A Supercomputing API for the Grid*

Károly Bósa† and Wolfgang Schreiner†

Abstract. *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.*

1 Introduction

No parallel supercomputing 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 a newly developed distributed programming software framework and API for grid computing. The goal of this grid-based programming solution is to empower applications to perform scheduling decisions on their own and utilize the information about the grid environment in order to adapt their algorithmic structures to the particular situation.

Our system is not another grid workflow tool (it is not designed for applications that are just collections of independent sequential components). On the contrary, it is a real parallel programming framework which hides low-level execution details from the applications by employing abstract execution models for heterogeneous grid environments. Moreover, 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 multilevel 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 envi-

*The work described in this paper is supported by the Austrian Grid Project, funded by the Austrian BMBWK (Federal Ministry for Education, Science and Culture) under contract GZ BMWF-10.220/0002-II/10/2007.

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

ronment. 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.

The rest of the paper is organized as follows. First we give in Section 2 a short survey about the state of the art. Then we outline in Section 3 our idea about predefined communication schemas for grid-distributed applications which served as basis for our work. In Section 4 we give an overview on the overall software architecture of our grid programming framework. In Section 5 we present our topology-aware API. We discuss in Section 6 the applied scheduling algorithm in detail. Finally, we present in Section 7 a performance comparison with benchmark results between MPICH-G2 and our system and conclude in Section 8.

2 State of the Art

A grid environment is inherently parallel, distributed, heterogeneous and dynamic, both in terms of involved resources and their performance [9]. While it may be possible to build grid applications using existing programming tools [8], they are not particularly well-suited for developing and managing flexible compositions or deal with heterogeneous hierarchies of machines, data and network with heterogeneous performance. [13] investigates what properties and capabilities grid programming tools should possess to support not only efficient grid codes, but their effective development. Currently there is not any tool that addresses all requirements in all situations.

Due to the OGF standardization efforts to compose a simple, stable, and uniform high-level programming interface for the Grid, the *Simple API for Grid Applications (SAGA)* [10] was designed and implemented in the last couple of years. SAGA integrates the most common grid programming abstractions, but it yields only limited support for inter-process communication and it is not aimed to cover any requirements of large-scale, high-performance computing on the grid.

Obtaining high-performance on the grid requires a balance of computation and communication among all involved resources. Currently this can be done by manually managing computations, communications and data locality using message-passing (e.g.: MPI) or remote procedure call (e.g.: GridRPC). Although GridRPC [15] may become an OGF standard as a parallel programming interface for the grid (and it is supported by SAGA), it is restricted to the client-server model (as any other remote procedure call API) and lacks the versatility and power of message-passing based APIs.

While MPI addresses some of the challenges in high-performance grid computing, it was originally designed only for clusters or other homogeneous network environments [14]. A parallel programming environment evolved for the grid must be *topology-aware* in that sense that it must be aware of and exploit the characteristic of an available physical network architecture. Typical topology-aware programming tools are e.g. *MPICH-G2* [12] and *MPICH-VMI* [17]. Both of them are grid-enabled MPI implementations based on the MPICH library. MPICH-G2 uses some grid services provided by the Globus Toolkit pre-Web Service architecture. MPICH-VMI utilizes the middleware communication layer *Virtual Machine Interface (VMI)*. The most important difference between MPICH-G2 and MPICH-VMI is that:

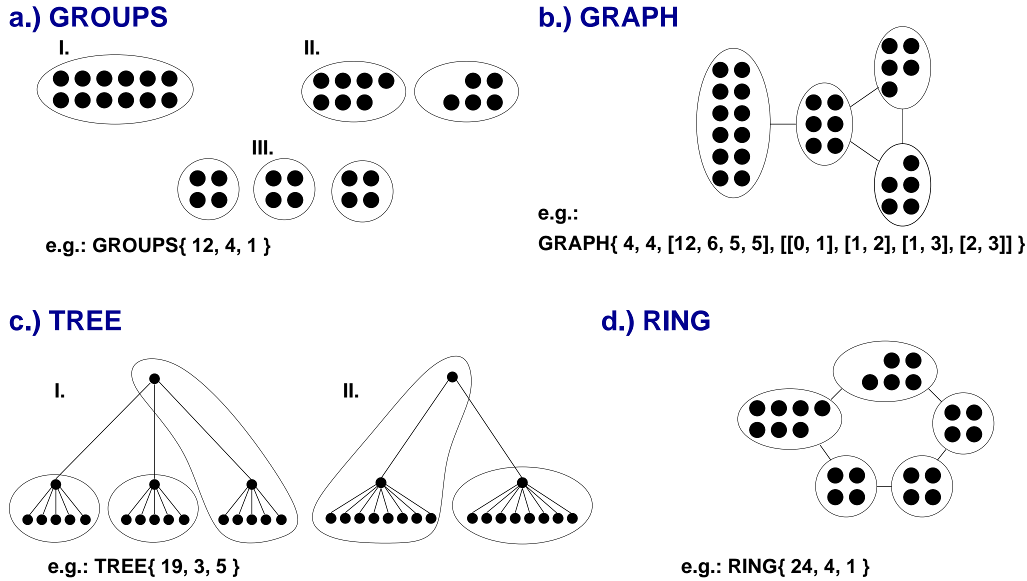


Figure 1. Applicable Communication Schemas

- while the earlier requires the user to manually provide the physical topology of the network (upto 4 levels) using the *Resource Specification Language (RSL)* (if the user does not provide this information, the processes of parallel programs are simply assigned to grid machines in sequence, according to a given *"machines"* file),
- the latter constructs a 2-levels network topology at runtime using the *Grid Cluster Resource Manager (GCRM)* which is an external service on the *TeraGrid*.

In 2007, a successor of MPICH-G2 was released called *MPIg* [3] which has already been able to start parallel programs via the Web-Service architecture of the Globus Toolkit and which provides some performance enhancements compared to MPICH-G2.

Summarizing the achievements of these MPI-based systems, we can say that existing topology-aware programming tools 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.

3 Predefined Communication Schemas

In our approach, we conform the communication structure of parallel programs to particular grid resources by applying some predefined communication schemas which classify the point-to-point connections between the processes as "often used" and "rarely used" communication links. By this, the schema assigned to a parallel program specifies the intended communication patterns of the program in heterogeneous network environments. The schemas are formulated in a simple XML-based syntax.

In general the schemas arrange processes into *local groups*. The processes of a local group are supposed to interact each other frequently; they are therefore intended to execute in a local network

environment (cluster or LAN). The following kinds of communication schemas are currently employed [5] (see Figure 1):

Groups This schema defines a prescription how processes of a parallel program may be partitioned into local groups. In the schema we can specify the number of processes, a minimum size for local groups and a divisor for the number of the local groups. We assume that each group can interact with any other group with equal chance (so the supposed communication pattern among the local groups is a fully connected graph).

In the example depicted in Figure 1a, the schema requires to schedule 12 concurrent processes into local groups such that each group consists of 4 processes at least. The prescribed divisor is 1, thus there is no restriction for the number of the groups. There are many possible distributions which fulfill these conditions (some candidates are presented on Figure 1a). The execution engine of our system attempts to find a corresponding process distribution which can be scheduled to the available physical grid resources such that the assessed execution time of the program is minimal.

Graph This schema is similar to "Groups". However here one can give the accurate size of each local group and can define additionally edges/links between the groups such that a communication pattern among the groups is described (see Figure 1b). The purpose of these edges is to schedule the connected groups close to each other (in terms of latency of the physical grid architecture). Of course, this notation does not mean that only those groups can interact with each other that are bound to each other by such a predefined edge (it is just used to indicate the preferences for scheduling). If no edges are defined by a given schema "Graph", then the schema is treated as a schema "Groups" with fixed group sizes (so in this case it is supposed that the communication pattern among the local groups is a fully connected graph).

Tree This schema specifies a tree-like multilevel parallelism with the given number of processes and the given number of tree levels. The sizes of the local groups located on the lowest level (the level of leaves) are not fixed but just a minimum number of processes comprised in such groups is prescribed (the non-leaf processes represent one-element local groups).

If we regard the example depicted on Figure 1c, where a schema "Tree" is given with 19 processes and with 3 levels such that the minimum size of the leaf groups is 5. Again we can find several tree structures which fulfill the given requirements. The execution engine applies a heuristic to select and schedule a tree structure as optimal as possible (first it is allowed to map the structure to one local network environment, if this is not possible, then it is split and assigned to two or three, etc). In addition, the engine attempts to place the parent processes of the leaves (local managers) close to their children processes (in terms of latency).

Ring This schema is similar to schema "Groups" (its argument list is the same, too), but the local groups always compose a ring (see Figure 1d). In the case of this schema, the execution engine takes care of the placing of the groups, such that neighbors in the ring are scheduled on the physical grid architecture close to each other (in terms of latency).

There is an additional schema called *singleton* which is used for scheduling such applications which were designed only for homogeneous network environments (like pure MPI programs). In this schema only the number of the processes is specified and these processes are assigned to the same local network environment.

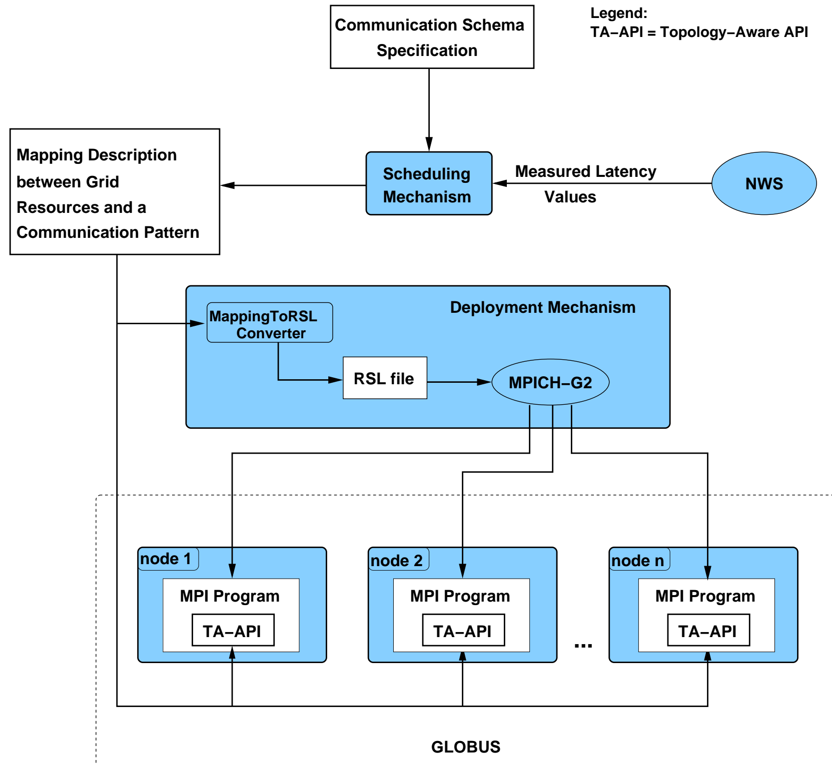


Figure 2. Overview on the Software Framework

4 Software Architecture

We have implemented a first prototype version of our software framework [5, 6] called *"Topology-Aware API for the Grid"* (*TAAG*). The implementation is based on the pre-Web Service architecture of the Globus Toolkit [1] and on MPICH-G2 [12]. Our system consists of three major components (see Figure 2):

Scheduling Mechanism This component depends on the *Network Weather Service (NWS)* [18], 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 (see Section 6). 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:

Initialization and Termination:

```
int TAAG_Init(char *file)
int TAAG_Initialized(int *flag)
int TAAG_Free()
```

Calls Related to Groups/Graphs:

```
int TAAG_Group_number(int *nr)
int TAAG_Group_rank(int rank, int *grpRank)
int TAAG_Group_size(int grpRank, int *nr)
int TAAG_Group_members(int grpRank, int nr, int *members)
int TAAG_Group_element(int grpRank, int index, int *rank)
int TAAG_Group_MPIGroup(int nrProcs, int *ranks,
        int nrGroups, int *grpRanks, MPI_Group *grp)
int TAAG_Group_degree(int grpRank, int order, int *nr)
int TAAG_Group_neighbours(int grpRank, int order,
        int nr, int *grps)
int TAAG_Group_distance(int grpRank1, int grpRank2, int *nr)
int TAAG_Group_way(int grpRank1, int grpRank2,
        int nr, int *grps)
```

Calls Related to Trees:

```
int TAAG_Tree_isTree(int *flag)
int TAAG_Tree_isLeaf(int rank, int *flag)
int TAAG_Tree_root(int *rank)
int TAAG_Tree_depth(int *depth)
int TAAG_Tree_level(int rank, int *level)
int TAAG_Tree_parent(int rank, int *parent)
int TAAG_Tree_width(int rank, int level, int *nr)
int TAAG_Tree_children(int rank, int level,
        int nr, int *ranks)
```

Calls Related to Grid Resources:

```
int TAAG_Host_number(int *nr)
int TAAG_Host_address (int rank, char *address)
int TAAG_Host_properties(int rank, int *nrCPUs,
        int *nrProcs)
int TAAG_Host_processes(int rank, int nr, int *ranks)
int TAAG_Host_latency(int rank1, int rank2,
        double *latency)
```

Figure 3. TAAG API Calls with respect to Initialization/Termination, Grid Resources as well as "Groups", "Graph" and "Tree" Schemas

- 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 MPICH [2] programming library. 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 more details, see Section 5

The system has already been tested successfully on the sites `altix1.uibk.ac.at` (Altix 350) and `alex.jku.austriangrid.at` (Altix ICE 8200) of the Austrian Grid.

5 The Topology-Aware API

The API is an addition to the MPICH [2] programming library. Its purpose is to inform a parallel program

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

All these tasks are performed according to the XML-based mapping file (which is generated by the scheduling mechanism). For all programs which intend to use any calls of our API, the header file *taag.h* must be included in its source. A part of the API is presented on the Figure 3; the function calls can be classified as follows:

Initialization/Termination calls initialize, check or deallocate the corresponding data structures which describe the runtime environment and the algorithmic structures composed by the processes (based on the mapping file provided by the scheduling mechanism), see Figure 3.

Groups/Graph-related calls deal with the local groups and their relations, see Figure 3 (since the "Tree" and "Ring" schemas describe special graphs composed by local groups the calls in this class can be applied in case of all kinds of communication schema). The simpler "Groups/ Graph"-related calls return values like the number of groups, the rank of a group to which a given process belongs, the number of processes in a group and the ranks of all or some processes contained by a group.

The call `TAAG_Group_MPIGroup` composes a single MPI_Group structure from some given individual processes and/or from the processes of some given local groups (by which new MPI communicators can be created easily for carrying out collective operations among the specified processes). Further calls are used if some edges are specified among the local groups; they determine the number of neighbor groups, the ranks of the neighbor groups (first order neighbor to n-th order neighbors), the distance of two groups and the way how to get from one group to another along the specified edges (if possible).

Tree-related calls deal with communication patterns based on the schema "Tree", see Figure 3. Some of them decide about whether the given mapping file describes really a tree structure or whether the given process rank belongs to a leaf process; other calls give back the rank of the root process, the depth of the entire tree, the level on which a given process is located, the rank of the parent process of a given process and finally the number of processes occur on a certain level of a given subtree and the ranks of these (child) processes.

Ring-related calls deal with communication patterns based on the schema "Ring". This class of API calls contains only three different calls. One of them checks whether the given mapping file describes really a ring structure. The other two return the ranks of the left or of the right neighbor groups, respectively.

Grid Resources-related calls query the momentarily available grid resources such as the number and the names of the allocated hosts as well as the number of CPUs, the number and ranks of processes located on the host where the given process resides. Last but not least the call `TAAG_Host_latency` returns the network latency value between two given processes (if they are executed on the same host the return value is 0).

By this API, programmers need to specify mainly the roles of the processes in a given hierarchy, but they do not need to know in advance the particular characteristic of the available grid architecture. For instance, in the skeleton of the tree-like multilevel parallelism in Figure 4, essentially only the process roles such as root, leaves and others are described after the initialization of the runtime environment.

We demonstrate the versatility of the TAAG API, by a simple distributed example application [6] which can be used to establish different kinds of tree-like multilevel parallelism on the grid according to a mapping file. A skeleton of this example program is presented in Figure 4. First, the program initializes the runtime environment and the algorithmic structures composed by the processes according to the mapping file (whose name and location are always specified as the first argument of the program by the deployment mechanism of the TAAG system). Then the root of the tree creates some computational tasks and distributes them among its child processes, which in turn distribute further

```
#include <mpi.h>
#include <taag.h>
...
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&nrProcs);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
TAAG_Init(argv[1]);

TAAG_Tree_root(&root);
if (rank == root) {     /***** root branch *****/
      TAAG_Tree_width(rank, 1, &nrChildren);
      TAAG_Tree_children(rank, 1, nrChildren, children);
      ...
} //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);
          ...
    }//if
    else {    /**** leaf branch ****/      ...    } //else
} //else
TAAG_FREE();
MPI_Finalize();
...
```

Figure 4. A Skeleton of the source of a Tree-Like Multilevel Parallel Application

these tasks among their children till the tasks reach the leaf processes. The leaves process the tasks and return the outcome to the root via their parents.

The 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 by applying different "Tree" schemas (with various number of processes arranged into arbitrary tree levels, etc.). The completed version of this program code has been tested successfully together with various tree structures on several sites of the Austrian Grid.

6 The Scheduling Mechanism

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) [7]. Nowadays the technique of communication-aware mappings is recalled in connection with heterogeneous multi-cluster and grid environments [16].

6.1 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 [18] 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

a.) Composing Latency Clusters

Input:

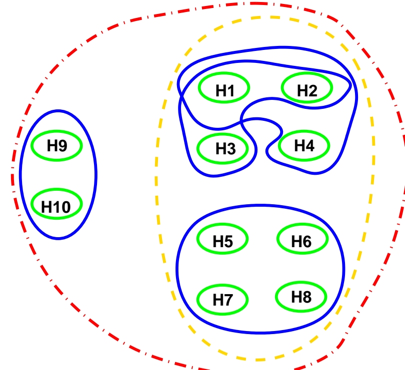
Hosts	H1	H2	H3	H4	H5	H6	H7	H8	H9	H10
Forecast for Available CPUs	4.0	2.0	2.0	2.0	2.0	2.0	2.0	2.0	4.0	4.0

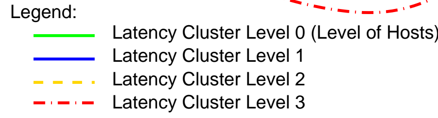
Input:

	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

Output:





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 5. Composing Latency Clusters and Process Partitions

cluster cannot exceed a certain value (some of these latency clusters may comprise some others with less maximum latency level), see Figure 5a. 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 5b.

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. Only those partitions are kept for the mapping which either contains only one group or has 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).

   - 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.

6.2 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 latency value of the pre-defined connections instead of the average latency value of all connections among the groups.

6.3 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 latency value of the pre-defined connections instead of the average latency value 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).

6.4 Disadvantage of the Algorithm

The algorithm assumes that on each host of a grid architecture an NWS sensor runs and the latencies between all pairs of hosts are measured. 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 performance 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 [18]. This problem may be overcome with a careful, network topology dependent configuration of the Network Weather Service (by establishing a corresponding clique hierarchy).

7 Comparative Benchmarks with MPICH-G2

To prove the efficiency of the concept of our communication schema based programming and scheduling solution, we performed some comparative benchmarks with TAAG and (pure) MPICH-G2. For this, we had to find a problem that requires a structured communication pattern which can be adapted to heterogeneous network environments, but which can still be implemented easily in pure MPI, too. We chose the well-known n-body problem as a basis of our tests. Here a large number of particles

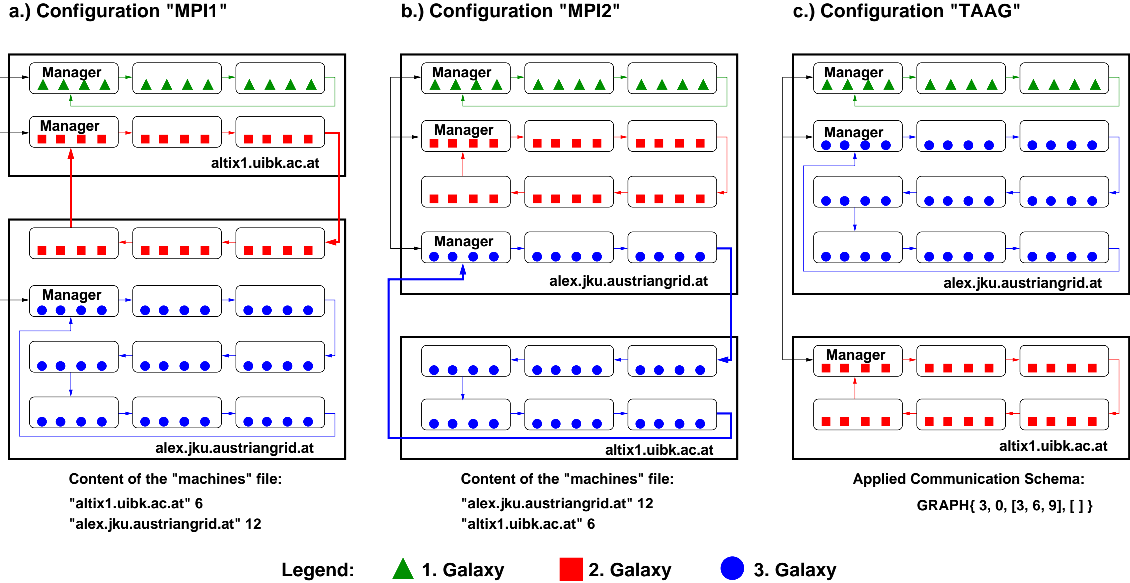


Figure 6. The Applied Test Configurations for the Pure MPI Program and for the TAAG-Based Version.

is given (with their positions and masses) as input; the task is to compute the future positions of the particles by taking into account the mutual gravitational attraction among them.

In more detail, we choose as the core of our demonstration an open-source n-body simulation [11]. On the basis of this application we implemented two variants of a toy solution, one in pure MPI and one in TAAG. They solve a special case of n-body problem, where the given particles are arranged into some subsystems called galaxies. The computation of the new positions of all particles of a galaxy is assigned to a disjunct groups of processes. For simplicity, we assume that the galaxies are located far enough from each other such that a collision is not possible between any two of them during the investigated time intervals; the initial setup is prepared accordingly. Due to this assumption a galaxy can be regarded from the other galaxies as a single heavy particle (as long as two galaxies do not approach each other or collide).

So for computing the next position of a particle, on the one hand the actual positions of all other particles of the comprising galaxy and on the other hand the total masses of and the coordinates of the central mass points of the other galaxies are required. The former data is shared among the corresponding processes by applying a ring pipeline communication pattern on the lower level in which $(n-1)*n$ messages are sent in every turn (where n is the number of those processes maintaining the particles of a galaxy). The latter data have to be calculated and exchanged among the appointed manager processes of process groups on the higher level (a more general version of the program which will be prepared to take into consideration the collision between two or more galaxies is under development). We have to notice that the implementation of even such a simple two levels algorithm was already much more complicated in pure MPI than with the usage of the API of TAAG.

In the test cases, we used a setup in which the particles are organized into three galaxies such that the sizes of the galaxies are in proportion 1:2:3. Furthermore the particles were always distributed evenly among the processes, so each process maintains the positions of the same amount of particles.

The test cases were executed on two Austrian Grid sites `alex.jku.austriangrid.at` residing in Linz and `altix1.uibk.ac.at` residing in Innsbruck. The former consists of Intel Xeon

Nr. Of Procs.	Particles /Procs.	Time Steps	Configurations			Speedups	
			Mpi1	Mpi2	Taag	Mpi1/Taag	Mpi2/Taag
18	500	10	1.5806s	2.2361s	1.3031s	1.212x	1.715x
18	1000	10	6.0735s	7.7509s	5.1444s	1.18x	1.506x
18	2000	10	23.8034s	29.3047s	20.3422s	1.17x	1.44x
18	4000	10	107.3881s	136.3655s	87.1078s	1.232x	1.565x
18	1000	60	40.4837s	48.6121s	32.7855s	1.234x	1.482x
18	1000	240	161.1072s	203.6982s	133.141s	1.21x	1.529x

Figure 7. Execution Times and Speedups

processors (2.5GHz) the latter comprises Intel Itanium processors (1.4GHz). The two machines are connected via a gigabit WAN connection provided by the Austrian Academic Computer Network (ACONet). In our tests, we used 18 processes and 18 CPUs (12 CPUs on `alex` and 6 CPUs on `altix1`) such that every process can be assigned to a separate CPU (so 3 CPUs are assigned to the smallest galaxy, 6 CPUs to the middle one and 9 CPUs to the largest galaxy).

In MPICH-G2, "machines" files are used to list the computers on which we wish to run our programs. Next to an enumerated machine name in each line, a number appears which specifies the maximum number of processes that can be executed on the machine. The processes are always scheduled to the machine listed first. Then if we intend to use more processes than can be established on the first machine, the remaining processes are scheduled to the subsequent machine in the list. Consequently, the only way to influence the scheduling of the processes from a "machines" is by changing the order of the machine names (it is supposed that the maximum number of executable processes are fixed for each machine).

For scheduling the processes of the MPI program, we applied two different configurations given in "machines" files (see Figure 6).:

- According to the first configuration labeled as *"MPI1"* where the machine `altix1` is listed first, the processes that maintain the particles of the second galaxy are distributed between `altix1` and `alex` (see Figure 6a).
- According to the second configuration labeled as *"MPI2"* where the machine `alex` is listed first, the processes that maintain the particles of the third galaxy are distributed between `alex` and `altix1` (see Figure 6b).

Hence, in both MPI-related configurations from all the $n*(n-1)$ messages sent on the corresponding ring pipeline, $2*(n-1)$ messages have to be sent via the WAN network connection in every time steps (where n is the number of processes maintaining the particles of a galaxy).

For scheduling the processes of the TAAG-based version of the program we applied the following communication schema

$$GRAPH\{3,0,[3,6,9],[]\}$$

which yields the optimal scheduling such that the first and the third galaxies are maintained on the `alex` and the second galaxies are maintained on the `altix1` (see Figure 6c). Apart from the way

how the group of processes are established and assigned to galaxies, the both versions (the pure MPI and the TAAG-based versions) of the program contain the same piece of code (we avoided the usage of any convenient statements or programming structure introduced by our API).

The execution times presented on Figure 7 are average values of 5 computations and do not include the overhead of the job submission in Globus. As it can be seen that the reachable speedup factor is independent how we decreased or increased either the number of the particles or the time interval of the simulations in the test cases. The use of our topology-aware API always speeded up the simulation by a factor of 1.2 compared to the "MPI1" configuration and by a factor of 1.5 compared to the "MPI2" configuration roughly. We have also investigated the case in which the optimal scheduling is possible by MPICH-G2, too (e.g.: on both grid sites 9-9 CPUs are available). In these test cases, the average execution times of the two versions of the application were quite alike. Consequently, we can state that the usage of our TAAG framework offers at least as efficient program execution on the grid as MPICH-G2.

In all likelihood in a more realistic test circumstances (in a testbed containing more than two clusters) the achieved speedups would be higher because the chances for remote communication in a pure MPI solution are much higher. In the future, we intend to develop an efficient distributed n-body simulation (probably on the basis of the hierarchical Barnes-Hut algorithm [4]) whose execution beyond a certain problem size shall require a more complex decomposition of process groups among physical hardware resources. Since in these proposed simulations many to many grid sites interact with each other according to a compound logical hierarchy (instead of the presented simple algorithmic structure consisting of only two machines), we expect a more significant performance gain on the side of the TAAG framework (in comparison with pure MPICH-G2).

8 Conclusions

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 [3]. 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.

References

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

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

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

[4] J.E. Barnes and P. Hut. A Hierarchical O(N Log N) Force Calculation Algorithm. *Nature*, 324(4):446–449, December 1986.

[5] 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.

[6] 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.

[7] 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.

[8] I. Foster and C. Kesselmann. *Computational Grids*, chapter 2, pages 15–51. In [9], 1999.

[9] I. Foster and C. Kesselmann, editors. *Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufman, 1999.

[10] T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. von Laszewski, C. Lee, A. Merzky, H. Rajic, and J. Shalf. SAGA: A Simple API for Grid Applications. High-Level Application Programming on the Grid. *Computational Methods in Science and Technology*, 12(1):7–20, 2006.

[11] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[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] C. Lee, S. Matsuoka, D. Talia, A. Sussman, N. Karonis, G. Allen, and J. Saltz. A Grid Programming Primer. Global Grid Forum, Advanced Programming Models Working Group, August 2001.

[14] Motohiko Matsuda, Tomohiro Kudoh, and Yutaka Ishikawa. Evaluation of MPI Implementations on Grid-connected Clusters Using an Emulated WAN Environment. In *3rd International Symposium on Cluster Computing and the Grid (CCGRID)*, page 10, 2003.

[15] H. Nakada, S. Matsuoka, K. Seymour, J. Dongarra, C. Lee, and Casanova. A GridRPC Model and API for End-User Applications. GridRPC Working Group of Global Grid Forum, June 2007.

[16] 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.

[17] A. Pant and H. Jafri. Communicating Efficiently on Cluster Based Grids with MPICH-VMI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 23–33, Washington, DC, USA, 2004. IEEE Computer Society.

[18] 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.