Austrian Grid

Report on the Second Prototype of a Distributed Supercomputing API for the Grid

Document Identifier:	**AG-D4-2-2009_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	28.09.2009	Initial Version	K. Bosa, W. Schreiner

Report on the Second Prototype 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

September 24, 2009

Report on the Second Prototype of a Distributed Supercomputing API for the Grid

Károly Bósa Wolfgang Schreiner

September 24, 2009

Abstract

We participate in the Austrian Grid Phase 2 within the frame of the activity "Grid Research". We deal with development of a distributed programming tool 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. Our goal is to design and implement a software framework and an API that can be used for developing grid-distributed parallel programs without leaving the level of the language in which the core application is written. The planned solution will be able to eliminate some algorithmic challenges of nowadays grid programming. In this paper, we report on the second prototype of our topology-aware software system extended with the description of our newly developed scheduling algorithm.

1 Introduction

No application can execute efficiently on the grid that is not aware of the fact that it runs in an heterogeneous network environment with heterogeneous nodes. We report on an ongoing work whose 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 and utilizing the information about the grid environment in order to adapt their algorithmic structure 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.

Regard an example where a user intends to execute a tree-like multi-level parallel algorithmic solution on the grid. She specifies in advance that

the given application should consist of 20 processes organized into a 3-levels tree structure. On the lowest level leaves belonging to the same parent process should 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 (nearly) optimal distribution of processes on the momentarily available grid resources and to start the processes according to this distribution. Furthermore, our API is able to apply at runtime a corresponding mapping between the predefined roles of processes in the specified hierarchy (global manager, local manager and workers) and the allocated pool of grid nodes such that it minimizes the execution time.

In [3] we outlined our idea, discussed the design of our software framework with some implementation issues and presented the first version of our Topology-Aware API. Then in [4] we reported on the implemented and functioning first prototype of our software system. Furthermore, we described the finalized API and explained the source code of a simple distributed example application which can be used to establish different kinds of the tree-like multilevel parallelism on the grid and which represents well the versatility of our API.

Now we report on an updated architecture of our software system, which contains some modifications and extensions in addition to the first prototype version [4]: Section 2 gives a short overview about the up-to-date software architecture. In Section 3, we focus on our newly developed scheduling algorithm which is currently under implementation and which is the last missing key component of our software framework. Section 4 deals with the extended execution framework of our system. Finally Appendix A describes a new example application which presents how to perform MPI collective operations among some single processes and some *local groups* of processes (this kind of structure was introduced by our communication schema based programming solution).

2 The Up-to-Date 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 [3]: 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

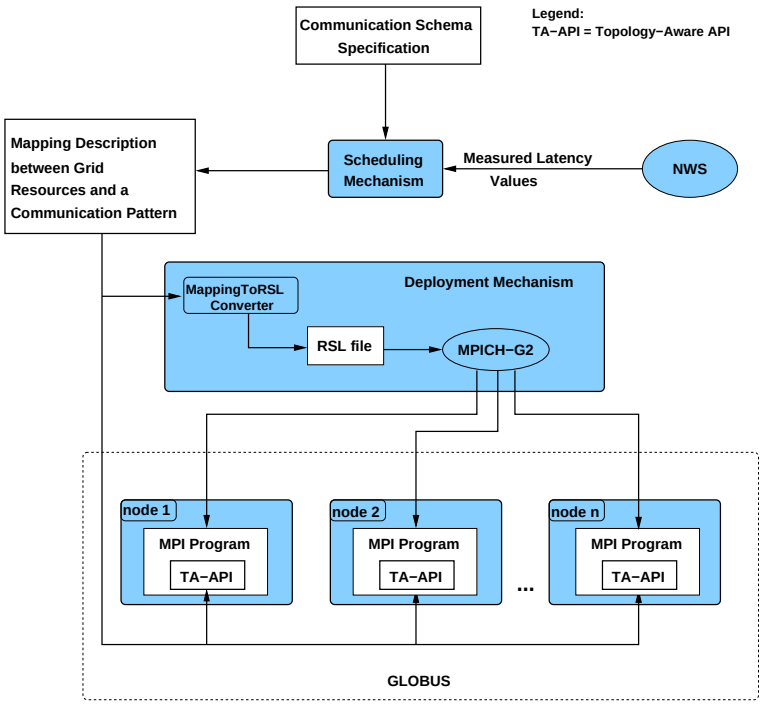

Figure 1: Overview on the Software Framework

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.

The second prototype of our software framework is still based on the pre-Web Service architecture of the Globus Toolkit [2] and MPICH-G2 [5]. Our solution consists of three major components (see Figure 1):

Scheduling Mechanism depends on the *Network Weather Service (NWS)* [6], which became a de facto standard in the grid community as a performance prediction tool. According to the forecasts values provided by NWS, the Scheduling Mechanism finds a nearly optimal mapping between the specified communication schema and the available grid resources.

Before the execution of a parallel program on the grid, the Scheduling Mechanism adapts and maps the preferred communication pattern of

the program to the available grid resources such that it heuristically minimizes the assessed execution time. The output is an XML-based *mapping file* (describing a mapping between the network topology and the given communication pattern) for the Deployment Mechanism and the Topology-Aware API. The algorithm applied by the Scheduling Mechanism is described in detail in Section 3. The final version of the second prototype will contain an implementation of the presented algorithm.

Deployment Mechanism is based on the starting mechanism of the grid-enabled MPI implementation MPICH-G2 [5]. In contrast to the previous version of our software framework, we do not use anymore any *Resource Specification Language (RSL)* script for starting programs which contained redundant information compared with the mapping file. Now we use the mapping file for this purpose, too. The Deployment Mechanism distributes the mapping file on the corresponding grid nodes and starts the processes of the given program on the grid according to the mapping file. For the description how to use this tool, see Section 4.

Topology-Aware API [4] is an addition to the MPICH programming library. Its purpose is to query mapping files and inform parallel programs how their processes are assigned to some physical grid resources and which are the designated roles for these processes, such as: in which local group a particular process is involved; which are the characteristics of local groups, of graphs (e.g.: neighborhoods of a group, distance of two groups, etc.), of trees (e.g.: depth of a tree, parent and children of a process, etc.) or of rings.

For representing the versatility of our API, we have developed some simple distributed example applications, see [4] and Appendix A.

Our supercomputing API and the Deployment Mechanism have already been tested successfully on the grid sites `lilli.edvz.uni-linz.ac.at` (Altix 4700) and `altix1.jku.austriangrid.at` (Altix 350).

3 The Scheduling Algorithm

The task of the Scheduling Mechanism is to find a partitioning of processes based on the given schema which can be mapped nearly optimally to the available hardware resources.

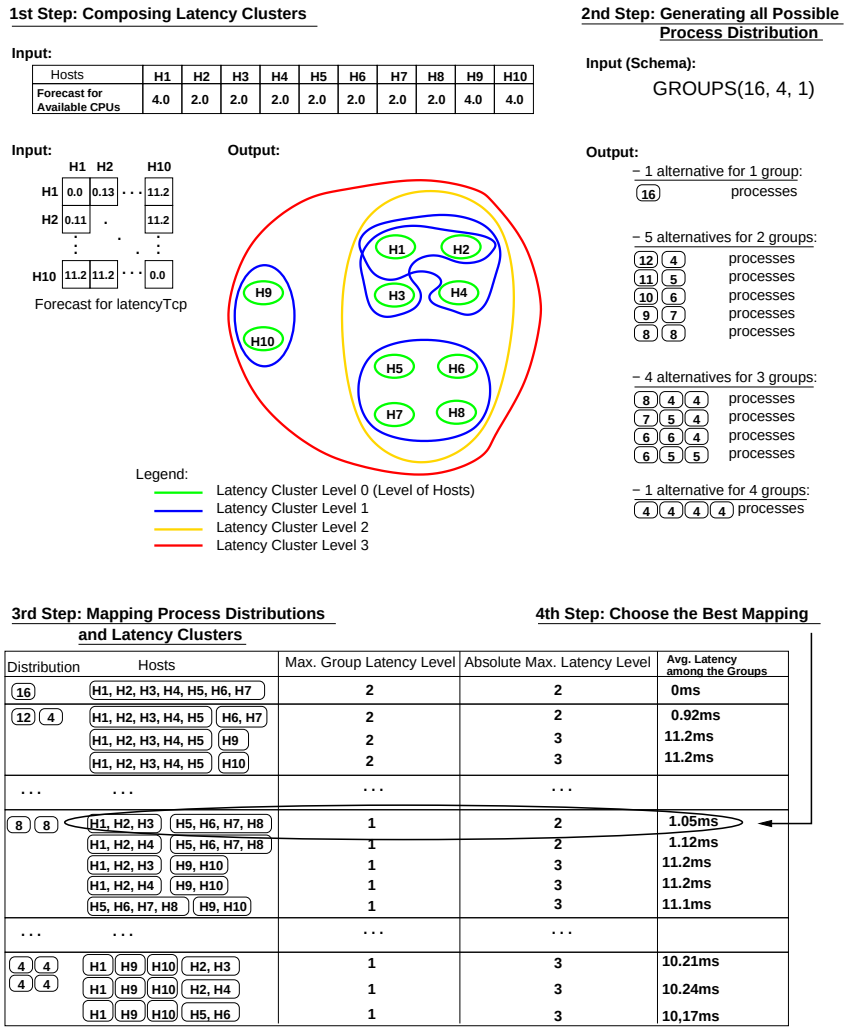

Figure 2: The Scheduling Algorithm

This mechanism expects as input the list of hosts, an up-to-date forecast for the available CPU fractions on these hosts, an up-to-date forecast for the latency values in milliseconds predicted for each pair of hosts and finally the schema which specifies the preferred communication pattern of a program. The first three are provided by the Network Weather Service while the last one is given by the user in an XML format.

Figure 2 presents how the scheduling algorithm maps a given "groups" schema (with 16 processes which can be organized some local groups whose sizes are not less than 4) to a particular grid infrastructure consisting of 10 hosts (there are 3 hosts with 4 CPUs respectively, the others have only 2).

3.1 The Scheduling Algorithm in Details

The algorithm applied by the Scheduling Mechanism works as follows (in the case of the schema type "groups"):

1. **Classification of *Latency Values*:** First, we classify the given latency values according to their order of magnitude. This classification is performed with the help of a particular factor we call the *LatencyClassFactor* (=4, empirically determined value). We compose distinct classes from the latency values started with the absolute minimum latency value such that in a class the values cannot be greater than the product of the absolute minimum latency value and of the `LatencyClassFactor` raised to the subsequent power (started with 1). 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. **Composing *Latency Clusters*:** Then we introduce the notion of *latency clusters*. A latency cluster consists of some hosts such that the latency levels of the links between any two member hosts of such a cluster cannot exceed a certain value. Such latency clusters have the following properties:

   - Each host itself is a latency cluster with level 0.
   - Latency clusters need not be distinct (two or more latency clusters may have intersection, e.g.: within a LAN a link is already used intensively between two hosts; in this case this link may have higher latency level than the others in the LAN, therefore, two latency clusters are created within this LAN, see the hosts `H3` and `H4` in the first step on Figure 2).
   - A latency cluster may contain other latency clusters with less maximum latency levels.
   - For each latency cluster, we keep track the sum of the available CPU fractions of the hosts contained by the latency cluster.

   We determine all possible latency clusters and store them in a list which is sorted according to maximum latency levels in ascending order. Furthermore the latency clusters that have the same maximum latency level are sorted according to the number of the available CPUs in descending order in the list.

The first and the second (current) steps are independent from any given schema and they can hence be performed before the user intends to schedule a program with its communication schema.

3. **Generating Process Partitions:** We determine all those process partitions (where processes are organized into various local groups) which fulfil the given communication schema and store them in a list (see the second step in Figure 2).

4. **Mapping Latency Clusters and Process Partitions:** Now we find the optimal mapping for each group in every process partition. The groups which are parts of the same process partition are mapped in a descending order according to their size (greater groups mapped earlier). Since the list of latency clusters is ordered (first ascending order according the maximum latency level then descending order according to the available CPUs) we always start the search at the beginning of this list and choose the first latency cluster for a group which fits (such that the size of the group is less than or equal to number of the available CPUs in the latency cluster.

   But if a group can be mapped to more than one latency cluster which have the same maximum latency level then we also generate all the mapping alternatives (the reason for this that we attempt to find the mapping in which the groups are located closest to each other in terms of latency).

   (**Remark:** One can ask why we do not simply map all processes to the first fitting latency cluster in the list instead of the mapping of the generated partitions of processes. The answer is that the topology of such a latency cluster may not be the optimal for the preferred communication pattern specified by the given schema. Of course we prefer to choose a mapping comprising less number of local groups, see step 5).

5. **Choosing the Best Mapping:** This step can be regarded as a part of the previous step and it must be performed every time after a new alternative mapping was generated. We always compare the newly generated mapping with the mapping which was the best so far. We always store only the current best mapping. The comparison is based on the following conditions:

   - We choose the mapping where the maximum latency level of the local groups is smaller;

- if the maximum latency levels of the local groups in the two mappings are equal then, we choose the mapping where the absolute maximum latency level is smaller;
- if the absolute maximum latency levels of the two mappings are equal, then we choose the mapping which consists of less number of local groups;
- ultimately if both mappings consists of the same number of local groups, then we choose the mapping where the average latency of the links among the local groups is less (see Figure 2).

3.2 Differences in the Case of the Schema "Graph"

In the case of the schema "graph" the algorithm is slightly different because the number of groups and their sizes are fixed by the given schema. So in this case we deal only with one possible process partition and we can therefore skip the third step of the algorithm and we did not compare in the last step which mapping alternatives contain less number of local groups.

Additionally since the schema "graph" specifies edges of a graph (which are pre-defined connections among the local groups), in the comparison of the mapping alternatives (in step 5) instead of the absolute maximum latency level and average latency value of all connections among the groups we apply the maximum latency level of and average latency value of the pre-defined connections among the groups (except if no edges are defined by the schema, because in this case we assume we have a fully connected graph which is the same implicit assumption as in the case of the schema "groups").

3.3 Differences in the Case of the "Tree" and the "Ring" Schemas

In case of the schema "tree" and the schema"ring" the algorithm slightly differs from the description presented in Section 3.1 as well. Although the number and the sizes of the local groups are not specified in advance, but each possible partition contains pre-defined connections among its local groups. Hence, we apply the maximum latency level and average latency value of the pre-defined connections among the groups in the comparison in the last step (instead of absolute maximum latency level and average latency value of all connections among the groups).

Furthermore, in the case of trees we take into account that the local managers should be scheduled together with the corresponding leaf groups (mapped to the same latency cluster).

3.4 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 [6]. This problem can be overcome with a careful, network topology dependent configuration of the Network Weather Service (by establishing a corresponding clique hierarchy).

4 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 [5]. 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.

4.1 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 [4] 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 A.

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 [4].

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 [3].

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 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 [3]. 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* [3] 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 [5], too.

```
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 XML_DEFAULT "ringWith6Groups.xml"

008: void create_message(int rank, int grp, char* s) {
009:    sprintf(s,"MESSAGE FROM P%d (from G%d)",rank, grp);
010: }

011: void process_message(int step, int rank, int grp, char* s) {
012:    printf("IN STEP %d: P%d (from G%d) RECEIVED: \"%s\".\n",
                  step, rank, grp, s);
013: }

014: int main(int argc, char *argv[]) {
015:    int nrProcs, rc, flag;
016:    int rank, localRootRank, remoteRootRank;
017:    int grpRank, leftGrpRank, rightGrpRank;
018:    MPI_Comm mpiComm1, mpiComm2;
019:    MPI_Group mpiGrp1, mpiGrp2;

020:    char buff[MSG_SIZE];

021:    rc = MPI_Init(&argc,&argv);
022:    if (rc != MPI_SUCCESS) {
023:       printf("Error starting MPI program.\n");
024:       MPI_Abort(MPI_COMM_WORLD, rc);
025:    }
026:    MPI_Comm_size(MPI_COMM_WORLD,&nrProcs);
027:    MPI_Comm_rank(MPI_COMM_WORLD,&rank);

028:    if (argc > 1) {
029:       rc = TAAG_Init(argv[1]);
030:    }
031:    else {
032:       rc = TAAG_Init(XML_DEFAULT);
033:    }

034:    if (rc != TAAG_SUCCESS) {
035:       fprintf(stderr,"Error (code %d) initializing the TAAG
                        structure on process %d.\n", rc, rank);
036:       MPI_Finalize();
```

```
037:       return rc;
038:    } //if

039:    TAAG_Ring_isRing(&flag);
040:    if (!flag) {
041:       fprintf(stderr, "Error (code %d) the given schema is
                  NOT a ring.\n", TAAG_ERR_SCHEMA);
042:       TAAG_Free();
043:       MPI_Finalize();
044:       return TAAG_ERR_SCHEMA;
045:    } //if

046:    TAAG_Group_rank(rank, &grpRank);

047:    TAAG_Ring_right(grpRank, &rightGrpRank);
048:    TAAG_Ring_left(grpRank, &leftGrpRank);

049:    /****************** First Step ************************/
050:    if (grpRank % 2 == 0) { //even group rank
051:       TAAG_Group_element(grpRank, 0, &localRootRank);
052:       TAAG_Group_MPIGroup(1, &localRootRank, 1, &rightGrpRank,
                  &mpiGrp1);
053:       if (rank == localRootRank) create_message(rank, grpRank,
                  buff);
054:    } //if
055:    else { //odd group rank
056:       TAAG_Group_element(leftGrpRank, 0 , &remoteRootRank);
057:       TAAG_Group_MPIGroup(1, &remoteRootRank, 1, &grpRank,
                  &mpiGrp1);
058:    } //else

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

061:    //sending/receiving broadcast from the root of each even
        //groups to all element of its right neighbor
062:    if (mpiComm1 != MPI_COMM_NULL) { /* not every process is
                                        involved in the broadcast */
063:       MPI_Bcast(buff, MSG_SIZE, MPI_CHAR, 0, mpiComm1);
064:       process_message(1, rank, grpRank, buff);
065:    }
```

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

070:       if (rank == localRootRank) create_message(rank, grpRank,
              buff);
071:    } //if
072:    else { //even group rank
073:       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] 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.

[4] 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.

[5] 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.

[6] 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.