An Approach for Deploying Externally Defined MPI Communicators at Runtime

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

Abstract

When writing parallel applications according to the MPI standard especially for hierarchical computing environments, the recognition of the underlying heterogeneous hardware structure at application level is not trivial at all. Although the MPI standard tries to support the application programmer with some process grouping and mapping facilities (notably the communicator concept and the topology mechanism), the actual hardware hierarchy is usually still kept opaque. In this paper, we present a generalized approach that allows the programmer to create suitable process groups according to the given topologies by externally defining MPI communicators in corresponding XML files. We further introduce a small external library for MPI implementations that is able to parse those XML files and can build the desired communicators at runtime. That way, the actual hardware hierarchy becomes visible also at application level.

1 Introduction

An important feature of the Message Passing Interface (MPI) [15] 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 initial started processes into sub-groups, each forming a new self-contained communication domain represented by such a communicator object [13].

This concept usually follows a *top-down* approach where the process groups are built according to the communication patterns required by the parallelized algorithm. In this way, hierarchical communication structures within the algorithms can easily be implemented on top of the MPI layer. However, since most MPI implementations normally assume homogeneous hardware environments, the processes are usually mapped onto the available processors in a transparent way so that an arbitrary (or at least an implementation-dependent) process-to-processor mapping is most likely to result. That means that the MPI runtime system usually does not draw any association between the logical communication patterns of the algorithm on the one side and the underlying physical hardware topology on the other side. Although the MPI standard offers such an association feature in terms of the MPI topology mechanism, its intended functionality is very rarely realized by generic MPI implementations [19, 18].

Heterogeneity-Aware MPI In contrast to ordinary MPI implementations, heterogeneity-aware MPI libraries often provide dedicated adaptation features which help the application programmer to adapt the algorithms' communication patterns to the respective heterogeneity of the physical communication topology. However, the implementation of such optimization features normally follows a *bottom-up* method where the topology information must be passed from the MPI runtime system to the application in a more or less unconventional way. At this point, two different ways can be followed: the one way is to aim to keep standard conformity, whereas the other way is to sacrifice source code compatibility e.g. for a more convenient handling. For instance, several heterogeneity-aware MPI libraries supply the programmer with additional predefined MPI communicators that try to reflect the underlying hardware hierarchy. However, if the symbol names of those additional communicators are already set at compile time of the MPI library, an application breaks with the standard when using them and hence makes its source code less portable.

Remainder of the Paper In this paper, we present a generalized approach that circumvents the communicator determination at compile time by providing the programmer with the ability to use self-predefined MPI communicators whose group composition is just determined at runtime. After a brief recapitulation of the communicator concept in Section 2, we present a small additional library for MPI implementations in Section 3 that follows our approach and which is capable of building the desired communica-

tors for the application. Although the approach introduced is mainly intended to build communicators following the *bottom-up* method for heterogeneous computing environments, it can also be applied for a *top-down* strategy where the communicators are built, guided by the programmer, according to the patterns of the respective algorithm. Therefore, application examples for both those methods are presented in Section 4. An overview of related work and possible future extensions to our work concludes the paper in Section 5.

2 The MPI Communicator Concept

In order to follow the novelty of the approach introduced in this paper, one needs some basic knowledge about the common way of dealing with MPI communicators. For that reason, we briefly resume the MPI communicator concept and its handling on application level in this section. For more detailed explanations, please refer to [10, 11, 13].

Intra-Communicators An MPI communicator is, as already mentioned in the introduction, an abstract object that represents a (sub-)group of parallel processes defining an explicit communication space. According to the MPI standard, there exist three predefined communicators in every MPI environment. Those communicators are:

- `MPI_COMM_WORLD` — specifying all started processes within an MPI run
- `MPI_COMM_SELF` — identifying each MPI process itself
- `MPI_COMM_NULL` — pseudo communicator representing invalid communicators

The first and, to a very minor degree, also the second of those can be used by an application programmer for (intra-group) communication as well as for deriving new communicator groups. For that purpose of creating new communicators, the standard defines, amongst others, the function `MPI_Comm_split()` that partitions the group of a parent communicator into disjoint subgroups. Each process of those subgroups in turn will be associated either with the respective communicator created or with `MPI_COMM_NULL` if the calling process does not take part in any of the defined subgroups. At this point it should be emphasized that the created child communicators within one call to the split function will get the same communicator names assigned. That in turn means that a symbol name representing a communicator object on application level does not necessarily represent identical groups for the different processes at runtime (see Figure 1).

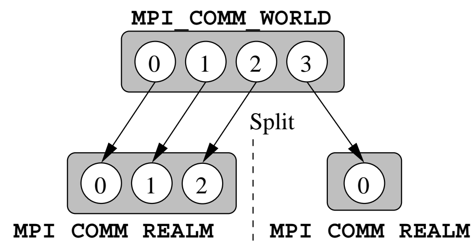

Figure 1. Splitting of a Communicator

Inter-Communicators Besides the regular communicators introduced in the last preceding paragraph, which are intended for *intra*-group communication, there also exists a further type of communicators, which is designated for *inter*-group communication. Therefore, communicators of this second type are formally called *inter-communicators*, whereas those of the prior type are formally called *intra-communicators*. The difference between both types is that an inter-communicator is associated with two groups of processes: a local and a remote group.

In the context of inter-communicators, the processes are always identified by their rank within the *remote* group. That way, messages sent or received via an inter-communicator are always exchanged between processes of the two disjoined groups. Therefore, the creation of a new inter-communicator is based on the inter-linkage of two disjoined intra-communicators by calling the `MPI_Intercomm_create()` function.

At this point it should be emphasized that the returned data type of an inter-communicator is the same as for a regular intra-communicator. Hence, in order to be able to distinguish between those communicator types represented by the same data type, the standard provides the `MPI_Comm_test_inter()` function.

Process Identification When defining new MPI communicators, the application programmer can build them either according to the *top-down* approach, which is the usual case, or according to the *bottom-up* approach for hardware awareness. In the first case, the world ranks of the processes can serve as a unique differentiator among them.

The second case, following the bottom-up approach, is a little bit more complicated: First of all, each process has to determine on which node of the hardware topology it is running. This identification can be done by utilizing the result of the `MPI_Get_processor_name()` function, whereas the programmer has to supply the correlation between processor names and the actual topology to be represented by the new communicators. By comparing its own name with a list of processor names assigned to corresponding communicator groups, each process can then determine if it becomes part of a new communicator or not.

3 Implementation Details

3.1 A Library for Communicator Creation

So far, an application programmer usually has to encode the process identification described in the former section directly into the respective application. In order to ease the creation of self-defined communicators on the one hand and to make the process identification more flexible and independent from the application code on the other hand, we have developed a small software library that relieves the programmer from dealing with the communicator creation procedure.

For that purpose, the library provides a special function that only expects a reference to an uninitialized MPI communicator object as well as the symbol name of that object. When called, the function will look up a table containing the descriptions of the new communicators indexed by the communicator name. If the desired communicator can be found in the table, each calling process will check the respective description in order to determine whether it becomes part of a new communicator group or not. Afterwards, the participating processes will build the new communicator while the others will just return a reference to `MPI_COMM_NULL`.

In order to supply the building function with the needed information stated in the lookup table, an additional initialization function must firstly read the desired communicator configurations from an appropriate XML file. The displacement 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 needs not necessarily be written by a user or an application programmer. In fact, the XML file containing the communicator definitions can rather be generated by a process scheduler, for instance, or even by the runtime environment of a heterogeneity-aware MPI implementation.

3.2 The XML Configuration Files

The reasons for XML [20] being the file-format of choice are that it is human readable, easy to understand and widely used. Another important reason was that it is highly hierarchical structured and thus represents the computer structure in use quite well. There exist some public domain XML parsers of which we chose `libxml2`[1] for our implementation.

The data is being structured by so-called *elements*. Usually an element consists of the wanted information surrounded by a dedicated start tag and the corresponding end tag. For example, in our implementation we use `<processor>igor</processor>` to specify a processor named `igor`. For the definition of a process group represented by a communicator, we have introduced the `<comm>` tag. This element may harbor `processor` elements as well as other `comm` elements, allowing a recursive definition style. To name a communicator, we use a corresponding attribute within the tag like `<comm name="MPI_COMM_RED">...</comm>`.

Identification by Processor Names In order to identify the nodes of the hardware topology according to the bottom-up approach, the communicators have to be associated with the respective processor names. Therefore, consider the following example of a short configuration file:

```
<comm name="MPI_COMM_RED">
    <processor>pd-01</processor>
    <processor>pd-02</processor>
    <comm name="MPI_COMM_PINK">
        <processor>pd-02</processor>
    </comm>
</comm>
<comm name="MPI_COMM_RED">
    <processor>pd-03</processor>
</comm>
<comm name="MPI_COMM_BLACK">
    <processor>pd-04</processor>
</comm>
```

Here, the first communicator, named `MPI_COMM_RED`, contains the processors `pd-01` and `pd-02`, whereby the latter is also in a sub-communicator named `MPI_COMM_PINK`. It is important to know that each sub-communicator may only consist of processors that are also defined in its parent communicator. Therefore it is for instance not possible in the above example configuration for the sub-communicator to include the processor `pd-03`, since it is not in the parent-communicator. As already stated in Section 2, the MPI standard allows for different communicator groups to having the same symbol name (as above with `MPI_COMM_RED`), which is also supported by our library.

An additional feature is that processor names may also include regular expressions, as for example `p[d4]-01`. This expression matches the processors `pd-01` and `p4-01` which can be quite useful for example if you have two clusters at hand that use the same naming scheme. (In fact, *PD* and *P4* are names of two actual cluster installations at our institute.)

Although the new communicator-internal processor ranks are typically derived from the order of occurrence in the XML file, they can also be stated explicitly via an additional `key` attribute. However, in case of a regular expression, the communicator-internal ranks are determined by the alphabetical order of the actual processor names.

[1] http://xmlsoft.org/

Inter-Communicators In analogy to Section 2, it is possible to define *inter*-communicators, too. In the XML file this can be accomplished in a way alike the following:

```
<intercomm name="MPI_COMM_INTER">
  <first color="1">MPI_COMM_RED</first>
  <second>MPI_COMM_BLACK</second>
</intercomm>
```

This code fragment defines an inter-communicator named `MPI_COMM_INTER` between the communicators `MPI_COMM_RED` and `MPI_COMM_BLACK`. Since inter-communicators may only connect two communicators that have the same parent, the `intercomm` element may only stand within a `comm` element (or in the top level node with `MPI_COMM_WORLD` being the common parent). In our implementation, different communicators with the same name are distinguished by their different `colors`, which again is just a value between zero and the number of the equally named communicators minus one. If the communicator name is unambiguous, the color statement can be omitted. That means in this example that the inter-communicator will represent an interlinking domain between the processors `pd-03` and `pd-04`.

3.3 Portable Integration into Applications

To us, it was very important that our approach chosen for deploying externally defined communicators ensures portability. Portability means that the library introduced is interoperable with any underlying MPI library on the one side, and that the respective applications can still be written in a standard conform manner on the other side. However, in order to utilize the new features introduced, an application has to be written in a *distinctive* way that will be described below. That way, the application can not only be compiled with and without the additional library but can also still be started in both cases while switching back to the standard communicator environment in the latter case of lacking support.

For that reason, we have chosen to place our library transparently between the application and the respective MPI implementation. Thus, the call to the new communicator creation function described in Section 3.1 becomes invisible to the application by hiding it inside faked MPI commands. Therefore, the application merely has to include an also faked `mpi.h` header instead of the corresponding header file of the native MPI library when being compiled.

Nevertheless, another possible way is to place the call to the new communicator creation function directly into the application code and using preprocessor directives for ensuring portability. However, in this paper we want to focus on the former alternative described.

Faked MPI Functions When using this option, the externally defined communicators are built at runtime during an appropriate call of `MPI_Comm_rank()` or `MPI_Comm_size()`, assuming that those are one of the initial MPI functions called which expect a communicator as one of the arguments. For that purpose, all occurrences of those function calls are replaced within the application via the preprocessor by the following directives and prototypes (for `MPI_Comm_size()` in an analogous manner):

```
#define \
 MPI_Comm_rank( a, b ) \
 MPIX_FAKE_Comm_rank( &a, b, #a )

int MPIX_FAKE_Comm_rank
  ( MPI_Comm *comm,
    int      *rank,
    char     *name );
```

That way, the library can get aware of the respective communicator name in case the function is called with the object's symbol name as an immediate argument.

That is for example:

```
MPI_Comm_rank(MPI_COMM_BLACK, &rank);
```

Whereupon the symbol name `MPI_COMM_BLACK` is passed as the third argument of the fake function into the string `name`. By searching in the previously memorized look up table, the library can now determine whether the given communicator is an externally defined one, and if so, how to create the desired entity. Furthermore, since the preprocessor also converts the former *call-by-value* style for the communicator argument into a *call-by-reference* one, a reference of the currently built communicator entity can now be returned back to application level. By this means, a second search for this communicator becomes unnecessary for further MPI function calls because the returned reference now actually represents a valid MPI communicator.

Usage at Application Level Nevertheless, since the library has to decide whether it is a first call or not, all communicators that are assumed to be defined externally have to be explicitly declared as `MPI_COMM_NULL` before being used (or rather, before being *built*). However, due to the fact that a call with a `NULL` communicator will most likely result in an abort of the running program in common MPI environments, an application has to take appropriate measures in order to be still consistent with the standard. For that purpose, an application should ensure that an MPI function also returns in case of an erroneous communicator argument. This is usually done by setting the MPI error handler to `MPI_ERRORS_RETURN`. That way, the application can determine on its own whether a communicator is valid or not.

In order to clarify the handling of those issues, refer to the following code example:

```
MPI_Comm MPI_COMM_RED
       = MPI_COMM_NULL;

MPI_Errhandler_set(MPI_COMM_WORLD,
                   MPI_ERRORS_RETURN);

if(MPI_Comm_rank(MPI_COMM_RED, &rank)
   == MPI_SUCCESS)
{
  /* I am part of MPI_COMM_RED! */
  . . .
}
else
{
  /* I am NOT in MPI_COMM_RED! */
  . . .
}
```

Initially, a variable of the data type `MPI_Comm` is declared for the new communicator `MPI_COMM_RED` and initialized with `MPI_COMM_NULL`. Since the error handler of the MPI environment gets instructed to return all occurring errors to the application level, the subsequent call to `MPI_Comm_rank()` with a `NULL` communicator does neither abort if the new communicator could not be found in the external communicator definition, nor in the case that the application was built without our library's support. In both cases, all calling processes will discover that they are not part of `MPI_COMM_RED`. However, in the other case of an adequately defined external communicator, the calling processes will build the new communicator according to its grouping definitions from the XML file right within the faked `MPI_Comm_rank()` function. At this point it should be emphasized that the creation of a new communicator is a collective operation within the parent group. That in turn means that even though they may become not part of the new communicator group, all processes within the parent group have to call the respective function.

For inter-communicators, all these descriptions can be applied in a similar manner with the exception that `MPI_Comm_test_inter()` is used as the communicator building fake function:

```
if(MPI_Comm_test_inter(MPI_COMM_INTER,
   &flag) == MPI_SUCCESS) && (flag))
{
  /* inter-communicator created! */
  . . .
}
```

And also in this case, the function must be called by *all* processes within the parent communicator because it serves as the so-called *bridge* communicator in MPI terms.

4 Application Examples

The examples presented here are derived from parallel algorithms that perform so-called nearest neighbor exchanges of row and column halos from a 2D array [4, 10, 21]. We have chosen this communication pattern because it is a common operation for domain decompositions applied in parallel simulation applications. Such a decomposition scheme is exemplarily shown in Figure 2, where 12 processors work on a 3×4 block partitioned domain. As one can see, the resulting communication pattern is quite structured since only directly neighboring pairs of processors are exchanging messages in a horizontal and vertical manner.

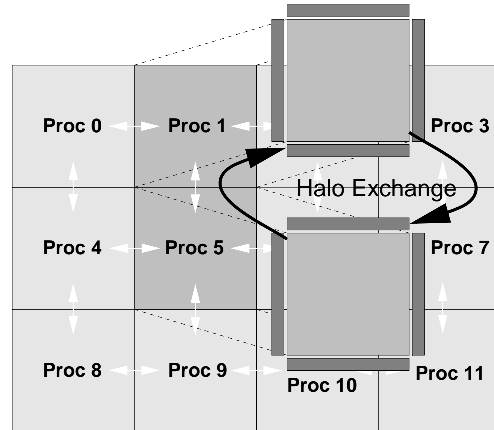

Figure 2. Domain Decomposition

A Top-Down Process Grouping According to this communication pattern, the processes can obviously be arranged into horizontal and vertical communicating subgroups as quoted below:

Group	Processes	Group	Processes
Horizontal 0	0, 1, 2, 3	Vertical 0	0, 4, 8
Horizontal 1	4, 5, 6, 7	Vertical 1	1, 5, 9
Horizontal 2	8, 9, 10, 11	Vertical 2	2, 6, 10
		Vertical 3	3, 7, 11

Of course, this simple grouping scheme can easily be implemented inside an application, that means without deploying externally defined communicators. Nevertheless, externally defined communicators can still be useful here in order to map the *virtual* topology (that is the algorithm's communication pattern) onto the underlying (homogeneous) hardware topology. If the underlying network is, for example, a Cartesian mesh, then an optimal *virtual to physical* topology mapping can be performed by placing the processes onto the appropriate processors as denoted in Figure 3. In this example, the processor names are composed of a tuple

that indicates the position (row and column) of a processor in the mesh network. Thus, by creating the above process groups according to this naming scheme , an ideal mapping can be accomplished.

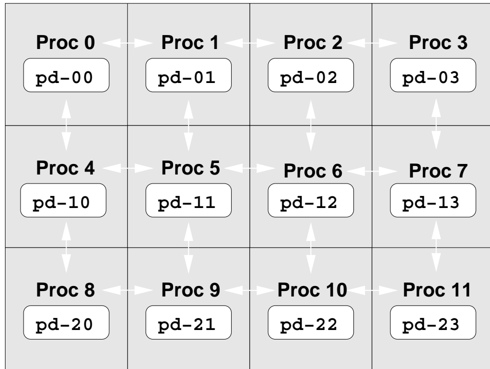

Figure 3. Process to Processor Mapping

When using externally defined communicators for that purpose, just the following XML entries have to be supplied with substituted x and y:

```
<comm name="MPI_COMM_HORIZONTAL_x">
  <processor>pd-[x][0-3]</processor>
</comm>
<comm name="MPI_COMM_VERTICAL_y">
  <processor>pd-[0-2][y]</processor>
</comm>
```

At this point it should be mentioned that the MPI topology mechanism is exactly what the standard defines to overcome this issue. In particular, the `MPI_Cart_create()` function serves as an easy way to create a new communicator with a Cartesian topology attached [10, 13]. Furthermore, an MPI implementation may reorder the processes within this function call for a better performance. Unfortunately, this reorder mechanism is only very rarely realized in a beneficial way in common MPI implementations [19, 18]. Since our library provides an explicit rank reordering determined on the basis of the processor names representing the actual hardware topology, its utilization can obviously be helpful if the respective MPI implementation does not offer appropriate mapping facilities on its own.

An Example following the Bottom-Up-Approach As already pointed out in the introduction, many heterogeneity-aware MPI implementations provide the application programmer with additional adaptation features that should support an appropriate process mapping onto the (mostly) hierarchical physical topology. However, the realizations of those features are usually not conforming to the standard and, moreover, depend on the MPI implementation used. That means that when adapting an application to a hierarchical topology by using the auxiliary features offered by a certain MPI implementation, the application becomes bound to this particular environment. In fact, this issue was the origin of the work presented here since we were looking for a portable way to specify hierarchical topologies in an MPI convenient manner.

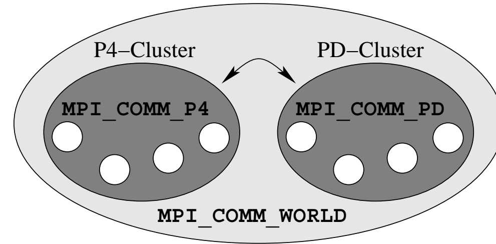

Figure 4. Two Coupled Clusters

Assume the following two-tier hierarchical system consisting of two coupled clusters in Figure 4. In such a coupled system, the interlinking network between the clusters obviously constituted the system's bottleneck, whereas the cluster internal connections are usually built up from dedicated high performance interconnects. Hence, in order to be able to forward messages along the inter-cluster link, while still be able to benefit from the fast internal cluster networks, an MPI implementation with multiple network support needs to be applied. There exist a couple of multi-network capable MPI implementations like Open MPI [9] or MPICH/Madeleine [1] and special *Grid-enabled* MPI libraries like MPICH-G2 [12], PACX-MPI [3] and GridMPI [14]. All of those libraries are proven to run large-scale applications and most of them offer an individual implementation of the above mentioned adaptation features.

However, at this point, an application programmer now has the opportunity to abandon the use of those intrinsic features by utilizing our approach of externally defined communicators that reflect the system's hierarchy in a portable way. Although in this case an additional communicator configuration file needs to be supplied, this can be either stated by a user who possesses the information about the hardware structure, or this file can be generated by an automated mechanism.

Currently, we have already implemented such a mechanism into the runtime environment of MetaMPICH [16], a Grid-enabled MPI library that has also been developed at our institute. MetaMPICH allows the user to configure the coupled system in a very detailed way via so-called *meta-configurations* that help to provide an explicit definition of each cluster involved. By extending such a meta-configuration, it is now possible for the user to include an additional communicator assignment into the configuration in order to provide self-named MPI communicators repre-

senting the respective cluster sites. The following paragraph shows an exemplary section of such a meta-configuration that typically contains many more items than shown here, as for example the types of the internal networks and the information about the interlinking topology between the sites. For more information about MetaMPICH and the syntax of its meta-configurations, please refer to [5].

```
METAHOST p4_cluster
{
  NODES = p4-01,p4-02,p4-03,p4-04;
  INTRACOMM = "MPI_COMM_P4";
  . . .
}

METAHOST pd_cluster
{
  NODES = pd-01,pd-02,pd-03,pd-04;
  INTRACOMM = "MPI_COMM_PD";
  . . .
}

CONNECTIONS
PAIR p4_cluster pd_cluster
-  { INTERCOMM = "MPI_COMM_INTER" }
. . .
```

Although a meta-configuration is not coded in XML but in a proprietary syntax, a designated parser, which is part of the MetaMPICH runtime system, can read this configuration and is able to setup the needed XML file containing the desired communicator definitions. For that purpose, the name and the path to the XML file are passed via environment variables to our library, whereas the runtime system of MetaMPICH has to assure the accessibility of the generated XML file on all relevant nodes. For the presented example, the resulting XML file would look like the following:

```
<comm name="MPI_COMM_P4">
    <processor>p4-01</processor>
    <processor>p4-02</processor>
    <processor>p4-03</processor>
    <processor>p4-04</processor>
</comm>
<comm name="MPI_COMM_PD">
    <processor>pd-01</processor>
    <processor>pd-02</processor>
    <processor>pd-03</processor>
    <processor>pd-04</processor>
</comm>
<intercomm name="MPI_COMM_INTER">
    <first>MPI_COMM_P4</first>
    <second>MPI_COMM_PD</second>
</intercomm>
```

The user can, of course, choose arbitrary communicator names representing the cluster sites. That way, it is possible to adapt an application e.g. for a hierarchical system consisting of two or more coupled sites without being bound to any actual system. Moreover, since the creation of the XML file may also be delegated to another instance than MetaMPICH, as for example to a topology analyzing tool, the application also becomes independent of the runtime environment in use.

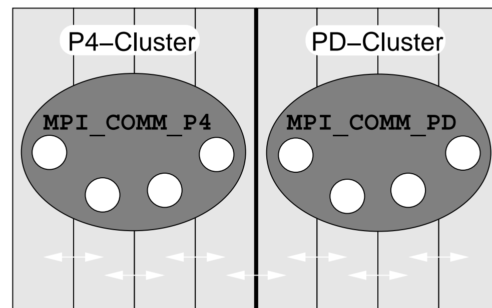

Figure 5. Application on Coupled Clusters

As a result, also the recently introduced application example can easily be adapted to a hierarchical system as denoted in Figure 5. In this exemplary case, the domain is partitioned into strips (columns) so that only two processors have to communicate across the inter-cluster link. For that purpose, the processes can now be grouped by the corresponding *intra*-communicators representing their respective cluster sites, whereas an additional *inter*-communicator can serve to handle the communication between the two clusters. That way, an adaptation of the application to the heterogeneous system can be achieved by applying a *partially synchronous* relaxation scheme between the sites (in this case, the inter-cluster halo exchanges are just performed in a periodic manner), while sill being fully synchronous within the clusters. By this means, the inter-cluster communication bottleneck can be compensated by employing accessory computing power in terms of additional iteration steps. For more details about this adaptation and optimization approach, please refer to [21, 2, 6].

5 Conclusions, Outlook and Related Work

In this paper, we have presented our approach to simplifying the communicator creation for an MPI application programmer without loosing the freedom of choosing an arbitrary underlying MPI library on the one hand, and, moreover, without breaking the applications' source code portability and standard conformity on the other hand. By using XML, it is also possible that not the programmer himself needs to write the communicator configuration file, but, given an appropriate plug-in, this can also be automatically done, for example, by a process scheduler, by a topology

analyzing tool or even by the MPI runtime environment itself. Further application areas may be the automated and standardized communicator definition by load balancers or by domain decomposition tools that are able to provide simulation applications with appropriate process grouping schemes for a given problem to be solved on a certain system.

Currently, we plan to develop a plug-in for the MP-Cluma cluster management tool [17] that should allow the user to compose the desired communicators in a very convenient way. MP-Cluma has also been developed at our institute in order to enable a uniform and comfortable startup of MPI applications on heterogeneous systems. Since MP-Cluma offers a Java-based graphical frontend to the user, we want to include an intuitive drag-and-drop facility for an easy grouping of processes. And, furthermore, since MP-Cluma needs to collect information about the respective hardware environment, the inclusion of an additional topology analyzer seems obvious.

There also exists some related work within this scope of GUI-based handling of MPI artefacts like communicators and MPI-related data types: VisualMPI [8] and BladeRunner [7] are tools that help the user to program MPI applications by representing those data types in a visual and abstract way. However, both projects focus rather on a semi-automated code generation at development time of an MPI application than on the mapping of communication patterns at runtime, as we do.

References

[1] O. Aumage and G. Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2003.

[2] H. E. Bal, A. Plaat, M. G. Bakker, P. Dozy, and R. F. H. Hofman. Optimizing Parallel Applications for Wide-Area Clusters. In *Proceedings of the IPPS/SPDP Workshops on Parallel and Distributed Processing*, Orlando, Florida, April 1998.

[3] T. Beisel, E. Gabriel, M. Resch, and R. Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proceedings of the 5th European PVM/MPI Users' Group Meeting*, September 1998.

[4] D. P. Bertsekas and J. N. Tsitsikilis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Englewood Cliffs, N.J., 1989.

[5] Chair for Operating Systems, RWTH-Aachen, University. *MP-MPICH – User Documentation & Technical Notes*.

[6] 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 (ISPDC 2007)*, Hagenberg, Austria, July 2007.

[7] B. R. T. Donald P. Pazel. Intentional MPI Programming in a Visual Development Environment. In *Proceedings of the 2006 ACM symposium on Software visualization SoftVis '06*. ACM Press, September 2006.

[8] D. Ferenc, J. Nabrzyski, M. Stroinski, and P. Wierzejewski. VisualMPI - A Knowledge-Based System for Writing Efficient MPI Applications. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting*, volume 1697 of *Lecture Notes in Computer Science*, Barcelona, Spain, September 1999.

[9] R. L. Graham, G. M. Shipman, B. W. Barrett, R. H. Castain, G. Bosilca, and A. Lumsdaine. Open MPI: A High-Performance, Heterogeneous MPI. In *Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[10] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI - second edition*. Scientific and Engineering Computation series. MIT Press, Cambridge, 1999.

[11] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. Scientific and Engineering Computation series. MIT Press, Cambridge, 1999.

[12] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5), 2003.

[13] M. Snir, W. Otto, S. Huss-Lederman, D.W. Walker and J. Dongarra. *MPI: The Complete Reference*. Scientific and Engineering Computation Series. MIT Press, Cambridge, 1996.

[14] M. Matsuda, Y. Ishikawa, Y. Kaneo, and M. Edamoto. Overview of the GridMPI Version 1.0. In *Proceedings of the SWoPP05, Japan*, 2005.

[15] MPI Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputing Applications*, 1994.

[16] M. Pöppe, S. Schuch, and T. Bemmerl. A Message Passing Interface Library for Inhomogeneous Coupled Clusters. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2003)*, Nice, France, April 2003.

[17] S. Schuch and M. Pöppe. MP-Cluma - A CORBA Based Cluster Management Tool. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2004)*, Las Vegas, USA, June 2004.

[18] R. Thakur and W. Gropp. Open Issues in MPI Implementation. In L. Choi, Y. Paek, and S. Cho, editors, *Advances in Computer Systems Architecture, 12th Asia-Pacific Conference, ACSAC 2007, Seoul, Korea, August 23-25, 2007, Proceedings*, volume 4697 of *Lecture Notes in Computer Science*, pages 327–338. Springer, 2007.

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

[20] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). *http://www.w3.org/TR/xml/*, September 2006.

[21] B. Wilkinson and M. Allen. *Parallel Programming - Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 2nd edition, 2005.