MPIXternal: A Library for a Portable Adjustment of Parallel MPI Applications to Heterogeneous Environments

Carsten Clauss, Stefan Lankes, Thomas Bemmerl
Chair for Operating Systems, RWTH Aachen University
Kopernikusstr. 16, 52056 Aachen, Germany
{clauss, lankes, bemmerl}@lfbs.rwth-aachen.de

Abstract

Nowadays, common systems in the area of high performance computing exhibit highly hierarchical architectures. As a result, achieving satisfactory application performance demands an adaptation of the respective parallel algorithm to such systems. This, in turn, requires knowledge about the actual hardware structure even at the application level. However, the prevalent Message Passing Interface (MPI) standard (at least in its current version 2.1) intentionally hides heterogeneity from the application programmer in order to assure portability. In this paper, we introduce the MPIXternal library which tries to circumvent this obvious semantic gap within the current MPI standard. For this purpose, the library offers the programmer additional features that should help to adapt applications to today's hierarchical systems in a convenient and portable way.

1. Introduction

Today, hierarchical hardware architectures have become prevalent in the area of high-performance computing. Examples range from Grid-related wide-area computing and cluster-of-clusters, through ccNUMA systems and NoRMA clusters with shared-memory nodes, to multi-core-related cache level hierarchies. If one wants to exploit such systems at their best, an adaptation of the parallel algorithms to the respective hardware structure becomes inevitable, because otherwise the system's bottlenecks (e.g. in terms of communication) will limit the overall performance. Thus, a likewise *hierarchical* algorithm design will accommodate such systems. This design approach seeks to consolidate information at different levels of the hardware hierarchy in order to reduce the message traffic and to avoid the congestion of bottlenecks [7]. As a result, this approach for optimization requires knowledge about the respective hardware structure either within the application and/or inside the communication library, as for example within an MPI implementation [12].

In order to assure portability, the MPI standard intentionally hides the actual hardware structure from the application programmer, at least in its current version 2.1 [9]. Due to this, it should be the responsibility of every MPI implementation to exploit the underlying target system as efficiently as possible [14]. However, in doing so, the MPI runtime system would have to be aware of the properties of the actual parallel algorithm. Although the standard offers such a notifying mechanism in terms of the so-called *process topologies*, for various reasons, this *top-down*-related approach is not realized in most common MPI applications, as well as not implemented beneficially in most common MPI libraries. For that reason, many heterogeneity-aware MPI implementations provide the application programmer with additional adaptation features that follow a *bottom-up* approach. This means that these MPI implementations offer *non*-standard-conforming methods for passing hardware-related information up to the application level. At this point, a semantic gap in the current MPI standard concerning today's hierarchical systems becomes obvious: Since an MPI implementation cannot handle the adaptation of parallel algorithms to the target systems, the application programmer has to perform this, though lacking standard-conforming methods.

In this paper, we introduce a new additional library, called MPIXternal, that attempts to circumvent this semantic gap by providing programmers and users with additional features for supporting the optimization-related flow of information in both directions: up to the application as well as down to the respective MPI implementation. This is achieved by giving the programmer the ability to introduce hierarchy-awareness into the application, while the respective semantic of this awareness can be adjusted to each target environment by the user via an XML-coded configuration file. Figure 1 should illustrate this approach.

The remainder of the paper is organized as follows: First, we will discuss some useful extensions to the *communicator*

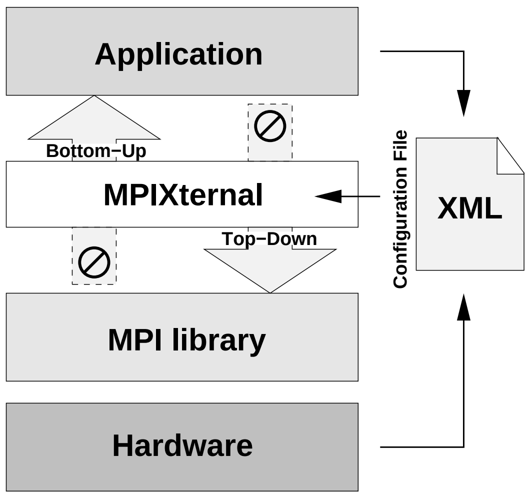

Figure 1. MPIXternal in the Layer Model

and *attribute* concept of MPI according to the *bottom-up* approach. Then, we will focus on topology mapping, as it is also provided by the MPI's process topology mechanism, and then go on to discuss how our library can accommodate its utilization in both directions. Finally, a brief summary and some prospects for the future conclude the paper.

2. Predefined MPI Communicators

An important feature of the Message Passing Interface is the communicator concept. This concept allows the application programmer to group the parallel processes by assigning them to abstract objects called *communicators*. For that purpose, the programmer can split the group of initially started processes into sub-groups, each forming a new self-contained communication domain. This concept usually follows the *top-down* approach where the process groups are built according to the communication patterns required by the parallelized algorithm.

2.1 Heterogeneity-aware MPI

However, this grouping concept can also be exploited in heterogeneous environments in order to group the processes according to a given hardware hierarchy. Although the implementation of such a hardware-oriented splitting procedure can also be done at the application level, it is usually not a good idea to burden the application programmer with this issue. This is because otherwise the programmer finds himself/herself faced with the task of inquiring explicitly after all needed environmental information (e.g. by querying the respective hostnames) and assuring the determination of an appropriate splitting scheme on her/his own.

For this reason, heterogeneity-aware MPI implementations usually accommodate the application programmer with additional predefined MPI communicators with group affiliations reflecting the actual hardware structure. Thus, by providing such additional communicators, an MPI implementation offers the programmer an adaptation feature following the *bottom-up* approach, where the topology information is passed via the MPI layer up to the application level in an abstracted manner. Although the application must still be written to be heterogeneity-aware in terms of utilizing those additional communicators, the application programmer is absolved from dealing with tangible hardware characteristics like hostnames, for example.

On the other hand, the MPI standard (at least in its current version 2.1) does not specify any other valid predefined communicators than the common `MPI_COMM_WORLD` and `MPI_COMM_SELF`. That means that heterogeneity-aware MPI implementations which provide such adaptation features of additional communicators are moving beyond the standard. This in turn means that an MPI application which makes use of such a feature becomes tied to this implementation and thus becomes less portable.

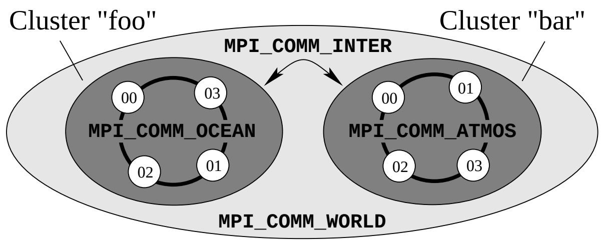

Figure 2. Two Linked Clusters

2.2 The MPIXternal Library

For that reason, and as already mentioned in the introduction, we have developed a new library that allows an application programmer to develop heterogeneity-aware applications without the need of relying on non-portable optimization features provided by a particular MPI implementation. This is accomplished by introducing a new and generalized approach for predefined MPI communicators whose semantics are specified externally within a suitable XML-coded configuration file. For this purpose, the library provides a special function that expects a reference to an uninitialized MPI communicator object as well as the name of this object. By means of this passed communicator name, the function can then look up a table containing the descriptions of all previously defined communicators being specified within the external configuration file. If an inquired communicator can be found in this table, each calling process checks whether it takes part in this predefined communicator or not. This identification can be done by means of comparing the own *processor name* (as returned by an internal call to the `MPI_Get_processor_name()` function) against a list of node names associated with that communicator. Afterwards, all included processes will build the inquired communicator, whereas the non-included ones will report their absence in reference to that new communicator.

2.3 An External Configuration File

For purposes of illustration, consider the example in Listing 1 and the associated Figure 2 of a communicator configuration which may be written for a *coupled-code* simulation on a hierarchical system consisting of two linked clusters. Here, the first communicator, named `MPI_COMM_OCEAN`, should cover all processes running on nodes belonging to a cluster named `foo`. This can be achieved by matching the hostnames against the stated cluster name (which is in fact a regular expression) because in this example it is assumed that the node names of the two clusters are composed in a simple `cluster_name:node_name` manner.

```
<comm name="MPI_COMM_OCEAN">
  <attribute key="DEPTH" value="3.8km"/>
  <cluster name="foo">
    <attribute key="CPU_SPEED" value="2.2"/>
  </cluster>
</comm>

<comm name="MPI_COMM_ATMOS">
  <attribute key="PRESSURE" value="101.325kPa"/>
  <node name="bar:00">
    <attribute key="CPU_SPEED" value="2.0"/>
  </node>
  <node name="bar:01>
    <attribute key="CPU_SPEED" value="2.2"/>
  </node>
  <node name="bar:02">
    <attribute key="CPU_SPEED" value="1.8"/>
  </node>
  <node name="bar:03">
    <attribute key="CPU_SPEED" value="2.0"/>
  </node>
</comm>

<intercomm name="MPI_COMM_INTER">
  <first>  MPI_COMM_OCEAN </fisrt>
  <second> MPI_COMM_ATMOS </second>
</intercomm>
```

Listing 1. Example Configuration

However, the second communicator in this example (`MPI_COMM_ATMOS`) is configured for containing just those processes running on the explicitly stated nodes `bar:00` up to `bar:03`. Thus, both the types of notation are supported by MPIXternal. In addition, also *inter*-communicators, as they are defined by the MPI standard for communication between *remote* groups of processes, can easily be specified within the external configuration file, as shown here in Listing 1 for `MPI_COMM_INTER`.

The relocation of the communicator definitions into an external configuration file offers several advantages and opportunities. For example, the application does not need to be recompiled if the desired grouping scheme or the processor names have changed. Furthermore, the configuration does not necessarily need to be written by a user or an application programmer. In fact, the XML file containing the communicator definitions can be generated, for example, by a process scheduler, by a topology analyzing tool or even by the MPI runtime environment itself.

2.4 Usage at Application Level

As already mentioned, the MPIXternal library (that can be used in addition to any MPI library) provides new and additional functions that aim to extend the MPI standard by the feature of externally adjustable semantics for MPI objects like communicators. An example for utilizing this feature at application level is shown in Listing 2, where an externally defined communicator (`MPI_COMM_OCEAN`) is built by a call to such a new function, namely `MPIX_Comm_external()`.

```
#define MPIX_Comm_external_(a,b) \
        MPIX_Comm_external(&a,#a,b)

#define MPIX_Keyval_external_(a,b) \
        MPIX_Keyval_external(&a,#a,b)

MPI_Comm MPI_COMM_OCEAN = MPI_COMM_NULL;
MPIX_Comm_external_(MPI_COMM_OCEAN, &flag);

if (flag) {
   /* I am part of MPI_COMM_OCEAN! */

   int DEPTH = 0;
   MPIX_Keyval_external_(DEPTH, &flag);

   if (flag) {
      /* Found attribute key for "DEPTH"! */

      char* value = NULL;
      MPI_Attr_get(MPI_COMM_OCEAN, DEPTH,
                   &value, &flag);

      if (flag) {
         /* Got attribute value for "DEPTH"! */

         printf("Ocean is %s deep.\n", value);
         . . .
```

Listing 2. Usage at Application Level

One may argue that using this extending API would cause the respective application to become tied to our additional library, just like when using other non-portable optimization features. However, first of all, our solution is independent from the kind of the underlying MPI implementation, and that in turn means that our approach is portable in such a way that it can be applied in any system providing an MPI runtime environment. Secondly, as we have already described in a former publication (see [4]), it is feasible that even when using our additional library, an application can still be written in a standard-conforming manner. For this purpose, our library offers two APIs: one that provides new functions (and which are therefore not standard-conforming) and one that covers common MPI function calls in such a manner that the externally defined semantics are attached in a *transparent* way. However, to simplify matters, we will focus on the former case of the new and additional API in the remainder of this paper.

3. MPI Attributes and Info Objects

Another frequently used way for passing environmental information from an MPI implementation to the application level is to utilize so-called *attributes*. In terms of MPI, such attributes belong to a ”caching” facility provided by the respective MPI implementation that allows an application to attach arbitrary pieces of information to MPI objects like communicators (see [9], p.221). Even though this facility has not been introduced to the standard with the actual intent of passing environmental information from an MPI implementation to an application, it still can be exploited for that purpose, for example by attaching additional *predefined* attributes to `MPI_COMM_WORLD`. In fact, even the standard defines a set of such predefined attributes that describe the actual execution environment of an MPI job. Furthermore, the standard also suggests that MPI vendors may add their own implementation specific attributes to this set (see [9], p.260). However, since the symbol names of such additional attributes are set at the compile time of the respective library, using them at the application level is not very portable.

For this reason, we have extended our library with the ability to attach *self-predefined* attributes to all externally defined communicators (and even to `MPI_COMM_WORLD`). That way, an application programmer can inquire after the values of those attributes via the functionalities provided by MPIXternal in an MPI implementation independent manner, whereas the application user is able to adjust the actual values to each respective environment later on. Moreover, the user can also exploit this mechanism in order to pass job-dependent parameters to the application in a topology- and group-oriented manner.

For illustration purposes, please consider Listing 1 and Listing 2 once again. As one can see, attribute keys and values are stated as *strings* in the configuration file. In contrast to this, the attribute retrieve functions of MPI (`MPI_Attr_get()`, or `MPI_Comm_get_attr()` in the case of MPI-2) expect a unique *integer* value identifying the respective attribute and return *void*-pointers to arbitrary data types. As a result, a prior attribute initialization via a call of `MPIX_Keyval_external()` becomes necessary, while the later returned value should be interpreted as a pointer to the stated value in the form of a *string*.

A further caching mechanism that has become available along with the MPI-2 standard is formed in terms of the so-called MPI *info* objects (see [9], pp.287ff). In contrast to MPI attributes, such an info object stores an unordered set of *strings*, each retrievable by an also string-coded key. For that reason, this mechanism may also be a practical way for passing runtime information to the application level in the form of such key/value-pairs. However, the normal usage of such info objects is to pass information in the other direction according to the *top-down* approach. That means that these objects have been introduced into the standard in order to provide MPI implementations with more information about the actual needs of the current application. Nevertheless, the other way around is also possible, and because of this, our library also supports an external predefinition of such info objects. Additionally, our implementation also provides a conversion feature that allows a programmer to demerge such an info object from the attributes of a communicator, and vice versa. However, since the usage of info objects instead of attributes is quite similar within the context of our library, we do not provide any code examples here.

4. Rank Reordering and Topology Mapping

In this section, we will focus on another important aspect in heterogeneous environments: *rank-reordering* in terms of *topology mapping*. This technique arises from the fact that in many parallel applications a linear ranking order of the processes does not adequately reflect their actual communication patterns. However, when splitting an existing communicator into separated sub-groups, the initial ranking order of the processes does not necessarily need to be retained. Particularly in the case of a hardware topology that still exhibits an non-homogeneous (or at least a highly structured) nature even within the separated process groups, an appropriated rank adjustment during the splitting procedure can lead to an improved communication performance. In order to clarify this, we take a second look at Figure 2 showing the two linked clusters, each built up in the form of a *ring* network. In the case of an application with a communication pattern that is characterized by a strictly *neighbor-to-neighbor* correlation, the placement of immediate neighbors on directly linked network nodes may be a good mapping scheme in order to avoid unnecessary network contention.

At this point it should be mentioned that the MPI standard offers this kind of mapping feature in terms of the *process topology mechanism* which follows the *top-down* approach (see [9], p.241ff). This means that the application programmer has to specify the respective communication pattern of an algorithm e.g. in terms of an unweighted graph (the so-called *virtual topology*) that then can be used by the MPI runtime system to find a good rank mapping onto the physical hardware topology. However, this generalized mechanism has neither been considered in many MPI applications nor has it been implemented beneficially in many common MPI implementations [14]. Actually, this mechanism is often just implemented in the most trivial way while simply *ignoring* the optional topology information.

4.1 Explicit Rank-Reordering

For this reason, we have also added an *explicit* rank-reordering feature to the MPIXternal library that can help to circumvent such a semantic gap between an algorithm's virtual topology and the topology of the underlying hardware. This is achieved by giving the user the ability to specify explicit rank-reordering schemes within the external configuration file according to the *bottom-up* manner. In doing so, whenever a new and externally defined communicator is created, an appropriate process-to-processor mapping can be enforced by the user for the resulting sub-group of processes. However, applying such an appropriate mapping scheme demands that the user is aware of both the physical as well as the virtual topology. While the actual hardware topology should usually be known by the user (or at least by the system's administrator), the logical communication pattern of a certain algorithm can easily be determined by applying profiling tools such as *MPE/Jumpshot*, *KOJAK* or the *Intel Trace Analyzer*™ (formerly known as *VAMPIR*) [1, 10, 11].

```
<comm name="MPI_COMM_OCEAN">
  <cluster name="foo"/>
  <reorder expr="(NAMERANK*2)%(COMMSIZE-1)"/>
</comm>

<comm name="MPI_COMM_ATMOS">
  <node name="bar:00" reorder="0"/>
  <node name="bar:01" reorder="1"/>
  <node name="bar:02" reorder="3"/>
  <node name="bar:03" reorder="2"/>
</comm>
```

Listing 3. Rank-Reordering Example

Listing 3 shows an example configuration for this kind of user-defined rank-reordering scheme, tailored to the sub-topologies of the two linked ring networks, as shown in Figure 2. Here, again, the hostnames can serve for considering the hardware topology, while the actual remapping is performed during the splitting procedure of the parent communicator (`MPI_COMM_WORLD`) into the two stated sub-communicators. As before, all nodes of cluster `bar` are stated explicitly within the configuration and also the new ranks are stated explicitly, whereas for cluster `foo`, the new ranks are determined by means of an *expression* to be evaluated by each respective process at runtime. As one can see, MPIXternal supports the substitution of variable names such as `NAMERANK` (representing the alphabetical order of each processor name within the new communicator) or `COMMSIZE` (representing the number of processes within the new communicator) for the evaluation of such expressions. Additionally, further variables are supported for addressing *world* and *parent rank*, as well as for *world* and *parent size*, respectively.

4.2 Stating Virtual Topologies

At this point one may argue that it is the responsibility of the MPI implementation to assure a proper process-to-processor mapping rather than being the user's liability. Nevertheless, even in the case of an MPI library being capable of applying a beneficial mapping scheme (based on the topology mechanism), the respective application also has to provide its virtual topology information to the implementation in order to gain an improved communication performance. Thus, in reality, it seems that usually at least one of these two counterparts (and oftentimes both) lacks the needed support for conducting a convenient rank-reordering.

```
<comm name="MPI_COMM_OCEAN">
  <cluster name="foo" />
  <topology type="virtual" reorder="true">
    <edge>
     <first rank="3"/>  <second rank="2"/>
    </edge>
    <edge>
     <first rank="2"/>  <second rank="1"/>
    </edge>
    <edge>
     <first rank="1"/>  <second rank="0"/>
    </edge>
  </topology>
</comm>

<comm name="MPI_COMM_ATMOS">
  <cluster name="bar" />
  <topology type="virtual" reorder="true">
    <edge>
     <first  expr=" COMMRANK " />
     <second expr="(COMMRANK-1)*(COMMRANK>0)"/>
    </edge>
  </topology>
</comm>
```

Listing 4. Stating Virtual Topologies

Therefore, besides applying explicit rank reordering schemes, MPIXternal also allows attaching virtual topologies, just similar to attributes, to all externally defined communicators (and even to `MPI_COMM_WORLD`). For this purpose, the user can state such topologies within the external configuration file in terms of unweighted graphs. In this way, the user can supply the runtime system with the needed virtual topology information even in the uncommon case where the respective MPI implementation supports topology-aware rank-reordering, whereas a given application does not utilize the MPI's corresponding topology mechanism.

Listing 4 shows an example configuration for such externally attached virtual topologies. As one can see, the edges of such topologies also can be described both *explicitly* (by stating *from*- and *to*-ranks) as well as *implicitly* (by stating appropriate *expressions*) within a configuration. In this example, each of the two stated topologies just describes an application's communication pattern which exhibits a strictly *neighbor-to-neighbor* characteristic. At this point it should be mentioned that a *first-to-second* edge does not imply a direction of the communication and that self-loops are allowed but ignored (see also [9], p.247).

At application level, the reordering mechanism is triggered during the creation procedure of each externally defined communicator (by an internal call of `MPI_Graph_map()`), provided that `reorder=true` is set within the configuration. However, since the ranks of an already existing communicator cannot be reordered afterwards, in the case of the invariably `MPI_COMM_WORLD`, rank reordering must be performed by creating and using a corresponding *clone* communicator (e.g. `MPI_COMM_REMAP`) instead.

4.3 Patching the Topology Mechanism

As already mentioned, stating a virtual topology can only be beneficial if the underlying MPI runtime system has the ability to utilize this information with the objective of determining an appropriate mapping scheme for the processes. Since many MPI implementations do not support such an automated mapping feature, we want to introduce our own approach for doing so in this subsection.

The general mapping problem is to find an embedding of a stated virtual topology into the physical topology of the underlying network so that communication costs become minimal. Although this problem is NP-complete, a number of good heuristics exist [8] that can help to yield nearly optimal results within a tolerable amount of time. However, selecting and applying a certain heuristic demands at least knowledge of the respective hardware topology in terms of a weighted (or even unweighted) graph. For this reason, an MPI runtime system would have to be notified about this graph, but common MPI implementations usually do not offer suitable mechanisms for doing so. Therefore, we have added such a facility to MPIXternal in order to give the user the ability also to describe the actual physical topology. Of course, just *describing* the hardware structures does not automatically lead to a good embedding scheme. Initially, an appropriate mapping algorithm has to solve the given mapping problem within the MPIXternal layer. However, at this point, we have to deal with a trade-off between the solution's quality and the solving time invested. The MPI standard does not specify how much time an implementation should spend on finding a good solution, since the application programmer and/or the user know(s) how much time could be beneficial in a given situation.

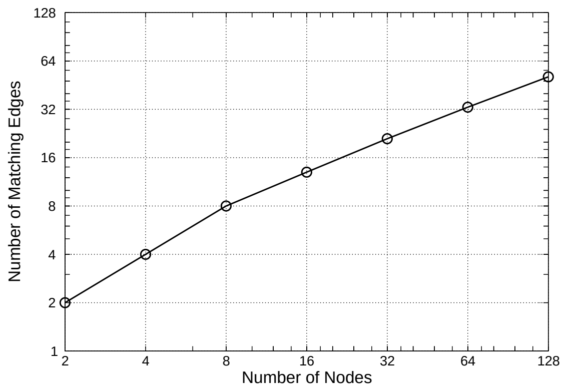

Figure 3. Mapping a Ring onto a Ring

```
<comm name="MPI_COMM_REMAP">
  . . .
<topology type="physical" timeout="1">
  <cluster name="foo"/>
  <cluster name="bar"/>
  <edge>
    <first  node="foo:03"/>
    <second node="bar:00"/>
  </edge>
  <edge>
    <first  node="foo:02"/>
    <second node="bar:01"/>
  </edge>
</topology>
```

Listing 5. Stating Physical Topologies

To keep it generic, we have decided not to use a certain heuristic but to base our mapping algorithm on a *time-bounded brute-force* method. This means that the user can specify how much time he or she is willing to spend on finding a good mapping scheme and that the best solution found by then will be applied afterwards. Therefore, our mapping algorithm follows a *random-based best-fit* strategy, which means that within the brute-force search for a good solution, mappings of processes onto nodes with an identical or at least similar number of edges are preferred. Additionally, this search is performed in parallel by all physical processors involved, each using a different random seed, in order to boost the search progress. Figure 3 shows exemplary results for a *ring-to-ring* mapping problem, determined within a timeout of *one second* by our brute-force algorithm. In this graph, the number of well-matching edges is plotted over the number of nodes used. As one can see, the algorithm can yield good mapping results especially for small but manageable numbers. However, for bigger and/or more complex systems, a heuristic tailored to the respective hardware topology could (and should) be provided by the respective user.

5. Some Selected Results

As already stated in the introduction, in case of heterogeneous environments, it becomes inevitable to adapt applications to the respective hardware structures. Especially in hierarchical systems, process grouping (e.g. by using MPI communicators) according to the respective hierarchy levels can be a practicable method for accommodating the system's nature. The performance impact of application optimization by applying such appropriate grouping schemes has already been studied in prior work [5, 3, 4, 2]. Therefore, for the scope of this paper, we will focus the evaluation here on the reordering techniques described in the former section.

For this evaluation, we have set up a synthetic heterogeneous environment, consisting of eight nodes which are linked together by a *little uncommon* topology, as shown in Figure 4. This topology can, for example, be understood as two clusters (each internally equipped with a high-speed cluster network) that are coupled by applying additional *gateway* connections.

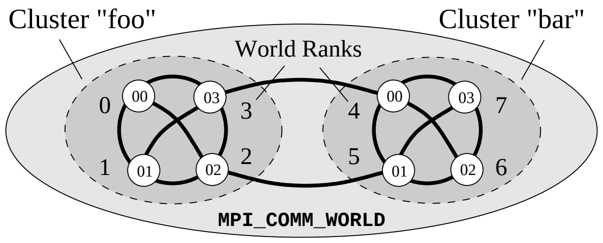

Figure 4. Topology Testing Environment

For building this testing environment, we have used a homogeneous cluster that features Gigabit-Ethernet as well as standard (Fast-) Ethernet. Therefore, we have configured the MPI runtime system to use Gigabit-Ethernet only between those pairs of processes which are directly linked by an edge within the hardware topology of Figure 5. Although all processes are still able to communicate to all the others via standard Ethernet, only pairs of processes on such directly connected nodes may benefit from communicating along the faster Gigabit-Ethernet connections.

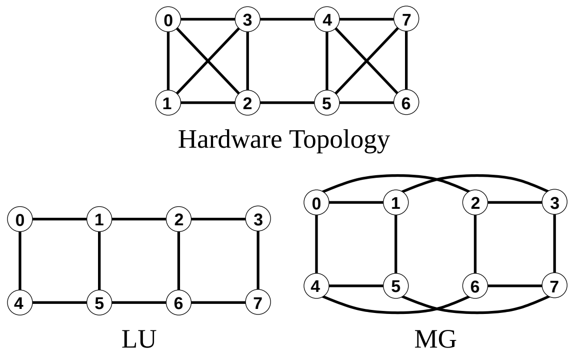

Figure 5. Physical and Virtual Topologies

The parallel application to be adjusted to this heterogeneous environment is emulated by two selected kernel algorithms of the *Numerical Aerodynamic Simulation* (NAS) benchmark suite [6], namely the LU (Lower/Upper triangular solver) and the MG (Multi-Grid-based 3D Poisson solver) benchmark. The communication patterns for these algorithms, as they can be determined e.g. via profiling, can be approximated for eight processes by the virtual topologies as shown in the lower part of Figure 5. As one can see, the pattern of the LU benchmark can obviously be embedded into the physical topology of our testing environment in an optimal way. Thus, when applying an appropriate rank-reordering, an improvement of the parallel performance of this algorithm can be assumed. In order to check this assumption, we have stated both the physical as well as the LU's virtual topology within an MPIXternal-related configuration file (Listing 5 shows an excerpt) and have just replaced all occurrences of `MPI_COMM_WORLD` by the externally defined `MPI_COMM_REMAP` in the respective source codes.

The charts in Figure 6 shows the measured performance results in terms of the gained speedups. For comparability, we have measured the runtime once for the direct mapped case (that means that no rank adjustment has been applied) and once for the case of reordered ranks, as they are determined by our mapping algorithm. Furthermore, we have also measured the runtime for the homogeneous case of a pure Gigabit-Ethernet cluster, and for a pure Fast-Ethernet cluster, respectively.

While the results for the LU benchmark confirm that the applied rank-reordering leads to a significant performance improvement (the gained speedup is here nearly identical to that of the homogeneous Gigbit case), remapping does not seem to be very worthwhile for the MG benchmark, at least in this test environment. The reason for this is the fact that the virtual topology of the MG algorithm is not perfectly embeddable into our simulated hardware topology. In fact, even when checking all possible mapping schemes, there will always be at least two communication-intense pairs of neighbors left that can no longer be mapped onto suitable connected nodes. As a result, the slow communication between these nodes dominates the overall parallel performance and this is why the rank-remapping is not very beneficial in this scenario.

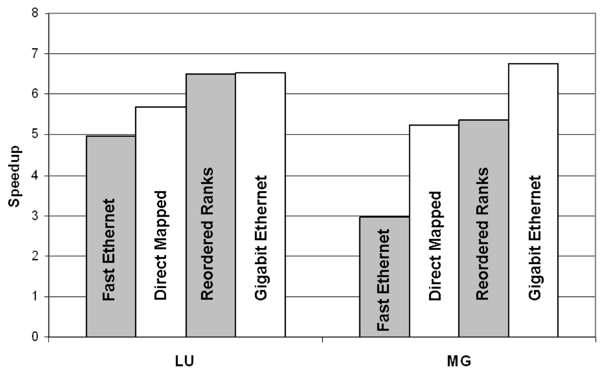

Figure 6. Measured Speedups for 8 Nodes

6. Conclusions

In this paper, we have presented our approach to accommodate the gap between today's hierarchical parallel computing systems and the flat non-hierarchical programming model of the current MPI standard. We have also shown that there are already existing facilities within the current standard (e.g. in terms of communicators) that have the ability to support a likewise hierarchical algorithm design. However, since the standard consciously hides heterogeneity, it is quite difficult for an application programmer to inquire into the needed information about the actual hardware structure in a convenient and portable way. As a result of this, we have introduced MPIXternal, our additional library, that allows the programmer to introduce hierarchy-awareness into the application, whereas the respective user can adjust this awareness to the actual target system later on. We have explained how this library can be utilized for providing optimization-related information according to both the *top-down* and the *bottom-up* approach.

Finally, it should be mentioned that the introduction of hierarchy-awareness has already been discussed during the standardization procedure of MPI-2.0 in terms of *cluster attributes* [13]. However, the current standard still lacks a portable optimization features for hierarchical systems. Nevertheless, the MPI-3 forum has fortunately announced that it will address this issue in the near future and, perhaps, our work may contribute some helpful suggestions to this discussion.

References

[1] A. Barak, S. Guday, and R. Laor. The MPE Toolkit for Supporting Distributed Applications. *Cuncurrency: Practice and Experience*, 4(6):459–480, September 1992.

[2] D. Becker, W. Frings, and F. Wolf. Performance Evaluation and Optimization of Metacomputing Applications. In *Proceedings of the 3rd Workshop on Communication in Cluster- and Grid-Systems (KiCC)*, RWTH Aachen University, Germany, December 2007.

[3] C. Clauss, S. Gsell, S. Lankes, and T. Bemmerl. A Fair Benchmark for Evaluating the Latent Potential of Heterogeneous Coupled Clusters. In *Proceedings of the 6th International Symposium on Parallel and Distributed Computing*, Hagenberg, Austria, July 2007. IEEE CS Press.

[4] C. Clauss, S. Gsell, S. Lankes, and T. Bemmerl. An Approach for Deploying Externally Defined MPI Communicators at Runtime. In *Proceedings of the 3rd Workshop on Communication in Cluster- and Grid-Systems (KiCC)*, RWTH Aachen University, Germany, December 2007.

[5] C. Clauss, M. Pöppe, and T. Bemmerl. Optimising MPI Applications for Heterogeneous Coupled Clusters. In *Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, Dresden, Germany, September 2004. IEEE CS Press.

[6] D. Bailey et al. The NAS Parallel Benchmarks. Technical Report RNR-91-002, NAS Systems Division, January 1991.

[7] A. Geist. MPI Must Evolve or Die. In *15th European PVM/MPI Users' Group Meeting*, volume 5205 of *LNCS*. Springer, September 2008.

[8] O. Krämer and H. Mühlenbein. Mapping Strategies in Message-based Multiprocessor Systems. *Parallel Computing*, 9(2):213–225, January 1989.

[9] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard – Version 2.1.* High-Perfomance Computing Center (HLRS), Stuttgart, Germany, September 2008.

[10] B. Mohr and F. Wolf. KOJAK: A Tool Set for Automatic Performance Analysis of Parallel Programs. In *Processing of the 9th International Parallel Processing Conference (Euro-Par'03)*, volume 2790 of *LNCS*, Klagenfurt, Austria, August 2003. Springer.

[11] W. E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80, January 1996.

[12] R. Rabenseifner. Some Aspects of Message-Passing on Future Hybrid Systems. In *15th European PVM/MPI Users' Group Meeting*, volume 5205 of *LNCS*. Springer, September 2008.

[13] The Message Passing Interface Forum. MPI-2 Journal of Development. Technical report, , 1997.

[14] J. L. Träff. Implementing the MPI Process Topology Mechanism. In *Proceedings of the IEEE ACM SC 2002 Conference*, Baltimore, USA, November 2002. IEEE CS Press.