Austrian Grid

A Prototype Implementation of a Distributed Supercomputing API for the Grid

Document Identifier:	**AG-D4-1-2009_1.pdf**
Status:	**Public**
Workpackage:	**4**
Partners:	**Research Institute for Symbolic Computation (RISC)**
Lead Partner:	**RISC**
WP Leaders:	**Wolfgang Schreiner (RISC)**

Delivery Slip

	Name	Partner	Date	Signature
From				
Verified by				
Approved by				

Document Log

Version	Date	Summery of changes	Author
1	31.03.2009	Initial Version	K. Bosa, W. Schreiner

A Prototype Implementation of a Distributed Supercomputing API for the Grid

Karoly Bosa
Wolfgang Schreiner

Research Institute for Symbolic Computation (RISC)
Johannes Kepler University Linz
{Karoly.Bosa, Wolfgang.Schreiner}@risc.uni-linz.ac.at

March 31, 2009

A Prototype Implementation of a Distributed Supercomputing API for the Grid

Károly Bósa   Wolfgang Schreiner

March 31, 2009

Abstract

We proposed to participate in the Austrian Grid Phase 2 within the frame of the activity "Grid Research". Our goal is to develop a distributed software framework and an API for grid computing which shall empower applications to perform scheduling decisions on their own, utilizing the information about the grid environment in order to adapt the algorithmic structure to the particular situation. The planned solution will be able to eliminate some algorithmic challenges of nowadays grid programming. Since the last report we implemented a prototype version of the proposed API and successfully tested on some resources of the Austrian Grid. In this paper, we present the current state of the implementation and envisage the next development steps.

1 Introduction

We proposed to participate in the Austrian Grid Phase 2 [1] within the frame of the activity "Grid Research". We deals with development of a distributed programming software framework and API for grid computing. This work in particular assists applications whose algorithmic structures do not lend themselves to a decomposition into big sequential components whose only interactions occur at the begin and the end of the execution of a component and that can be scheduled by a meta-level grid *workflow* language that implements communication between components by file-based mechanisms. Rather the planned solution empowers applications to perform scheduling decisions on their own, utilizing the information provided by the API about the grid environment at hand in order to adapt the algorithmic structure to the particular situation.

However, no application can execute efficiently on the grid that is not aware of the fact that it does not run in a homogeneous cluster environment with low-latency and high-bandwidth connectivity between all pairs of nodes, but in an environment with heterogeneous nodes. Correspondingly, the API does not hide this fact from the application but reflect the information provided by a grid management and execution environment to the programming language level such that the application can utilize this information and adapt its behavior to it, e.g., by mapping closely interacting activities to nodes within a network and minimizing communication between activities executing on nodes in different networks.

The proposed API however hides low-level execution details from the application by providing an abstract execution model that in particular allows to initiate activities and communicate between them independent of their physical location. The execution engine maps these abstract model features to the actual topology of the allocated physical grid resources.

In [6] we gave an overview on the State of the Art. In [5] we presented a design of our proposed distributed programming framework in details. In the current project phase, we implemented the first prototype version of this programming tool. This document is an update of our previous report [5] and we focus on the achievements of this first prototype version in it.

Section 2 gives a short overview on the updated design of our distributed programming framework. In Section 3, we discuss some implementation issues and we report on the current state of the implementation. In Section 4, we present a newly composed simple XML-based language which is used for describing the mapping between a generalized communication structure of a distributed application (in heterogeneous networks) and a particular grid topology. In section 5, we envisage the next implementation goals.

Appendix A contains the detailed description of our improved and fully implemented topology-aware distributed programming API. Appendix B gives an example code for the usage of the API mentioned above and represents its versatility (this distributed application together with the implemented API has already been tested on the architecture of the Austrian Grid). Appendix C presents an example for an XML-based mapping description between some physical hardware resources of Austrian Grid and some program processes organized into a logical structure (the example program in Appendix B was distributed and executed on the Austrian Grid resources among others according to this mapping description).

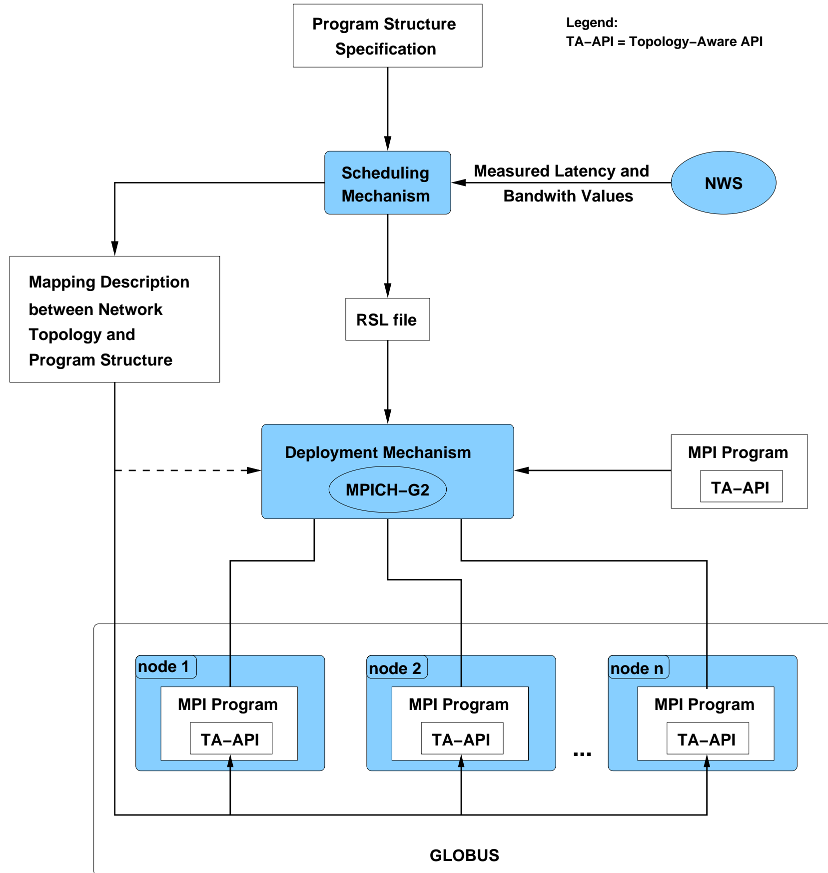

Figure 1: Overview of the Proposed Software System

2 An Overview on the Design

As we described in [5] our software framework consists of three major components (see Figure 1):

Scheduling Mechanism depends on the *Network Weather Service (NWS)* [9], which provides from time to time the information about the available CPUs and about actual latency (and perhaps bandwidth) of the communication channel between any two grid nodes. The Scheduling Mechanism attempts to classify the available computing resources (CPUs) according to the measured latency (and bandwidth) values and to build up a three level network hierarchy (intra-machine interactions, LAN interactions, slowest (WAN) interactions). This classification is refined periodically.

In our approach, the user assign to each given parallel program a pre-

defined *schema* [5] specifying a preferred communication structure of the program in heterogeneous network environments. Before each execution of a parallel program on the grid (either on the same physical grid architecture or a different one), the scheduling mechanism adapts and maps the generalized communication structure of the program to the composed topological hierarchy of the physical grid architecture such that it minimizes the assessed execution time. The output will be an *execution plan* in a XML-based file (describing a mapping between the network topology and a determined program structure, see Section 4) for the Deployment Mechanism and the Topology-Aware API.

Deployment Mechanism is based on the MPICH-G2 [8, 7] starting mechanism (gridified `mpirun` for Globus). First it generates *Resource Specification Language (RSL)* file from the XML-based execution plan provided by the Scheduling Mechanism and then it starts the processes of the given program on the corresponding grid nodes according to the content of this RSL file.

Topology-Aware API is an addition to the MPICH [3] programming library based on the MPI standard. The main purpose of this API is to assign the processes of a program to the allocated grid resources according to the execution plan generated by Scheduling Mechanism (each process must identify itself with a functional role assigned to its local grid node by the given execution plan).

3 Implementation

In this section, we report on the status of the implementation of our distributed programming framework called *"**T**opology-**A**ware **A**PI for the **G**rid" (TAAG).*

3.1 The Current State

In the current project phase, we implemented the first prototype version of the proposed software framework. Our current implementation based on the pre-Web Service architecture of Globus Toolkit [2] and grid-enabled MPI implementation MPICH-G2 and it includes the following parts:

- The main achievement accomplished in the current project phase is the complete implementation of the proposed Topology-Aware API.

The definition of API was also improved in the meantime. For the detailed description of this finalized API see Appendix A.

- We specified an XML-based language for describing the execution plan which is mapping between the network topology and the determined communication structure of a parallel program (see Section 4 an Appendix C)
- With the help of the *libxml2* C library [4] we implemented a parser for the composed XML-based mapping language (language of the execution plan) with which we then extended the implementation of our API.
- We also implemented a software utility which is able to generate RSL script from the XML-based execution plan. This program will be used by the proposed scheduling mechanism which will be implemented in the next project phase.
- We developed some simple distributed example applications for testing the functionality of our API and representing its versatility. In Appendix B we present such an example program which itself (without any modification in its source code) can be used to establish different kinds of the tree-like multilevel parallelism on the grid according to a given execution plan.
- Finally we worked out and implemented a preliminary version of an "Easy to Use" deployment mechanism, which facilitates the usage of our software framework in a real grid environment, see Section 3.3.

Our implemented supercomputing API together with all other software components listed above was successfully tested on the grid sites `altix1.jku.austriangrid.at` (Altix 350) and `lilli.edvz.uni-linz.ac.at` (Altix 4700).

3.2 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* [4] must be installed;

- the MPICH-G2 [8, 7], 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`

3.3 "Easy to Use" Deployment 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 in one step. This "Easy to Use" deployment procedure requires only a list about those machines which fulfill the conditions described in Section 3.2.

3.3.1 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, 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 2):

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

```
1.-> 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
2.-> 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)
3.-> 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 2: The “Easy to Use” Deployment in Action

```
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 `userApps` located directly under the directory to where our software was unpacked. This directory should contain a file called `Makefile` or `makefile`, in which the "make target" `all` is responsible for the compilation of the corresponding user application(s).

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 her application(s) (based on the TAAG programming framework) with the help of the command `mpirun -globusrsl` *`rsl.file`*.

The RSL (*R*esource *S*pecification *L*anguage) files which are used for the execution of any application can be generated from the XML-based execution plan files. But the XML-based execution plan files should be written by the user directly at the moment. In a later project phase, these XML and RSL files will be automatically generated by the proposed *scheduling mechanism* [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.

4 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.

<applicationName> 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 C 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.

5 Conclusions and Next Steps

We fulfilled all project goals appointed to the end of the current project phase and we presented the feasibility our originally proposed ideas and system design. The following developments steps are planned in the subsequent project phases:

April 2009 — September 2009 In this period, we work on the second skeleton prototype that will contain all the major components of the software system described in this document. We mainly focus on the development of the proposed scheduling mechanism (see Section 2) and improve the implementation of the deployment mechanism. We also plan to perform the first benchmarks with our software system and to present a preliminary efficiency analysis.

October 2009 — Based on our first experiences we refine the design and implement an initial version of the SOAP– and gridftp–based communication calls of the proposed API.

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.

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_MPIStructs (int nrProcs, int *ranks, int nrGroups, int *grpRanks, MPI_ Group *grp, MPI _Comm *comm) 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 and a MPI_Comm structures 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 or groups. null pointer can be applied for the last two arguments. 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 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 [5]. This program was 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 C, 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 their 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.

```
001: #include <stdlib.h>
002: #include <stdio.h>
003: #include <string.h>
004: #include <mpi.h>
005: #include <taag.h>
```

```
006: #define MSG_SIZE 160
007: #define NR_OF_TASKS  500
008: #define BUFFER_SIZE NR_OF_TASKS+2
009: #define EXIT_SIGNAL "EXIT"
010: #define XML_DEFAULT "first_tree.xml"

011: void create_task(int i, char* s) {
012:    sprintf(s,"TASK%d",i);
013: }

014: void process_task(int rank, char* s, char* t) {
015:    int group_rank;
016:    TAAG_Group_rank(rank, &group_rank);
017:    sprintf(t, "%s is processed by  %d in group %d.",
               s, rank, group_rank);
018: }

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

022:    int children[100];
023:    char outbuff[BUFFER_SIZE][MSG_SIZE];
024:    char inbuff[BUFFER_SIZE][MSG_SIZE];

025:    int indx_recv = 0;
026:    int child_indx = 0;
027:    int indx = 0;

028:    MPI_Request reqs[BUFFER_SIZE];
029:    MPI_Status stats[BUFFER_SIZE];
030:    MPI_Request req;
031:    MPI_Status stat;

032:    int triggeredExit = TAAG_FALSE;

033:    rc = MPI_Init(&argc,&argv);
034:    if (rc != MPI_SUCCESS) {
035:       printf("Error starting MPI program.\n");
036:       MPI_Abort(MPI_COMM_WORLD, rc);
```

```
037:    }
038:    MPI_Comm_size(MPI_COMM_WORLD,&nrProcs);
039:    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

040:    if (argc > 1) {
041:       rc = TAAG_Init(argv[1]);
042:    }
043:    else {
044:       rc = TAAG_Init(XML_DEFAULT);
045:    }

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

051:    TAAG_Tree_isTree(&flag);
052:    if (!flag) {
053:       fprintf(stderr, "Error (code %d) the given
              schema is NOT a tree.\n",
              TAAG_ERR_SCHEMA);
054:       TAAG_Free();
055:       MPI_Finalize();
056:       return TAAG_ERR_SCHEMA;
057:    }

058:    TAAG_Tree_root(&root);

059:    if (rank == root) {
060:    /***** root branch *****/
061:       TAAG_Tree_width(rank, 1, &nrChildren);
062:       TAAG_Tree_children(rank, 1, nrChildren, children);

063:       for (int i = 0; i < NR_OF_TASKS; i++) {
064:          create_task(i, outbuff[i]);
065:          MPI_Irecv(inbuff[i], MSG_SIZE,
                 MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                 MPI_COMM_WORLD, &reqs[i]);
```

```
066:         MPI_Isend(outbuff[i], MSG_SIZE,
                  MPI_CHAR, children[child_indx], i,
                  MPI_COMM_WORLD, &req);
067:         child_indx++;
068:         if (child_indx == nrChildren) child_indx = 0;
069:      }

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

074:      strcpy(outbuff[NR_OF_TASKS], EXIT_SIGNAL);
075:      for (int i = 0; i < nrChildren; i++) {
076:         MPI_Send(outbuff[NR_OF_TASKS], MSG_SIZE,
                  MPI_CHAR, children[i], 0,
                  MPI_COMM_WORLD);
077:      }
078:   } //if
079:   else {
080:   /***** non-root branch *****/
081:      TAAG_Tree_parent(rank, &parent);
082:      TAAG_Tree_isLeaf(rank, &flag);
083:      if (flag == TAAG_FALSE) {
084:      /***** non-leaf branch *****/
085:         TAAG_Tree_width(rank, 1, &nrChildren);
086:         TAAG_Tree_children(rank, 1, nrChildren, children);

087:         MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
                  MPI_CHAR, parent, MPI_ANY_TAG,
                  MPI_COMM_WORLD, &reqs[indx_recv]);
088:         while(!triggeredExit) {
089:            MPI_Wait(&reqs[indx_recv], &stat);
090:            indx = indx_recv;
091:            indx_recv++;
092:            if (!strcmp(inbuff[indx], EXIT_SIGNAL)) {
093:               triggeredExit = TAAG_TRUE;
094:               for (int i=0; i < nrChildren; i++) {
095:                  MPI_Send(inbuff[indx], MSG_SIZE,
                           MPI_CHAR, children[i], stat.MPI_TAG,
                           MPI_COMM_WORLD);
```

```
096:            } //for
097:         } //if
098:         else {
099:            MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
                     MPI_CHAR, MPI_ANY_SOURCE, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &reqs[indx_recv]);
100:            if (stat.MPI_SOURCE == parent) {
101:               MPI_Isend(inbuff[indx], MSG_SIZE,
                        MPI_CHAR, children[child_indx],
                        stat.MPI_TAG, MPI_COMM_WORLD, &req);
102:               child_indx++;
103:               if (child_indx == nrChildren) child_indx = 0;
104:            }
105:            else {
106:               MPI_Isend(inbuff[indx], MSG_SIZE,
                        MPI_CHAR, parent, stat.MPI_TAG,
                        MPI_COMM_WORLD, &req);
107:            } //else
108:         } //else
109:      } //while
110:   } //if
111:   else {
112:   /***** leaf branch *****/
113:      MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
               MPI_CHAR, parent, MPI_ANY_TAG,
               MPI_COMM_WORLD, &reqs[indx_recv]);
114:      while(!triggeredExit) {
115:         MPI_Wait(&reqs[indx_recv], &stat);
116:         if (!strcmp(inbuff[indx_recv], EXIT_SIGNAL)) {
117:            triggeredExit = TAAG_TRUE;
118:         } //if
119:         else {
120:            indx = indx_recv;
121:            indx_recv++;
122:            MPI_Irecv(inbuff[indx_recv], MSG_SIZE,
                     MPI_CHAR, parent, MPI_ANY_TAG,
                     MPI_COMM_WORLD, &reqs[indx_recv]);
123:            process_task(rank,
                             inbuff[indx],
                             outbuff[indx]);
124:            MPI_Isend(outbuff[indx], MSG_SIZE,
```

```
                          MPI_CHAR, parent, stat.MPI_TAG,
                          MPI_COMM_WORLD, &req);
125:             } //else
126:           } //while
127:         } //else
128:     } //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 initializes the corresponding data structures according to the execution plan comprised by the given 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.

C Example for the XML-based Execution Plan

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

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.:

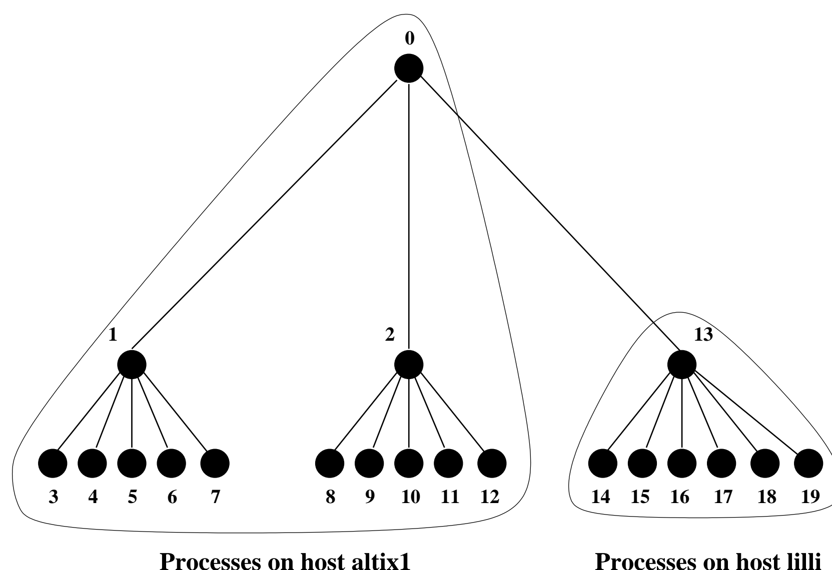

Figure 3: The execution plan given by the XML source below

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 3.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping SYSTEM "../../etc/taagXML.dtd">
<mapping>
  <applicationName>taagTree</applicationName>
  <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/local/agrid/agp11042/treeExample</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/agpool/app11042/treeExample</directory>
        <CPUs>128</CPUs>
        <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.

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.

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

[5] Karoly Bosa and Wolfgang Schreiner. Initial Design of a Distributed Supercomputing API for the Grid. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria., September 2008.

[6] Karoly Bosa and Wolfgang Schreiner. Report on the State of the Art Survey. Technical report, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria., March 2008.

[7] 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.

[8] 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.

[9] 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.