines a unique rank for the group

<**process**> is a child element of the element <`group`> and it contains a unique process rank (this tag can occur as child element of the element <`host`> as well, see below).

<**edge**> is a child element of the element <`graph`> and it has two attributes `''oneEndPoint''` and `''otherEndPoint''`. This element specifies an edge in the described logical structure of processes by connecting two local groups (referred by their identifiers).

<**topology**> is a child element of the element <`mapping`> and it describes how the corresponding processes are distributed on the allocated grid resources.

<**lan**> is a child element of the element <`topology`> and it enumerates some grid resources which belong to the same local network.

<**host**> is a child element of the element <`lan`> and it describes the features of a particular grid site and enumerates the particular processes scheduled to it.

<**hostname**> is a child element of the element <`host`> and it gives the hostname of the current host.

<**CPUs**> is a child element of the element <`host`> and it gives the number of CPUs on the current host.

<**directory**> is a child element of the element <`host`> and it gives the home directory of the user (how requested the mapping from the scheduling mechanism) on the current host.

<**process**> is a child element of the element <`host`> and it contains a unique process rank. The given process is scheduled to the current host.

<**latency**> is a child element of the element <`topology`> and it has two attributes `''row''` and `''column''`. This element contains a matrix provided by NWS software which contains average latency values between any two allocated hosts.

In Appendix D an example execution plan (mapping description) is presented which has already been employed to run a simple distributed application (based on our supercomputing API) on the architecture of the Austrian Grid.

C Example for Multilevel Parallelism

The following example represents the usage of the proposed tree-related calls described in Section A.5. The source code discussed below is an updated and corrected version of the one presented in [6]. This program has been tested together with the prototype version of our supercomputing API with different number of processes executed on the grid sites `altix1.jku.austriangrid.at` (Altix 350) and `lilli.edvz.uni-linz.ac.at` (Altix 4700).

The given example program itself (without any modification in its source code) can be used to establish different kinds of the tree-like multilevel parallelism on the grid

- which can be organized into arbitrary levels,
- which can comprise various number of processes and
- which can be split to local groups of processes in various manners (depending of the actual available hardware resources).

For instance, one possibility for the distribution of processes is the xml-based execution plan example described in Section D, with which this source code can be conjugated before its execution (Mentioned example consists of 20 processes organized into a 3 level tree structured and deployed on two grid sites).

In this program a global manager process distributes some computational tasks among its child processes. If these processes are not a leaf/worker processes then in turn they distribute these tasks among their children further until the tasks reach the worker/leaf processes at the bottom of the tree structure. After a worker accomplished a task it sends back to the global manager through its local manager.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include <taag.h>

#define MSG_SIZE 160
#define NR_OF_TASKS  500
#define BUFFER_SIZE NR_OF_TASKS+2
#define EXIT_SIGNAL "EXIT"
#define XML_DEFAULT "first_tree.xml"
```

```
void create_task(int i, char* s) {
   sprintf(s,"TASK%d",i);
}

void process_task(int rank, char* s, char* t) {
   int group_rank;
   TAAG_Group_rank(rank, &group_rank);
   sprintf(t, "%s is processed by  %d in group %d.",
          s, rank, group_rank);
}

int main(int argc, char *argv[]) {
   int nrProcs, rc, flag, nrChildren;
   int rank, root, parent;

   int children[100];
   char outbuff[BUFFER_SIZE][MSG_SIZE];
   char inbuff[BUFFER_SIZE][MSG_SIZE];

   int indx_recv = 0;
   int child_indx = 0;
   int indx = 0;

   MPI_Request reqs[BUFFER_SIZE];
   MPI_Status stats[BUFFER_SIZE];
   MPI_Request req;
   MPI_Status stat;

   int triggeredExit = TAAG_FALSE;

   rc = MPI_Init(&argc,&argv);
   if (rc != MPI_SUCCESS) {
      printf("Error starting MPI program.\n");
      MPI_Abort(MPI_COMM_WORLD, rc);
   }
   MPI_Comm_size(MPI_COMM_WORLD,&nrProcs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   if (argc > 1) {
      rc = TAAG_Init(argv[1]);
```

```
}
else {
   rc = TAAG_Init(XML_DEFAULT);
}

if (rc != TAAG_SUCCESS) {
   fprintf(stderr,"Error (code %d) initializing
        the TAAG structure on process %d.\n",
        rc, rank);
   MPI_Finalize();
   return rc;
} //if

TAAG_Tree_isTree(&flag);
if (!flag) {
   fprintf(stderr, "Error (code %d) the given
        schema is NOT a tree.\n",
        TAAG_ERR_SCHEMA);
   TAAG_Free();
   MPI_Finalize();
   return TAAG_ERR_SCHEMA;
}

TAAG_Tree_root(&root);

if (rank == root) {
/***** root branch *****/
   TAAG_Tree_width(rank, 1, &nrChildren);
   TAAG_Tree_children(rank, 1, nrChildren, children);

   for (int i = 0; i < NR_OF_TASKS; i++) {
      create_task(i, outbuff[i]);
      MPI_Irecv(inbuff[i], MSG_SIZE,
           MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
           MPI_COMM_WORLD, &reqs[i]);
      MPI_Isend(outbuff[i], MSG_SIZE,
           MPI_CHAR, children[child_indx], i,
           MPI_COMM_WORLD, &req);
      child_indx++;
      if (child_indx == nrChildren) child_indx = 0;
   }
```

```
      MPI_Waitall (NR_OF_TASKS, reqs, stats);
      for (int i = 0; i < NR_OF_TASKS; i++) {
         printf("%s\n", inbuff[i]);
      } //for

      strcpy(outbuff[NR_OF_TASKS], EXIT_SIGNAL);
      for (int i = 0; i < nrChildren; i++) {
         MPI_Send(outbuff[NR_OF_TASKS], MSG_SIZE,
              MPI_CHAR, children[i], 0,
              MPI_COMM_WORLD);
      }
   } //if
   else {
   /***** non-root branch *****/
      TAAG_Tree_parent(rank, &parent);
      TAAG_Tree_isLeaf(rank, &flag);
      if (flag == TAAG_FALSE) {
      /***** non-leaf branch *****/
         TAAG_Tree_width(rank, 1, &nrChildren);
         TAAG_Tree_children(rank, 1, nrChildren, children);

         MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
              MPI_CHAR, parent, MPI_ANY_TAG,
              MPI_COMM_WORLD, &reqs[indx_recv]);
         while(!triggeredExit) {
            MPI_Wait(&reqs[indx_recv], &stat);
            indx = indx_recv;
            indx_recv++;
            if (!strcmp(inbuff[indx], EXIT_SIGNAL)) {
               triggeredExit = TAAG_TRUE;
               for (int i=0; i < nrChildren; i++) {
                  MPI_Send(inbuff[indx], MSG_SIZE,
                       MPI_CHAR, children[i], stat.MPI_TAG,
                       MPI_COMM_WORLD);
               } //for
            } //if
            else {
               MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
                    MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &reqs[indx_recv]);
               if (stat.MPI_SOURCE == parent) {
                  MPI_Isend(inbuff[indx], MSG_SIZE,
                       MPI_CHAR, children[child_indx],
                       stat.MPI_TAG, MPI_COMM_WORLD, &req);
                  child_indx++;
                  if (child_indx == nrChildren) child_indx = 0;
               }
               else {
                  MPI_Isend(inbuff[indx], MSG_SIZE,
                       MPI_CHAR, parent, stat.MPI_TAG,
                       MPI_COMM_WORLD, &req);
               } //else
            } //else
         } //while
      } //if
      else {
      /***** leaf branch *****/
         MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
              MPI_CHAR, parent, MPI_ANY_TAG,
              MPI_COMM_WORLD, &reqs[indx_recv]);
         while(!triggeredExit) {
            MPI_Wait(&reqs[indx_recv], &stat);
            if (!strcmp(inbuff[indx_recv], EXIT_SIGNAL)) {
               triggeredExit = TAAG_TRUE;
            } //if
            else {
               indx = indx_recv;
               indx_recv++;
               MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
                    MPI_CHAR, parent, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &reqs[indx_recv]);
               process_task(rank,
                            inbuff[indx],
                            outbuff[indx]);
               MPI_Isend(outbuff[indx], MSG_SIZE,
                    MPI_CHAR, parent, stat.MPI_TAG,
                    MPI_COMM_WORLD, &req);
            } //else
         } //while
      } //else
   } //else
```

```
129:    TAAG_Free();
130:    MPI_Finalize();
131:    return 0;
132: }
```

Comments:

lines 001–005 comprise the required includes.

line 006 defines the constant `MSG_SIZE`, which is the maximum size of the MPI messages.

line 007 defines the constant `NR_OF_TASKS`, which is the number of tasks distributed among the processes.

line 008 defines the constant `BUFFER_SIZE`, which is the maximum number of messages can be stored in a message buffer used in this program.

line 009 defines the constant `EXIT_SIGNAL`, which is a predefined message. If a process receives this message it finishes its execution.

line 010 defines the constant `XML_DEFAULT`, which is a filename. This file name is used if no command line argument is given for the program.

lines 011–013 define a function called *create_task* which returns a string description of the subsequent computational task.

line 014–018 define a function called *process_task* whose input is a previously mentioned task description and whose output is the outcome of this task (in string format).

lines 040–045 allocate and initialize the corresponding data structures according to the execution plan comprised by the given mapping file.

line 051 checks whether the given execution plan describes a program structure “tree”.

line 058 determines the root process of the tree hierarchy.

lines 059–078 describe the behavior of the root process of the tree. In line 061 it determines number and in line 062 the rank of its children processes. Then it generates a given number of computational tasks, distributes them among its children and waits for the results. If all results were received, it prints out them. Finally, root starts to disseminate a message `EXIT_SIGNAL` to its each child and it finishes its execution.

line 081 determines the parent process of the current non-root process in the tree.

line 082 decides whether the current process is a leaf in the tree.

lines 084–110 describe the behavior of the intermediate scheduler processes in the tree. It determines number and the rank of children of the current process. Then local scheduler is blocked, until a message is received. If the message was sent by its parent process it forwards it to one of its children. Otherwise, it forwards it to its parent. If the local scheduler receives a message `EXIT_SIGNAL`, it forwards this message to its all children, then it finishes its execution.

lines 113–128 describe the behavior of the leaf processes in the tree. A leaf is blocked, until a computational task arrives in a message from its parent. Then it processes the task and sends the result back to its parent. If a message `EXIT_SIGNAL` is received, the leaf process finishes its execution.

line 129 deallocates the data structures applied by our library.

D Example for the XML-based Execution Plan

The example presented below is an XML-based description of an execution plan for the grid application described in Section C. The name of the executable of the application is given between the XML tags `<executable>`.

An execution plan usually consists of two major parts:

- the part given between the XML tags `<graph>` describes how the processes are organized into higher-level logical structures (e.g.: how the processes are clustered to groups, and how these groups are planned to interact with each other, etc.),
- the part given between the XML tags `<topology>` describes how the processes are distributed on the physical resources on the grid (e.g.: which processes are executed on which grid hosts, what are the working directories on these hosts, etc.).

The given example describes a three level tree structures which is composed by 20 processes executed on the grid sites `altix1.jku.austriangrid.at` (Altix 350) and `lilli.edvz.uni-linz.ac.at` (Altix 4700), see Figure 6.

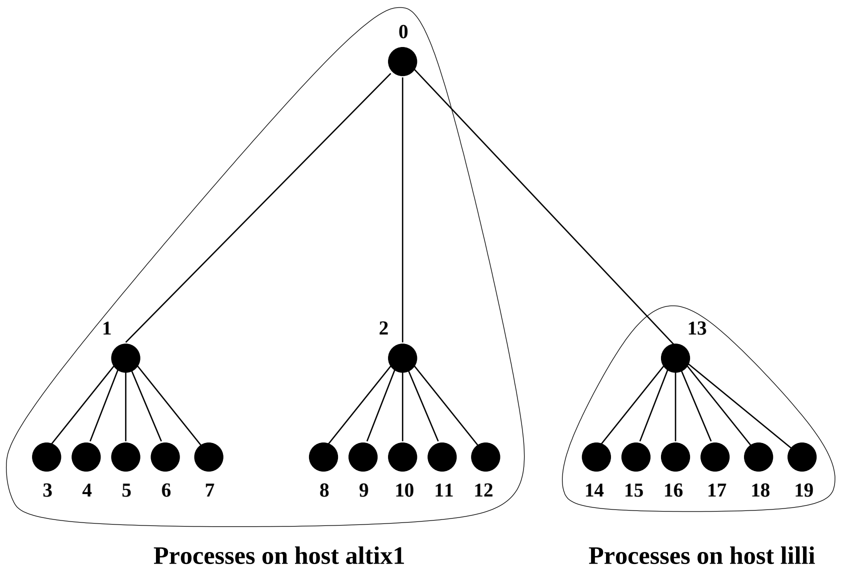


Figure 6: The execution plan given by the XML source below

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping SYSTEM "../../etc/taagXML.dtd">
<mapping>
  <executable>taagTree</executable>
  <applicationId>treeWith3levels</applicationId>
  <timeStamp>00000000</timeStamp>
  <graph>
    <type>tree</type>
    <group id="0">
      <process>0</process>
    </group>
    <group id="1">
      <process>1</process>
    </group>
    <group id="2">
      <process>2</process>
    </group>
    <group id="3">
      <process>3</process>
      <process>4</process>
      <process>5</process>
      <process>6</process>
      <process>7</process>
    </group>
    <group id="4">
      <process>8</process>
      <process>9</process>
      <process>10</process>
      <process>11</process>
      <process>12</process>
    </group>
    <group id="5">
      <process>13</process>
    </group>
    <group id="6">
      <process>14</process>
      <process>15</process>
      <process>16</process>
      <process>17</process>
      <process>18</process>
      <process>19</process>
    </group>
    <edge oneEndPoint="0" otherEndPoint="1"/>
    <edge oneEndPoint="0" otherEndPoint="2"/>
    <edge oneEndPoint="0" otherEndPoint="5"/>
    <edge oneEndPoint="1" otherEndPoint="3"/>
    <edge oneEndPoint="2" otherEndPoint="4"/>
    <edge oneEndPoint="5" otherEndPoint="6"/>
  </graph>
  <topology>
    <lan id="0">
      <host id="0">
        <hostname>altix1.jku.austriangrid.at</hostname>
        <CPUs>16</CPUs>
        <directory>$(HOME)/taag/examples/tree</directory>
        <process>0</process>
        <process>1</process>
        <process>2</process>
        <process>3</process>
        <process>4</process>
        <process>5</process>
        <process>6</process>
        <process>7</process>
        <process>8</process>
        <process>9</process>
        <process>10</process>
        <process>11</process>
        <process>12</process>
      </host>
      <host id="1">
        <hostname>lilli.edvz.uni-linz.ac.at</hostname>
        <CPUs>128</CPUs>
        <directory>$(HOME)/taag/examples/tree</directory>
        <process>13</process>
        <process>14</process>
        <process>15</process>
        <process>16</process>
        <process>17</process>
        <process>18</process>
        <process>19</process>
      </host>
    </lan>
    <latency row="0" column="0">0.0</latency>
    <latency row="0" column="1">1.0</latency>
    <latency row="1" column="1">0.0</latency>
  </topology>
</mapping>
```

Note: The XML tag `<latency>` will be used in our second prototype version to provide information about the average latency between two given hosts, but it is not used at the moment. Hence, we have just adjusted its value to one second in the case of two distinct hosts.

E Example How to Use MPI Collective Operation among Groups and Processes

The following example program presents how to perform MPI collective operations among local groups and single processes, too. The source code discussed below is an corrected and updated version of the one presented in [6]. This program was tested with different number of processes on the grid sites `altix1.jku.austriangrid.at` (Altix 350) and `lilli.edvz.uni-linz.ac.at` (Altix 4700).

This program is an artificial example that assumes its processes partitioned to even number of local groups which compose a ring. Such a communication pattern can be specified by the schema *ring* [6] as follows:

$$RING\{nrOfProcs, minSizeOfGroups, \mathbf{2}\}$$

In the first two arguments the number of processes and the minimum size of the local groups is given. The third argument is a restriction for the scheduling mechanism such that its value must always be a divisor of the number of the local groups (in our example this third argument is 2 because the program requires even number of local groups).

The program sends broadcast messages round in the ring between neighbor groups in two steps:

- In the first step the root process of every even group (local group with even group rank) sends a broadcast to all elements of its right neighbor group.
- In the second step the root process of every odd group (local group with odd group rank) sends a broadcast to all elements of its right neighbor group.

Since we apply MPICH-G2 as an underlying software architecture, the performed broadcast operations are topology aware [12], too.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <mpi.h>
#include <taag.h>

#define MSG_SIZE 160
#define XML_DEFAULT "ringWith6Groups.xml"

void create_message(int rank, int grp, char* s) {
   sprintf(s,"MESSAGE FROM P%d (from G%d)",rank, grp);
}

void process_message(int step, int rank, int grp, char* s) {
   printf("IN STEP %d: P%d (from G%d) RECEIVED: \"%s\".\n",
               step, rank, grp, s);
}
```

```
int main(int argc, char *argv[]) {
   int nrProcs, rc, flag;
   int rank, localRootRank, remoteRootRank;
   int grpRank, leftGrpRank, rightGrpRank;
   MPI_Comm mpiComm1, mpiComm2;
   MPI_Group mpiGrp1, mpiGrp2;

   char buff[MSG_SIZE];

   rc = MPI_Init(&argc,&argv);
   if (rc != MPI_SUCCESS) {
      printf("Error starting MPI program.\n");
      MPI_Abort(MPI_COMM_WORLD, rc);
   }
   MPI_Comm_size(MPI_COMM_WORLD,&nrProcs);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   if (argc > 1) {
      rc = TAAG_Init(argv[1]);
   }
   else {
      rc = TAAG_Init(XML_DEFAULT);
   }

   if (rc != TAAG_SUCCESS) {
      fprintf(stderr,"Error (code %d) initializing the TAAG
                    structure on process %d.\n", rc, rank);
      MPI_Finalize();
      return rc;
   } //if

   TAAG_Ring_isRing(&flag);
   if (!flag) {
      fprintf(stderr, "Error (code %d) the given schema is
             NOT a ring.\n", TAAG_ERR_SCHEMA);
      TAAG_Free();
      MPI_Finalize();
      return TAAG_ERR_SCHEMA;
   } //if
```

```
TAAG_Group_rank(rank, &grpRank);

TAAG_Ring_right(grpRank, &rightGrpRank);
TAAG_Ring_left(grpRank, &leftGrpRank);

/****************** First Step ***********************/
if (grpRank % 2 == 0) { //even group rank
   TAAG_Group_element(grpRank, 0, &localRootRank);
   TAAG_Group_MPIGroup(1, &localRootRank, 1, &rightGrpRank,
          &mpiGrp1);
   if (rank == localRootRank) create_message(rank, grpRank,
          buff);
} //if
else { //odd group rank
   TAAG_Group_element(leftGrpRank, 0 , &remoteRootRank);
   TAAG_Group_MPIGroup(1, &remoteRootRank, 1, &grpRank,
          &mpiGrp1);
} //else

//MPI_Comm_create is a collective operation
MPI_Comm_create(MPI_COMM_WORLD, mpiGrp1, &mpiComm1);

//sending/receiving broadcast from the root of each even
//groups to all element of its right neighbor
if (mpiComm1 != MPI_COMM_NULL) { /* not every process is
                                 involved in the broadcast */
   MPI_Bcast(buff, MSG_SIZE, MPI_CHAR, 0, mpiComm1);
   process_message(1, rank, grpRank, buff);
}

/****************** Second Step ***********************/
if (grpRank % 2 == 1) { //odd group rank
   TAAG_Group_element(grpRank, 0, &localRootRank);
   TAAG_Group_MPIGroup(1, &localRootRank, 1, &rightGrpRank,
          &mpiGrp2);

   if (rank == localRootRank) create_message(rank, grpRank,
          buff);
} //if
else { //even group rank
   TAAG_Group_element(leftGrpRank, 0 , &remoteRootRank);
074:       TAAG_Group_MPIGroup(1, &remoteRootRank, 1, &grpRank,
                  &mpiGrp2);
075:    } //else

076:    //MPI_Comm_create is a collective operation
077:    MPI_Comm_create(MPI_COMM_WORLD, mpiGrp2, &mpiComm2);

078:    //sending/receiving broadcast from the root of each odd
        //groups to all element of its right neighbour
079:    if (mpiComm2 != MPI_COMM_NULL) { /* not every process is
                                     involved in the broadcast */
080:       MPI_Bcast(buff, MSG_SIZE, MPI_CHAR, 0, mpiComm2);
081:       process_message(2, rank, grpRank, buff);
082:    }

083:    if (mpiComm1 != MPI_COMM_NULL) MPI_Comm_free(&mpiComm1);
084:    if (mpiComm2 != MPI_COMM_NULL) MPI_Comm_free(&mpiComm2);
085:    MPI_Group_free(&mpiGrp1);
086:    MPI_Group_free(&mpiGrp2);

087:    TAAG_Free();
088:    MPI_Finalize();
089:    return 0;
090: }
```

Comments:

lines 001–005 comprise the required includes.

line 006 defines the constant `MSG_SIZE`, which is the maximum size of the MPI messages.

line 007 defines the constant `XML_DEFAULT`, which is a filename. This file name is used if no command line argument is given for the program.

lines 008–010 define a function called `create_message` which generates a string. The string will be sent in a broadcast.

lines 011–013 define a function called `process_message` which writes out its string argument together with some additional information on the standard output.

lines 028–033 allocate and initialize the corresponding data structures according to the mapping file comprised by the given file.

line 039 checks whether the given mapping file describes a program structure "ring".

line 046 determines the rank of the group in which the current process is involved.

line 047 determines the rank of the right neighbor group of the current group.

line 048 determines the rank of the left neighbor group of the current group.

First Step:

lines 050-054 are executed only on the processes of EVEN local groups.

line 052 composes a MPI group on each process of every EVEN local group, which comprises the root process (the first element) of the current group and all processes of the right neighbor group (the order of the processes in the created MPI group is always the following: first the given processes in the given order, then the processes of the given group the given order).

line 053 generates a string message on the root process of the current group.

lines 055-058 are executed only on the processes of ODD local groups.

line 057 composes a MPI group on each process of every ODD local group, which comprises all processes of the current group and the root process (the first element) of the left neighbor group.

line 059 establishes some MPI communicators according to the previously created MPI groups from the MPI_COMM_WORLD. Attention, the statement `MPI_Comm_create` is a collective operation (concerning the communicator given in its first argument), therefore, all processes must perform it even those of them which are not involved in the created MPI groups.

lines 062-065 check whether the current process involved in the given communicator. If it is, then a broadcast is performed within this communicator (from its first process to its all processes) and the received message will be displayed by the function `process_message`.

Second Step:

lines 067-071 are executed only on the processes of ODD local groups.

line 069 composes a MPI group on each process of every ODD local group, which comprises the root process (the first element) of the current group and all processes of the right neighbor group.

line 070 generates a string message on the root process of the current group.

lines 072-075 are executed only on the processes of EVEN local groups.

line 074 composes a MPI group on each process of every EVEN local group, which comprises all processes of the current group and the root process (the first element) of the left neighbor group.

line 077 establishes some MPI communicators according to the previously created MPI groups from the MPI_COMM_WORLD. Attention, the statement `MPI_Comm_create` is a collective operation (concerning the communicator given in its first argument), therefore, all processes must perform it even those of them which are not involved in the created MPI groups.

lines 079-082 check whether the current process involved in the given communicator. If it is, then a broadcast is performed within this communicator (from its first process to its all processes) and the received message will be displayed by the function `process_message`.

lines 083-086 free the created MPI structures (MPI groups and MPI communicators).

line 087 deallocates the data structures applied by our software framework.

References

[1] Austrian Grid Project Home Page. `http://www.austriangrid.at`.

[2] Globus Toolkit. `http://www.globus.org/toolkit/`.

[3] MPICH Project Home Page. `http://www-unix.mcs.anl.gov/mpi/mpich1/`.

[4] MPIg Release Home. `ftp://ftp.cs.niu.edu/pub/karonis/MPIg/`.

[5] The XML C Parser and Toolit of Gnome. `http://xmlsoft.org/`.

[6] Karoly Bosa and Wolfgang Schreiner. Initial Design of a Distributed Supercomputing API for the Grid. Austrian Grid Deliverable AG-D4-2-2008_1, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, September 2008.

[7] Karoly Bosa and Wolfgang Schreiner. A Prototype Implementation of a Distributed Supercomputing API for the Grid. Austrian Grid Deliverable AG-D4-1-2009_1, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, March 2009.

[8] Karoly Bosa and Wolfgang Schreiner. A Supercomputing API for the Grid. In Jens Volkert et al., editor, *Proceedings of 3rd Austrian Grid Symposium 2009*, pages –. Austrian Grid, Austrian Computer Society (OCG), September 28-29 2009. To Appear.

[9] Karoly Bosa and Wolfgang Schreiner. Report on the Second Prototype of a Distributed Supercomputing API for the Grid. Austrian Grid Deliverable AG-D4-2-2009_1, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, September 2009.

[10] Woei-Kae Chen and Edward. F. Gehringer. A Graph-Oriented Mapping Strategy for a Hypercube. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 200–209, New York, NY, USA, 1988. ACM.

[11] W. Gropp and E. Lusk. Installation and Users Guide to MPICH, a Portable Implementation of MPI Version 1.2.7p1 The globus2 device for Grids. `http://www-unix.mcs.anl.gov/mpi/mpich1/docs/mpichman-globus2/mpichman-globus2.htm`.

[12] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, May 2003.

[13] Juan Manuel Orduña, Federico Silla, and José Duato. On the Development of a Communication-Aware Task Mapping Technique. *J. Syst. Archit.*, 50(4):207–220, 2004.

[14] Rich Wolski, Neil T. Spring, and Jim Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.