Austrian Grid

Initial Design of a Distributed Supercomputing API for the Grid

Document Identifier:	**AG-D4-2-2008_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	26.09.2008	Initial Version	K. Bosa, W. Schreiner

Initial Design 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 26, 2008

Initial Design of a Distributed Supercomputing API for the Grid

Károly Bósa Wolfgang Schreiner

September 26, 2008

Abstract

We proposed to participate in the Austrian Grid Phase 2 within the frame of the activity "Grid Research". We intend to 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 outline our idea concerning the proposed software system and already discuss some implementation details.

1 Introduction

We proposed to participate in the Austrian Grid Phase 2 [1] within the frame of the activity "Grid Research". We intend to deal with development of a distributed programming software framework and API for grid computing. This work shall in particular assist 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 shall empower 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 in 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 and connections that dramatically vary between (at least) three different levels: the processors within a grid node, the grid nodes within the same network, and grid nodes in different networks linked by wide-area connections. Correspondingly, the API shall not hide this fact from the application but reflect the information provided by the 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 shall however hide 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 has to map these abstract model features to the appropriate underlying mechanisms: to initiate an activity on a local machine or on a machine within the same administrative authority, simply a process may be started, to initiate an activity on a remote node may mean to contact a corresponding service on that machine, provide the appropriate credentials, and ask the service to start the activity.

In this document, after we give a short survey about the relevant part of the "State of the Art" in Section 2, we outline our idea in Section 3 and discuss some implementation issues concerning our proposed software system in detail in the further sections. In Section 8, we describe the proposed API in details and give some examples which present the usage of this API. Finally, we outline an implementation plan for the major components of the proposed software system in Section 9.

2 State of the Art

This section is only a short overview of [5]. To execute efficiently nowadays advanced grid applications, the fundamental grid middleware services (e.g.: security, resource allocation, etc.) are not sufficient anymore, but some additional grid service layers are required to introduce like *performance prediction* (see Figure 1). The term performance prediction in the context of grid denotes a group of grid services which provide assessment for the performance of various grid resources in advance for a limited period of time.

An example for the applying of performance prediction is the ASKALON

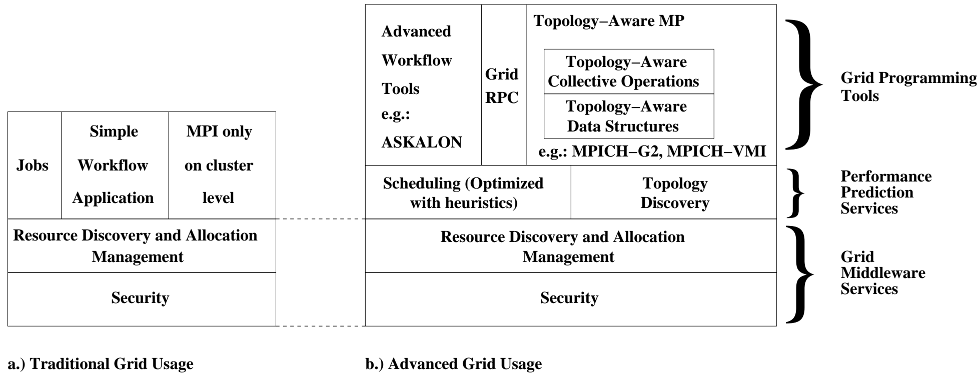

Figure 1: Traditional and Advanced Programming of the Grid

project [7, 15, 14] that develops integrated environments under the WSRF framework of Globus Toolkit 4 to support the development and execution cycle of scientific workflows on dynamic grid environments. ASKALON employs its own prediction model based on historical data collected through a well-defined experimental design and training phase. The output of this prediction service is used in scheduling mechanisms enhanced with global optimization heuristics to find good mappings onto the grid that minimize the execution time.

A typical performance prediction service is the *Network Weather Service (NWS)* [16] software system, which became a de facto standard in the grid community as it is used by major Grid middlewares like Globus, to gather qualitative information about the current state of a platform (both network and CPUs) and to predict its short-term performance.

The *NetSolve* [4] software system which is an implementation of the *Grid Remote Procedure Call (GridRPC)* [11] API of the OGF applies NWS for performance prediction, too.

2.1 Performance Prediction and Topology Discovery

The efficient use of the Grid resources can only be achieved from a parallel program (e.g.: a MPI program) through the use of accurate network information. Information such as the network topology is crucial to achieve tasks such as running network-aware applications [10], efficiently placing servers [6], or predicting and optimizing collective communications performance [9].

However, the description of the structure and characteristics of the network interconnecting the different Grid resources is usually not available to users. This is mainly due to security (fear of Deny Of Service attacks) and privacy reasons (hiding bottlenecks). Hence there is a need for tools which

employ common end-to-end measurements, like bandwidth and latency (but also interference measurements in an ideal case,i.e., whether an interaction between a pair of machines has non-negligible impact on the interaction of another pair of machines, namely they may use the same physical link) and automatically construct models of network platforms.

The consequence is whereas *topology discovery* is not part of the performance prediction conventionally, but network topology discovery tools for the grid must rely on only *application-level measurements* (i.e., measurements that can be done by any application running on a computing grid without any specific privilege) that are typically provided by performance prediction services as NWS. (NWS is able to report end-to-end bandwidth, latency and connection time, which are typical application-level measurements.)

2.2 Topology Aware Programming Tools

It is not enough to discover the characteristic of an available physical grid architecture, but a programming environment must be aware of and exploit this information. Such typical *topology-aware* programming environments are:

MPICH-G2 [2, 8] is a grid-enabled implementation of the MPI-1 standard which is based on the MPICH [3] library and which uses grid services provided by the Globus Toolkit pre-Web Service (pre-Web Service) architecture for user authentication, resource allocation, I/O management, process control and monitoring. MPICH-G2 implements topology-aware collective operations that minimize the communication via the slowest channels.

MPICH-G2 describes a topology with a four levels array where each level represents a communication channel: TCP over WAN (level 0), TCP over LAN (level 1), TCP over machine networks (clusters, level 2) and vendor MPI library over high performance network (level 3). MPICH-G2 assigns to every process at each level a non-negative integer named color; processes with same colors can communicate over the corresponding channel.

MPICH-VMI [13] is a grid-enabled MPI implementation which is also based on MPICH [3] and utilizes the *Virtual Machine Interface (VMI)* [12], which is a middleware communication layer that addresses the issues of availability, usability, and management within heterogeneous wide-area grids. The most important differences between MPICH-VMI and MPICH-G2 are that

- In MPICH-G2 the user must provide manually the physical topology of the network using either a Resource Specification Language (RSL) file directly (up to 4 level topology structure can be specified) or a *machine file* containing a description how CPUs are arranged on the available machines (it can result only 2 levels topology), while
- MPICH-VMI constructs a limited (2 levels) network topology at runtime using the Grid Cluster Resource Manager (GCRM) which is an external service on the TeraGrid.

Summarizing this, we can say that existing topology-aware programming tools provides the following functionalities:

- they either attempt to discover a limited (2 levels) network topology or expect a description of a (max. 4 levels) topology as an input,
- they forward and make available the given topology information on the level of their programming API and
- they optimize 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 (only in the case of collective operations).

3 The Idea

In section 2, we could see that existing topology-aware software tools, like MPICH-G2 and MPICH-VMI, can adapt the execution of the collective operations to the topology of a particular grid infrastructure. But they are still not able to adapt the point-to-point communication structure of a parallel programs to network topologies such that they achieve a nearly optimal execution time on the grid.

In our approach, we assign to a given parallel program a pre-defined schema describing a generalized communication structure designed for heterogeneous network environments. Then, this schema can be specialized further with some parameters according to some characteristics of the program. The outcome of this procedure is a *specification* of the preferred algorithmic structure of the program in heterogenous networks, see Section 4. Then we map this specification to a predicted performance model of an available physical network architecture in order to decrease the communication overhead during the execution as much as possible.

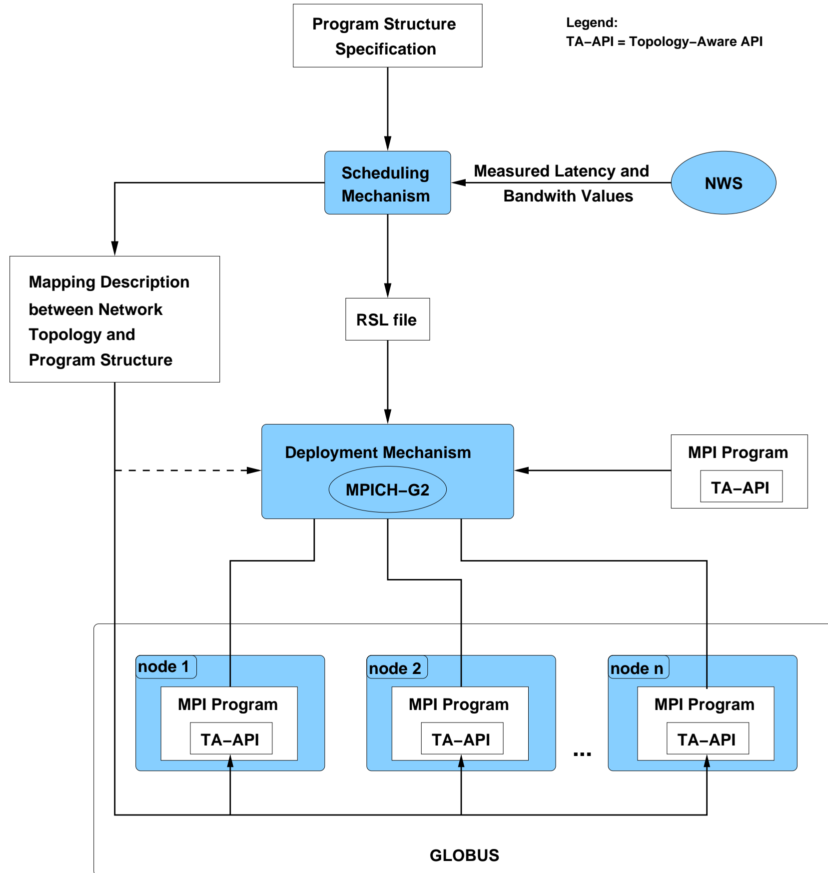

Figure 2: Overview of the Proposed Software System

By this, our software framework will be able to hide the algorithmic challenges of the topology dependency of a gridified MPI program from the programmer.

3.1 Overview on the Proposed Software Framework

Figure 2 depicts on overview about the proposed software system. In the design, we intend to compose a modular structure, where each component will interact some others via pre-defined interfaces (components denoted with ellipses on the figure are third party softwares which we may substitute in a later phase). Our planned software architecture consists of three major components:

Scheduling Mechanism depends on NWS, which provides from time to time the information about the available CPUs and about actual la-

tency (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.

Before each execution of a MPI parallel program on the grid (either on the same physical grid architecture or a different one), the scheduling mechanism maps the specified algorithmic 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* described in two files, an Globus RSL script for the Deployment Mechanism and a XML-based *mapping description* (between the network topology and a determined program structure) for the topology-aware API, see Section 5.

Deployment Mechanism is based on the MPICH-G2 starting mechanism (gridified `mpirun` for Globus). It takes the generated RSL file as input and starts the processes of the program on the corresponding grid nodes according to the content of the RSL file.

Topology-Aware API The main purpose of this API is to assign the processes of a program to the allocated grid resources according to the mapping description generated by Scheduling Mechanism (each process must identify itself with a functional role assigned to its local grid node by the given execution plan) The mapping description either will be "staged on" to the grid nodes when the MPI program is started or each process can download it from a grid catalog service (further investigation is needed to decide).

For instance, we can regard a hierarchical Manager/Worker algorithmic solution organized into a 3-levels tree of processes containing approximately 20 leaf-processes. The root process acts as the global manager, the processes on the second level are the local managers and the leaf-processes are dedicated to worker processes. The scheduling mechanism with the help of the NWS is able to determine an adequate distribution of the processes on some grid nodes, such that the point-to-point communications between the processes on the second level and the corresponding workers on the third level will be as efficient as possible (preferring clusters and LANs). Then the deployment mechanism will be able to allocate the chosen group of processes. Furthermore, the topology-aware API used in the application is able

to apply in runtime a corresponding mapping between the predefined roles in the specified hierarchy (global manager, local manager and worker) and the allocated pool of grid nodes, such that it minimize the execution time.

This approach should definitely be more efficient than existing topology-aware solutions (e.g.:MPICH-G2 and MPICH-VMI) which ignore the point-to-point structures of parallel programs.

3.2 Advantages and Disadvantages

The major advantages of the proposed solution are the following:

- It takes into consideration the point-to-point structure 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 software frameworks. This means the topology-aware collective operations of MPICH-G2 are still available, since MPICH-G2 is applied in the deployment mechanism for executing programs on the grid.

- Our system eliminates the algorithmic challenges of the topology-aware programming. The programmers should deal only with the problems which they are going to solve with the program (like in a homogeneous cluster environments).

- The distribution of the processes is always conformed to the actual loading of the network resources.

Possible disadvantages of the proposed solution can be the following:

Execution of old MPI Programs At present, most of the existing MPI programs which are intended to execute on the grid were designed originally for homogeneous environments. Hence, it is easy to find a simple structural specification for them and to run them with our solution, see Section 4. But in the case of existing MPI programs already assuming heterogeneous infrastructure (e.g.: MPICH-G2 programs) it might be challenging to find a proper algorithmic structure specification for a grid user.

Artificial Topology Our scheduling mechanism builds an assumed topological structure from latency (and bandwidth) values, which are influenced by network load. Therefore, it can be a considerable deviation between the composed structure and physical network topology.

Elongated Startup Period Since the mapping between the specification of the algorithmic structure of a program and the composed network topology is recommended to perform before each execution, startup period of an MPI program on the grid will be elongated.

4 Specifications for Heterogeneous Communication Structures

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

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

4.1 Single Group

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

$$SINGLETON\{nr, strictRestriction\}$$

with the arguments, we can specify the number of processes used by the program and a condition whether all processes must be schedule to the same local network environment. If it is not possible to find a cluster or LAN with the given number of available CPUs and the second parameter is true, then the scheduling will be unsuccessful. But if the second parameter is false, the

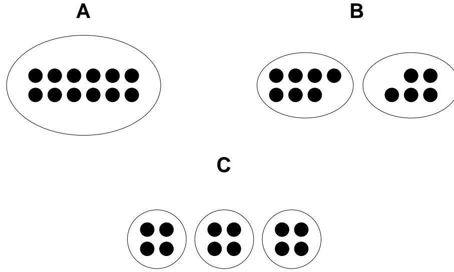

Figure 3: Some Distributions satisfying Specification $GROUPS\{12, 4\}$

scheduling mechanism always returns a possible distribution of the processes on some grid resources (which may belong to different local networks).

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

4.2 Set of Groups

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

$$GROUPS\{nr, minSizeOfGroups\}$$

The first argument is the number of processes and second is the minimum number of processes in a local group.

Example Let us regard the following specification,

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

which requires to schedule 12 concurrent processes into some local groups mapped to a heterogeneous grid environment, where each local group consists of 4 processes at least.

There are many possible distributions which fulfill this requirements (some candidates are depicted in Figure 3). The scheduling mechanism attempts to find an available local network environment (cluster or LAN) first, where all the processes can be executed (see Figure 3A). If it is not possible, the scheduling mechanism attempts to find a distribution which takes into account the minimum number of groups and fits to the current physical grid architecture (see Figure 3B and C).

4.3 Groups with Fixed Sizes

The schema *fixed-groups* is a similar structure to the schema groups, but here we can define the number of groups (first argument) and the precise number of the processes in each group respectively (further arguments):

$$FIXED - GROUPS\{nrOfGroups, [sizeOfGroup_1...sizeOfGroup_N]\}$$

Example In the following specification, we intend to schedule 18 processes

$$FIXED - GROUPS\{3, 5, 6, 7\}$$

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

4.4 Multi Level Parallelism – Tree

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

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

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

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

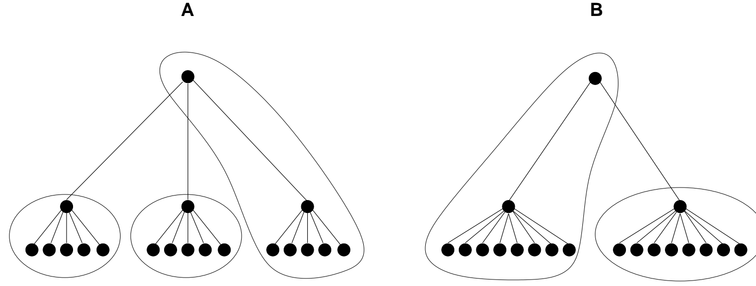

Figure 4: Some Distributions satisfying the Specification $TREE\{19, 3, 5\}$

Example If we take the following specification

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

then we can say that similarly to the schema groups we have more than one possible tree distribution which fulfill the given requirements (see Figure 4). The scheduling mechanism tries to distribute this tree structure of processes as optimal as possible (first into one local network environment and if it is not possible then int two or three ones (as it is showed by the Figure 4A and B). In addition, it attempts to place the global managers close (in terms of latency and bandwidth) to its children processes.

4.5 A Planned Addition: Ring

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

$$RING\{nr, minSizeOfGroups\}$$

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

The main reason why the schema ring is part of the proposed software system is that it represents a promising extension of the algorithmic structure specifications discussed above where not only the expectation against the local groups of participating processes are defined but the interacting activities among these groups are taken into consideration (this may lead some advantages for instance in the case of a P2P-based grid architecture).

5 The Mapping between Specifications and Physical Grid Architectures

The scheduling mechanism accomplishes the mapping between the specification of the algorithmic structure of a program and a physical grid architecture with the help of the measured latency (and in a later phase of the project, the bandwidth and CPU usage) values.

The expected input is an algorithmic structure specification presented in the previous section and the output is an execution plan described in the following two files:

Globus RSL script file is necessary for starting MPI programs via the gridified mpirun solution of MPICH-G2 and for preserving the topology-aware collective operations provided by MPICH-G2. In our case, this file is organized into jobs and subjobs, such that it describes a two levels structure of the distribution of concurrent processes on a particular grid architecture.

XML-based Mapping Description file is used by our proposed topology-aware API to assign the processes of the program to the allocated grid resources. This file describes the mapping between the processes and the grid architecture similarly to the RSL script, but in a more sophisticated and flexible way. For the exact structure of this file, see Section 6.

However these files contain redundant information about the distribution of the processes on the grid, but both of them are needed in the currently proposed version of our software system. In a later phase, MPICH-G2 may be substituted for another solution for executing MPI programs on the grid and the RSL script will perhaps not required anymore.

Later the scheduling mechanism (and both output file as well) can be refined further by querying the grid information system MDS and by determining on which machines (clusters) some vendor-MPI softwares are deployed.

6 XML-based Execution Plan

The XML-based mapping description file is a generated execution plan which consists of the following major components:

$$Mapping\{prgName, timeStamp, type, prgStructure, networkTopology\}$$

The first component is the name (or unique identifier) of the parallel program, the second one is a time stamp restricting the validity of the current mapping and the third one is the type of the given program structure (singleton, setOfGroups, tree). The fourth component is a precise *description of the program structure* adapted to physical network architecture (e.g.: the sizes of the groups are already fixed by the scheduling mechanism), see Section 6.1. The fifth component comprises a topology description of the computing resources going to be allocated and the assignment of these resources to particular processes, see Section 6.2.

6.1 Description of Program Structures

The program structure description is part of a particular execution plan (the output of the scheduling mechanism) and it describes a precise program structure which meets with the specification and also adapted to the topological hierarchy of the available grid architecture at the same time.

6.1.1 Singleton

The component *singleton* contains the number of the participating processes and a list of their identifiers.

$$singleton\{nrOfProcs., [listOfPID]\}$$

6.1.2 Groups, Fixed-Groups and Ring

The component *setOfGroups* is used to describe program structure specified by the schemas groups, fixed-groups and ring and it contains the number of the included groups, the minimum size of the groups and an embedded enumeration of these groups. The description of such a group consists of a unique group identifier, the number of processes belongs to this group and the list of identifiers of the participating processes.

$$
\begin{aligned}
&setOfGroups\{nrOfGroups, minSizeOfGroups, [\\
&\quad group\{grpID, nrOfProcs., [listOfPID]\},\\
&\quad ...,\\
&\quad group\{grpID, nrOfProcs., [listOfPID]\},\\
&]\}
\end{aligned}
$$

6.1.3 Tree

The component *tree* is applied to describe a hierarchy of concurrent processes which meets with the conditions specified in the schema tree. It contains the identifier of the root process, the depth of the tree, a minimum number of worker processes in a local group, the number of children of the root and the embedded enumeration of its subtree components (or its worker components depending on the depth of the tree).

$$
\begin{array}{l}
tree\{rootPID, depth, minSizeOfLeafGroups, nrOfChildren, [\\
\quad subTree\{nodePID, nrOfChildren, [\\
\quad\quad group\{grpID, nrOfProcs., [listOfPID]\},\\
\quad]\},\\
\quad ...,\\
\quad subTree\{...\}\\
]\}
\end{array}
$$

A component subtree contains the identifier root process of the subtree and the number of its children. The subtrees can be embed into each other until an arbitrary level (depending on the depth of the tree). Within the inner most subtree components, a group component (see Section 6.1.2 is defined which contains the list of worker processes (which are the leaves of the tree).

6.2 Description of Mapped Topologies

The component structure presented below has two major parts:

$$wan\{topology, grp2grpLatencies\}$$

The component *topology* is used to couple the process identifiers occurred in a given program structure (see Section 6.1) the grid resources going to be allocated and to provide additionally some network topology information related to these resources.

$$
\begin{array}{l}
topology\{[\\
\quad host\{address, nrCPUs, nrProcesses, [listOfPID]\}\\
\quad ...,\\
\quad host\{...\},\\
\quad lan\{avgLatency, [\\
\quad\quad host\{address, nrCPUs, nrProcesses, [listOfPID]\},
\end{array}
$$

$...,$

$host\{...\},$

$]\},$

$...,$

$lan\{...\}$

$]\}$

In order to improve the flexibility of our topology-aware API, the component *lan* (embedded into the component topology) contains latency (and perhaps bandwidth) information concerning the corresponding network environments. A component *host* can be embedded either into the component topology directly or into a component lan and it describes a chosen grid node with its network address, the number of its CPUs, the number of processes intended to run on it and a list of process identifiers occurred already a given program structure description.

The component *grp2grpLatencies* consists of an enumeration of the identifiers of the previously defined groups (see Section 6.1.2 and Section 6.1.3) ordered in pairs such that all combinations of the possible pairs occur. For each pair, the enumeration contains the average latency value between the two given groups and a flag which can be true or false such that the two groups are supposed to interact with each other according to the given schema (the value of the flag can be true only in the case of a ring of groups schema at the moment).

$grp2grpLatencies\{[$

$latency\{grpID_0, grpID_1, avgLatency, flag\},$

$...,$

$latency\{grpID_0, grpID_n, avgLatency, flag\},$

$...,$

$latency\{grpID_{n-1}, grpID_n, avgLatency, flag\},$

$]\}$

7 Resource Allocation and Execution

In the first version of our planned software system, the allocation of the chosen grid resources and the starting of a parallel program on the grid will be performed by the runtime system of the MPICH-G2 (under Globus Toolkit). This requires that the instances of the program must be deployed and compiled on the chosen grid resources. This runtime system expects a topology description formalized in a RSL file as input (which is generated by a our scheduling mechanism, see Section 5).

The usage of the MPICH-G2 also provides some topology-aware collective operations based on the content of the RSL file.

8 The Topology-Aware API

The proposed API is an addition to the MPI library 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 mapping description file (which was generated by the scheduling mechanism). A detailed description of this API is presented below (but further refinement is possible in the near future).

8.1 Header File

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

8.2 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*.

8.3 Calls wrt. Initialization and Termination

TAAG_Init (char *prg_str, char *exec_plan_file) allocates and initializes the corresponding data structures according to the generated execution plan file given in the second argument (see Section 6). The first argument is the name of the program structure (e.g.: singleton, groups, tree, etc.) described by the execution plan file. 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 (1) or false(0).

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

8.4 Calls wrt. Topology Structure

TAAG_GetCommLevel (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 $commLevel = 0$ then the two processes can interact each other only via WAN, but if $commLevel = 1$ they are located in the same LAN network and if $commLevel = 2$ they nest on the same host.

TAAG_GetProcsOnHost (int rank, int *nr, *procs) requires a process rank as input and it returns the number and ranks of all the processes residing on the same host.

TAAG_GetProcsOnLAN (int rank, int *nr, *procs) requires a process rank as input and (similarly with the previous call) it returns the number and ranks of all the processes residing on the same LAN.

TAAG_GetGrpLatency (int grpRank, double *latency) requires a rank of a group (see below in Section 8.5) as input and it returns the average latency value in the given group.

TAAG_GetGrp2GrpLatency (int grpRank1, int grpRank2, double *latency) requires the ranks of two groups (see below in Section 8.5) as input and returns the average latency value between the two groups.

TAAG_GetHostAddress (int rank, int *size, char *address) requires a process rank as input and returns the address of the host where the process resides.

TAAG_GetHostCPUs (int rank, int *nrCPUs) requires a process rank as input and returns the number of CPUs on the host where the given process resides.

8.5 Calls wrt. the Program Structure Group

Each group has a unique rank assigned by our library when the group initializes. A *group rank* (similarly to the process ranks) is an integer number and its scope start with 0 and ends with less but one than the number of groups.

TAAG_GetNrOfGrps (int *n) returns the number of groups contained by the execution plan.

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

TAAG_GetGrpMinSize (int grpRank, int *minSize) requires a group rank as input and returns the minimum size (minimum number of processes) of this group which was given in the schema specification.

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

TAAG_CreateMPIStructsForGrp (int grpRank, MPI_Group grp, MPI _Comm *comm) requires a group rank as input and creates the corresponding MPI_Group and MPI_Comm structures for the given group. This call is useful if some MPI collective operations are going to be used within the frame of the group (only for those processes which are members of the given group).

TAAG_GetConnectedGroups (int grpRank, int *nr, int *list) requires a group rank as input. If the execution plan defines which groups are planned to interact each other (e.g.: in the case of a schema ring), then this

call returns the number and the ranks of the "neighbor" groups of the given one.

8.6 Calls wrt. Program Structure Tree

The following section presents some calls which are related to the program structure tree. Each execution plan can describe one tree at most.

In a tree, the leaves ordered in some program structure groups (the leaves which belong to the same parent compose a group). Hence, every call presented in Section 8.5 above can be applied for these groups.

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

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

TAAG_GetNrOfLeaves (int *leaves) returns the number of leaf/worker processes in the entire tree.

TAAG_GetLevel (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_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 (1) or false (0).

TAAG_IsLocalMngr (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 (1) or false (0).

TAAG_GetChildren (int rank, int *nr, int *children, int *commLevel) requires a process rank and returns the number and the ranks of the children of the given process. The last parameter is an array as well and indicates on which network level a child can communicate with its parent. If $commLevel = 0$ then the two processes can interact each other only via WAN, but if $commLevel = 1$ they are located in the same LAN network and if $commLevel = 2$ they nest on the same host.

TAAG_GetParent (int rank, int *parent, int *commLevel) requires a process rank and returns the rank of the parent of the given process. The last parameter indicates on which network level the process can communicate with its parent (similarly to the call TAAG_GetChildren above).

TAAG_CreateMPIStructsForAllLvs (MPI_Group *grp, MPI_ Comm *comm) creates MPI_Group and MPI_Comm structures which include all leaf processes. This call is useful if some MPI collective operations are going to be used for all leaf processes.

TAAG_CreateMPIStructsForAllNonLvs (MPI_Group *grp, MPI _Comm *comm) creates MPI_Group and MPI_Comm structures which include all non-leaf processes. This call is useful if some MPI collective operations are going to be used for all non-leaf processes (i.e. root, schedulers, local schedulers).

8.7 Channels

The channels are proposed communication objects between two groups. Their goals are:

- to allow to make some collective operations (e.g.: broadcast) from a process of one group to all processes of another group
- to allow interactions among the members of two groups via different protocols (e.g.: SOAP, gridftp).
- to allow the usage of private networks (with non-public IPs).

A channel is always defined between two designated processes of the two groups. Other processes sends messages via the channel by the assistance of these two processes.

TAAG_Ch_Init (int rank1, int rank2, char *protocol, TAAG_ Ch *ch) establishes a channel between the two given groups and returns a channel handle, a pointer to TAAG_Ch struct. We can specify the communication protocol for the channel as well (e.g.: SOAP, gridftp, etc).

TAAG_Ch_Free (TAAG_Ch *ch) deallocates the given existing channel.

TAAG_Ch_Send (...) performs a blocking send from a process of a group to another process of another group via a predefined channel. Details are not clarified yet.

TAAG_Ch_Isend(...) performs a non-blocking send from a process of a group to another process of another group via a predefined channel. Details are not clarified yet.

TAAG_Ch_Bcast(...) performs a broadcast from a process of a group to all processes of another group via a predefined channel. Details are not clarified yet.

8.8 Some Examples

The following simple parallel programs gives some examples for usage of our proposed API.

8.8.1 An Example for Groups

The following example is a very artificial (dummy) program, but it represents well how to use the proposed groups-related calls described in Section 8.5.

In this example the program consists of even number of local groups (which is independent from the number and distribution of processes) and these groups are arranged in two pools according as their ranks are odd or even. See the comments related to the source code below.

```
01: #include "mpi.h"
02: #include "taag.h"
03: #include <stdio.h>
04:
05: #define BUFFER_SIZE 255
06:
07: int main(int argc, char *argv[]) {
08: int rc, nrMembers, nrGrps, numTasks;
09: int globalRank, localRank, grpRank;
10:
11: int members[BUFFER_SIZE];
12: int msg[BUFFER_SIZE];
13:
14: MPI_Status stat;
15: MPI_Request req;
```

```
16: MPI_Group mpiGrp;
17: MPI_Comm mpiComm;
18:
19: rc = MPI_init(&argc, &argv);
20: if (rc != MPI_SUCCESS) {
21:   printf("Error starting MPI program.\n");
22:   MPI_Abort(MPI_COMM_WORLD, rc);
23: }
24:
25: rc = TAAG_Init("groups", "exec_desc.xml");
26: if (rc != TAAG_SUCCESS) {
27:   printf("Error initializing the TAAG structure.\n");
28:   MPI_Abort(MPI_COMM_WORLD, rc);
29: }
30:
31: TAAG_GetNrOfGrps (&nrGrps);
32: if (nrGrps % 2 != 0) P
33:   printf("Error: the number of groups is odd.\n");
34: }
35: else {
36:   MPI_Comm_size(MPI_COMM_WORLD, &numTasks);
37:   MPI_Comm_rank(MPI_COMM_WORLD, &globalRank);
38:
39:   TAAG_GetGrpRank(globalRank, &grpRank);
40:   TAAG_GetGrpMembers(grpRank, &nrMembers, members);
41:
42:   TAAG_CreateMPIStructsForGrp(grpRank, &mpiGrp, &mpiComm);
43:   MPI_Group_rank (mpiGrp, &localRank);
44:
45:   if (grpRank % 2 == 0) {
46:     /***** branch for the groups whose ranks are even *****/
47:     if (globalRank == members[0]) { //(localRank == 0) is the same
48:       /***** local manager branch *****/
49:
50:       MPI_Recv(msg, BUFFER_SIZE, MPI_CHAR,
51:         MPI_ANY_SOURCE, MPI_ANY_TAG, mpiComm, &stat);
52:
53:       int nr, membersOddGrp[BUFFER_SIZE];
54:       TAAG_GetGrpMembers (grpRank-1, &nr, membersOddGrp);
55:       MPI_Send(msg, BUFFER_SIZE, MPI_CHAR, membersOddGrp[0],
56:         0, MPI_COMM_WORLD);
```

```
57:     }
58:     else {
59:       /***** DO SOME WORK HERE *****/
60:       MPI_Isend(msg, BUFFER_SIZE. MPI_CHAR, 0, localrank, mpiComm, &req);
61:     }
62:   }
63:   else {
64:     /***** branch for the groups whose ranks are odd *****/
65:
66:     if (globalRank == members[0]) { //(localRank == 0) is the same
67:       /***** local manager branch *****/
68:       MPI_Recv(msg, BUFFER_SIZE, MPI_CHAR,
69:         MPI_ANY_SOURCE,  MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
70:       MPI_Bcast(msg, BUFFER_SIZE, MPI_CHAR, 0, mpiComm);
71:
72:     }
73:     else {
74:       MPI_Bcast(msg, BUFFER_SIZE, MPI_CHAR, 0, mpiComm)
75:
76:     }
77:   }
78: }
79:
80: TAAG_Free();
81: MPI_Finalize();
82: }
```

Comments:

lines 01–03 comprise the required includes.

line 25 allocates and initializes the corresponding data structures for a program structure "groups" according to the content of the file called "exec_desc.xml".

lines 31–34 check whether the program consists of even number of groups.

line 39 determines the group rank for the given process.

line 40 determines the number and the ranks of the processes in the given group.

line 42 creates structures MPI_Group and MPI_Comm for the given group.

line 43 determines the local rank of the calling process in the given MPI_Group.

lines 45–62 describe the behavior of the groups whose ranks are even.

lines 47–57 describe the behavior of the local manager processes of the groups whose ranks are even. It waits a message from one of the workers of the group. If the message arrives it determines the global rank of the local manager of that group whose rank is less by one (odd) than its own rank and forwards the received message.

lines 58–61 describe the behavior of the worker processes of the groups whose ranks are even. They simply send a message to their local manager (after some work).

lines 63–77 describe the behavior of the groups whose ranks are odd.

lines 66–72 describe the behavior of the local manager processes of the groups whose ranks are odd. It waits until a message arrives from another local manager and broadcasts this message its group.

lines 73–76 describe the behavior of the worker processes of the groups whose ranks are odd. They simply wait for the broadcast.

line 80 deallocates the data structures applied by our library.

8.8.2 An Example for a Tree

The following example represents how to use the proposed tree-related calls described in Section 8.6 This source code can be conjugated with the tree specification example described in Section 4.4:

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

In this program a global manager process distributes some computational tasks among some local manager processes which distribute them further among the worker processes. After a worker accomplished a task it sends back to the global manager through its local manager.

Remark: This program assumes (and runs correctly) that the used blocking message passing routines (MPI_Send and MPI_Recv) are implemented with usage of system buffering.

```
01: #include "mpi.h"
02: #include "taag.h"
03: #include <stdio.h>
04:
05: #define BUFFER_SIZE 255
06: #define NR_OF_TASKS 500
07: extern int create_task(int, char*);
08: extern int process_task(char*, char*);
09:
10: int main(int argc, char *argv[]) {
11:   int numtasks, rc, flag, nrChildren;
12:   int rank, root, parent, level;
13:
14:   int children[BUFFER_SIZE];
15:   int levels[BUFFER_SIZE];
16:   int inmsg[BUFFER_SIZE], outmsg[BUFFER_SIZE];
17:
18:   MPI_Status stat;
19:
20:
21:   rc = MPI_Init(&argc,&argv);
22:   if (rc != MPI_SUCCESS) {
23:     printf("Error starting MPI program.\n");
24:     MPI_Abort(MPI_COMM_WORLD, rc);
25:   }
26:
27:   rc = TAAG_Init("tree", "exec_desc.xml");
28:   if (rc != TAAG_SUCCESS) {
29:     printf("Error initializing the TAAG structure.\n");
30:     MPI_Abort(MPI_COMM_WORLD, rc);
31:   }
32:
33:   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
34:   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
35:
36:   TAAG_GetRoot(root);
37:
38:   if (rank == root) {
39:      /***** root branch *****/
40:      int output[NR_OF_TASKS][BUFFER_SIZE];
41:      MPI_Request reqs[NR_OF_TASKS];
```

```
42:      MPI_Status stats[NR_OF_TASKS];
43:      TAAG_GetChildren(rank, &nrChildren,
44:           children , levels);
45:      int j = 0;
46:
47:      for (int i = 0; i < NR_OF_TASKS; i++) {
48:        create_task(i, outmsg);
49:        MPI_Irecv(output[i], BUFFER_SIZE, MPI_CHAR,
50:              children[j], i, MPI_COMM_WORLD, &reqs[i]);
51:        MPI_Send(outmsg, BUFFER_SIZE, MPI_CHAR,
52:              children[j++], i, MPI_COMM_WORLD);
53:        if (j == nrChildren) { j = 0; }
54:      }
55:
56:      MPI_Waitall (NR_OF_TASKS, reqs, stats)
57:
58:      for (int i = 0; i < NR_OF_TASKS; i++) {
59:         printf("The solution of the %d. task is \"%s\".\n",
60:                 stats[i].MPI_TAG, output[i]);
61:      }
62:   }
63:   else {
64:     /***** non-root branch *****/
65:     TAAG_GetParent(rank, &parent, &level);
66:     TAAG_IsLeaf(rank, flag);
67:     if (flag == 0) {
68:     /***** non-leaf branch *****/
69:       int j = 0;
70:       TAAG_GetChildren(rank, &nrChildren,
71:            children, levels);
72:       while(1) {
73:         MPI_Recv(outmsg, BUFFER_SIZE, MPI_CHAR,
74:          MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
75:         if (stat.MPI_SOURCE == parent) {
76:            MPI_Send(outmsg, BUFFER_SIZE, MPI_CHAR,
77:               childen[j++], stat.MPI_TAG, MPI_COMM_WORLD);
78:            if (j == nrChildren) { j = 0; }
79:         }
80:         else {
81:           MPI_Send(outmsg, BUFFER_SIZE, MPI_CHAR,
82:              parent, stat.MPI_TAG, MPI_COMM_WORLD);
```

```
83:         }
84:       }
85:     }
86:     else {
87:     /***** leaf branch *****/
88:       while(1) {
89:         MPI_Recv(inmsg, BUFFER_SIZE, MPI_CHAR,
90:            parent, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
91:         process_task(inmsg, outmsg);
92:         MPI_Send(outmsg, BUFFER_SIZE, MPI_CHAR,
93:             parent, stat.MPI_TAG, MPI_COMM_WORLD);
94:       }
95:     }
96:   }
97:
98:   TAAG_Free();
99:   MPI_Finalize();
100:}
```

Comments:

lines 01–03 comprise the required includes.

line 07 defines an external function called *create_task* which returns a string description of the subsequent computational task.

line 08 defines an external function called *process_task* whose input is a previously mentioned task description and whose output is the outcome of this task (in string format).

line 27 allocates and initializes the corresponding data structures for a program structure "tree" according to the content of the file called "exec_desc.xml".

line 36 determines the root process of the tree hierarchy.

lines 38–62 describe the behavior of the root process of the tree. In line 43 it determines number and 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.

line 65 determines the parent process of the current non-root process in the tree.

line 66 decides whether the current process is a leaf in the tree.

lines 67–85 describe the behavior of the local scheduler processes in the tree. In line 70 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.

lines 86–95 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.

line 98 deallocates the data structures applied by our library.

9 Development Plan

The development of the software system described in the previous sections are planned as follows:

1.10.2008 — 31.3.2009 In this period, we work on the first skeleton prototype which contains the implementations of the API calls described in Section 8.3, Section 8.5, Section 8.6 and additionally most of the API functions described in Section 8.4. Furthermore, we develop the initial version of the deployment mechanism (see Section 3.1).

1.4.2009 — 30.9.2009 In this period, we work on the second skeleton prototype that contains all the major components of the software system described in this document. We develop the proposed scheduling mechanism (see Section 3.1) and improve the implementation of the deployment mechanism. We finish the implementation of the API calls described in Section 8.4.

1.10.2009 — Based on our first experiences with the system we refine the design and implement an initial version of the SOAP– and gridftp–based communication calls of the proposed API (see Section 8.7).

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] MPICH-G2 Project Home Page. http://www.hpclab.niu.edu/mpi/.

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

[4] NetSolve/GridSolve Project Home Page. http://icl.cs.utk.edu/netsolve.

[5] 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., 2008.

[6] Pushpinder Kaur Chouhan, Holly Dail, Eddy Caron, and Frédéric Vivien. Automatic middleware deployment planning on clusters. *Int. J. High Perform. Comput. Appl.*, 20(4):517–530, 2006.

[7] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto Junior, and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing. *Concurrency and Computation: Practice and Experience*, 17(2-4), 2005. http://dps.uibk.ac.at/askalon/.

[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] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. *ACM SIGPLAN Notices*, 34(8):131–140, 1999.

[10] Arnaud Legrand, Hélène Renard, Yves Robert, and Frédéric Vivien. Mapping and load-balancing iterative computations. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):546–558, 2004.

[11] 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.

[12] S. Pakin and A. Pant. Vmi 2.0: A dynamically reconfigurable messaging layer for availablility, usability and management. 2002.

[13] Avneesh Pant and Hassan Jafri. Communicating efficiently on cluster based grids with mpich-vmi.

[14] Radu Prodan, Thomas Fahringer, Farrukh Nadeem, and Marek Wieczorek. Real-world workflow support in the askalon grid environment. In *CoreGRID Workshop on Grid Middleware*, Dresden, Germany, June 2007. Springer Verlag.

[15] Jun Qin, Marek Wieczorek, Kassian Plankensteiner, and Thomas Fahringer. Towards a Light-weight Workflow Engine in the ASKALON Grid Environment. In *Proceedings of the CoreGRID Symposium*, Rennes, France, August 2007. Springer-Verlag.

[16] 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.