MPI Pre-Processor: Generating MPI Derived Datatypes from C Datatypes Automatically

Éric Renault and Christian Parrot
GET / INT
Département Informatique
9, rue Charles Fourier
91011 Évry, France
Tel: +33 1 60 76 45 56 — Fax: +33 1 60 76 47 80
{ eric.renault, christian.parrot } @int-evry.fr

Abstract

The grid usage is facing the problem that consists in running existing sequential code for parallel execution transparently (ie. using source code without modification). AIPE is a middleware that deals with this problem. MPIPP is the software component we have developed to allow the generation of MPI derived datatypes from C datatype definitions automatically. The goal of this new tool is to make the building of complex messages easier for the end-user. Moreover, this paper shows that MPIPP goes farther in the complexity level of C datatypes that can be taken into account than any other similar tools have ever gone to.

1 Introduction

Radical changes in the way of taking up parallel computing have operated during the past years. For example, cluster computing [1] allows organizations to use a high computing power for a small investment (regarding supercomputers). Grid computing [4, 5] goes one step further providing a theoretically unlimited computing power while just connecting to the Internet. If more and more friendly tools are available to manage and develop on clusters, a less important effort have been done to help grid computing. However, in order to increase the number of grid users, it is necessary to make the usage of the grid easier. The AIPE project aims at providing users a middleware infrastructure to develop and process divisible load applications in a very simple way. This paper presents MPIPP, a tool making the generation of MPI derived datatypes from C datatypes automatic.

The document is organized as follows. The next section presents the AIPE project that motivated this work. Section 3 describes how the automatic translation of C datatypes to MPI derived datatypes have been made possible and how the piece of software we have developed have been introduced inside the usual compilation chain. The next section is a discussion around two tricky cases regarding the automatic translation: pointers and unions. Before the conclusion, the last section compares our solution to related works.

2 Motivation

This work has been required to overcome a difficulty encountered in the scope of the AIPE project (Automatic Integration for Parallel Execution) [10]. Thus, this section shortly reminds the goals of the AIPE project.

Let's consider a data stream supposed to be sequentially processed by a program that generates a result stream. This processing is assumed to be a bottleneck of a pipeline. This processing can be characterized by means of function `Work ( TypeIn data, TypeOut result )` in which enter data of type `TypeIn` and from which exit results of type `TypeOut`. Each piece of data arriving by the ingoing stream is processed by this function, the one after the other one. The result of each sequential execution is then put into the outgoing stream. From now on, it is supposed that the processing of a piece of data is independent from the processing of another.

The ultimate target of the AIPE project is to make the use of grids easier for the end-user. In particular, AIPE allows the user to benefit from parallel execution on a grid without any notworthy modification of the source code of the `Work()` function. To achieve this, AIPE builds executables which are able to run the sequential code of function `Work()` on several nodes of a grid for different data simultaneously. Thus the user must only provide: the compiled

code of the `Work()` function; both `TypeIn` and `TypeOut` definitions. Note that the use of AIPE does not require to insert anything in the code of function `Work()`, like some other tools do [2]. Providing these pieces of information is sufficient to produce the executable codes. To run the executable code, it remains to choose the grid nodes. The grain size for AIPE parallel processing is the one of the `Work()` function.

AIPE is based on the Master-Slave paradigm. According to scheduling considerations, the Master sends to Slaves the data taken from the ingoing data stream, to be processed by function `Work()`. Results produced by Slaves are received by the Master before being inserted into the outgoing results stream. Preference has been given to the Master-Slave paradigm for sake of both simplicity and robustness on several sides of the problem: security, communications, scheduling... The intrinsic data parallelism of the application prevents from communications between Slaves and thus provides a very sample exchange schema between the Master and each Slave.

To desynchronize, as much as possible, the reading of data from the ingoing stream from the repartition of the work by the Master, AIPE manages two processes: the one slices the ingoing stream into pieces of data of `TypeIn` type before storing each of them in a buffer (slots of shared memory); the second one read this buffer content slot by slot... The set up of this bufferization mechanism for the outgoing stream as well as the ingoing one relaxes synchronization constraints between both acquisition and treatment and weakens the flow fluctuations of the streams. In the same way, bufferization of data and results can be introduced in Slave programs running for receiving from and sending to the Master respectivily.

AIPE only requires grid management standard functions. The Globus ToolKit [4] is used to provide authentication and staging executables when submitting a job. To prevent from making expensive input/output transfers, processes (the Master and the Slaves) communicate data and results straightly from memory to memory (ie. without using extra-files for storing or transferring data from one process to another one). Hence a Globus compatible MPI [3] implementation (MPICH-G2 [8]) has been used to ensure the message passing communication requirements.

Due to the absolute necessity for the processings to be independent, AIPE is well suited for applications involving data parallelism. So, as one would expect, by exploitating this data parallelism AIPE allows to obtain good performance measurements (near linear speed-up).

To make AIPE as generic as possible types of messages sent from the Master to the Slaves (data) and from the Slaves to the Master (results) must be taken into account without restrictions and without extra effort from the end-user. Therefore, according to both `TypeIn` and `TypeOut` (parameters of funtion `Work()`) declarations, messages type definitions should be automatically generated. MPICH-G2 API offers functions enabling dynamic message type declaration in the process of execution. Taking into account `TypeIn` and `TypeOut` type definitions by these primitives remains to be done.

3 Description

Fig. 1 presents a generic abstraction for C datatypes. There are three ways to define a new type using the C programming language. The first one consists in creating a kind of alias to an already existing type (either a basic type or a user-defined type) using keyword `typedef`; the second one consists in aggregating pieces of information inside a single structure or union using keywords `struct` and `union` respectively; the last one is a combination of both previous ways in which a structure or a union is defined together with a synonym. In case of a structure or union is defined, a list of fields is provided, each one being associated a type (basic or user-defined) and a list of variables. A variable is associated a name, the level of indirections (the number of pointers to cross to reach the information of the given type) and a list of array sizes, one for each dimension.

As one can see, keyword `enum` is not taken into account directly as it just aims at defining an integer for which the set of values is specifyed at compilation.

MPI provides six functions to define the structure of new derived datatypes, five functions to get information about datatypes (basic or derived) and two functions for the creating and the destruction of new derived datatypes. However, only four are required to generate MPI derived datatypes from C datatypes automatically. Others are useful to create specific datatypes to transfert parts of a matrix or noncontiguous data from a given structure.

For MPIPP, the definition of MPI derived datatypes have been divided in two cases. The first case is the creation of a new MPI derived datatypes that consists in an array. In this case, the type is translated using the `MPI_TYPE_CONTIGUOUS` function for which only the number of elements and the size of each element in the array must be specified. If an array is composed of more than one dimension, the number of elements for the MPI derived datatype is the product of the size of each dimension. If the new MPI derived datatype is based on a basic or an already derived datatype, the new one is defined as an array composed of one dimension with a single element. The second case is the creation of a new MPI derived datatype that consists in a structure. In this case, the type is translated using the `MPI_TYPE_STRUCT` function for which only the offset, the size and the type of each element must be specified. For more complex datatypes, ie. those involving at the same time arrays and structures in any order and at any depth, a recursive definition have been implemented using the two cases described above. Note that

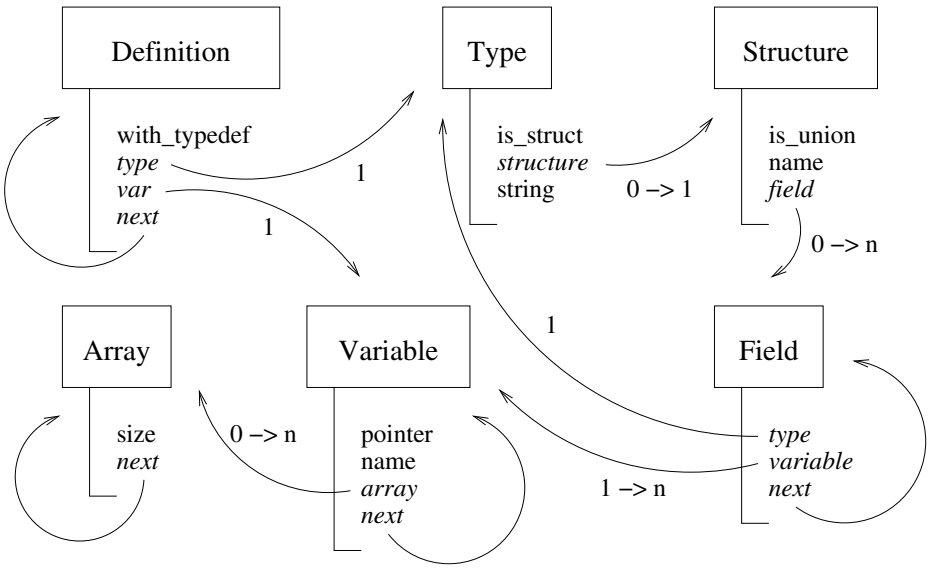

Figure 1. Data structures to store the definition of C datatypes.

the other functions provided by MPI to define complex derived datatypes are not useful here as the C programming language does not allow to request a specific *extent* and/or *stride*.

The list of C datatypes from the original file, to be transformed to MPI derived datatypes, is provided on the command line when invoking `mpipp`. This list can be hard coded to match the requirements of a specific software suite (this is what have been done for AIPE) or it can be the result of any script.

In order to create a new MPI derived datatype, one must use MPI functions. It is not possible to define this new type using a set of specific directives. Thus, the definition of the new MPI derived datatype is provided as a function which prototype is as follows (where XXX is the name of the C datatype to transform):

```
int MPI_Typedef_XXX ( MPI_Datatype * )
```

Then, it is the responsability of the developer to call this function in order to make this new MPI derived datatype available to its application.

The automatic generation by MPIPP of MPI derived datatypes from C datatypes have been made possible by introducing extra steps in the usual compilation chain as shown in Fig. 2. Typically, in order to make sure the definition of a given type is complete, the first step in the compilation chain (`cpp` for the C preprocessor) is executed. As the generated file is to be compiled by the effective C compiler (`cc`), it must contain all the information required by any type in the file (if not, as any type must be defined before being used, this file could not be compiled properly by `cc`). Thus, instead of feeding `cc` with the intermediate file generated by `cpp`, this intermediate file is transformed by `mpipp` (the MPI preprocessor we have developed). Both `lex` and `yacc` tools [9] have been used to parse the intermediate file generated by `cpp` and create the data structures presented in Fig. 1. Then, `mpipp` uses these data structures to produce a new C file in which the definition (according to the three cases above) of the new MPI derived datatypes have been appended to the original file automatically. This new source file is then provided to `cpp`. The intermediate file which results is then processed by the usual compilation chain.

4 Example

In order to illustrate the functionalities of MPIPP, the simplest way consists in providing an example and have a look at the code automatically generated by the software.

The command we have developed (`mpipp`) to perform the automatic translation of C datatypes to MPI derived datatypes requires the source code to be provided on the standard input and the result of the processing is then provided by the command to its standard output. Parameters provided on the command line is the list of C datatypes that must be processed by the command.

If `src` is a file that contains the definition of C datatypes and `dest` is the file in which the user wants to store the result of the processing from `mpipp`, the user can provide a command line close to the following one in order to generate the MPI derived datatype associated to C datatype `rectangle` automatically:

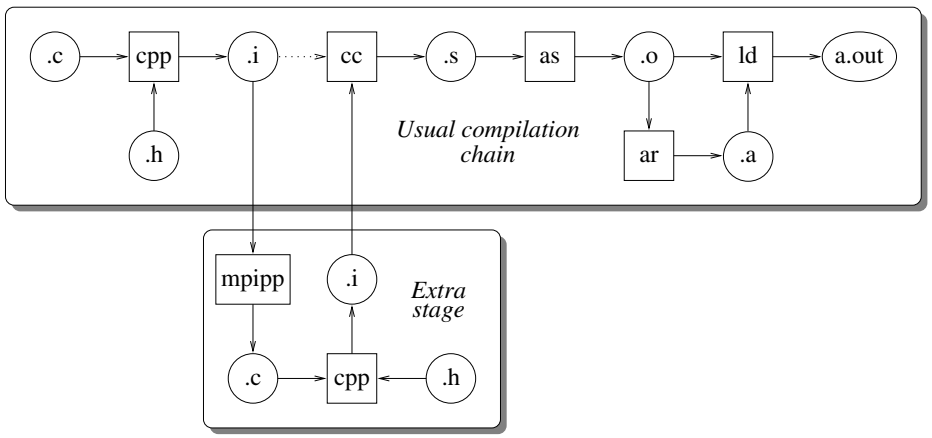

Figure 2. Introduction of MPIPP in the compilation chain.

```
cat src | mpipp rectangle > dest
```

Fig. 3 presents a part of the content of a file (file `src` on the command line above) including typical examples of structured datatype declarations written using the C programming language. In this example, two C datatypes are defined. The first one is a simple declaration for a string of characters; and the second one is a composed declaration using both predefined and user-defined C datatypes.

```
typedef char string [ 256 ] ;

typedef struct {
    int id ;
    string name ;
    struct {
        int length ;
        int width ;
    } size ;
} rectangle [ 100 ] ;
```

Figure 3. Input to `mpipp`.

Listings #1 to #5 present the code added by `mpipp` to the content of file `src` automatically, ie. the content of file `dest` when using the command line above.

Function `MPI_Typedef_rectangle` is composed of two parts: the first one (lines 5 to 25) declares both datatypes and variables used in the function and the second one (lines 27 to 66) provides the set of MPI calls for the definition of the new MPI derived datatype. Both parts are following the same scheme.

When the source code is parsed by `mpipp`, the piece of software creates a list of definitions which structure is presented in Fig. 1. Then, in order to define the new MPI derived datatype, a depth-first analysis is performed.

For the new definition of a flat datatype (ie. a datatype which is not a structure and which do not use a structure for its definition), the process is recursively iterated on the type on which the new datatype is based if the type is not a base type. Then, wether the original type is a flat datatype or not, the type is created using the pair of `MPI_Type_contiguous` and `MPI_Type_commit` functions (lines 27 and 28 for type `string`).

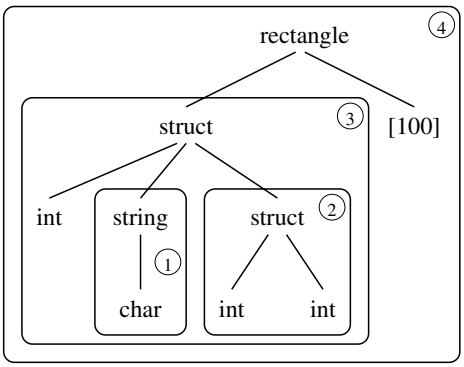

Figure 4. Depth-first analysis of the rectangle C datatype.

This example also highlights that whatever the way the structures are defined in the source code, the definitions of structures in the `MPI_Typedef_XXX` function is always limited to one level, exploding the definition in several parts to match this model (lines 7 to 10) in Listing #2. This way, it becomes possible to name each new datatype and therefore it is easier to get all its characteristics (especially size and offset). Moreover, this has no impact on the memory representation of data structures.

For the new definition of a non-flat datatype (ie. a type including structures either defined inline in the structure

or which type is defined outside the structure), the process is recursively iterated on the type associated to each field in the new datatype until the associated type is a base type. In our example, and using the depth-first analysis on type `rectangle` as shown in Fig. 4, it has been divided as follows: first, `string` is defined

```
 5 typedef char string[256];
 6 MPI_Datatype string_datatype;
   .
   .
   .
   // Array definition
27 MPI_Type_contiguous(256,MPI_CHAR,&string_
   datatype);
28 MPI_Type_commit(&string_datatype);
```

Listing #1

then, the inner most internal structure is defined

```
 7 struct name_2{
 8     int length;
 9     int width;
10 } struct_2_var[1];
11 MPI_Datatypestruct_2_datatype ;
12 MPI_Aintstruct_2_disp[3];
13 MPI_Datatype struct_2_datalist[3];
14 int struct_2_len[3],struct_2_base;
   .
   .
   .
   // Begin of structure
29 MPI_Address(struct_2_var,&struct_2_base);

   // Field ``length''
30 struct_2_datalist[0]=MPI_INT;
31 struct_2_len[0]=1;
32 MPI_Address(&(struct_2_var[0].length),str
   uct_2_disp+0);
33 struct_2_disp[0]-=struct_2_base;

   // Field ``width''
34 struct_2_datalist[1]=MPI_INT;
35 struct_2_len[1]=1;
36 MPI_Address(&(struct_2_var[0].width),str
   uct_2_disp+1);
37 struct_2_disp[1]-=struct_2_base;

   // End of structure
38 struct_2_datalist[2]=MPI_UB;
39 struct_2_len[2]=1;
40 MPI_Address(struct_2_var+1,struct_2_disp+
   2);
41 struct_2_disp[2]-=struct_2_base;

   // Structure definition
42 MPI_Type_struct(3,struct_2_len,struct_2_d
   isp,struct_2_datalist,&struct_2_datatype);
43 MPI_Type_commit(&struct_2_datatype);
```

Listing #2

this is followed by the definition of the external structure

```
15 struct name{
16     int id;
17     string name;
18     struct name_2 size;
19 } struct_var[1];
20 MPI_Datatype struct_datatype;
21 MPI_Aint struct_disp[4];
22 MPI_Datatype struct_datalist[4];
23 int struct_len[4],struct_base;
   .
   .
   .
   // Begin of structure
44 MPI_Address(struct_var,&struct_base);

   // Field ``id''
45 struct_datalist[0]=MPI_INT;
46 struct_len[0]=1;
47 MPI_Address(&(struct_var[0].id),struct_di
   sp+0);
48 struct_disp[0]-=struct_base;

   // Field ``name''
49 struct_datalist[1]=string_datatype;
50 struct_len[1]=1;
51 MPI_Address(&(struct_var[0].name),struct_
   disp+1);
52 struct_disp[1]-=struct_base;

   // Field ``size''
53 struct_datalist[2]=struct_2_datatype;
54 struct_len[2]=1;
55 MPI_Address(&(struct_var[0].size),struct_
   disp+2);
56 struct_disp[2]-=struct_base;

   // End of structure
57 struct_datalist[3]=MPI_UB;
58 struct_len[3]=1;
59 MPI_Address(struct_var+1,struct_disp+3);
60 struct_disp[3]-=struct_base;

   // Structure definition
61 MPI_Type_struct(4,struct_len,struct_disp,
   struct_datalist,&struct_datatype);
62 MPI_Type_commit(&struct_datatype);
```

Listing #3

the array finishes the definition.

```
24 typedef struct name rectangle[100];
25 MPI_Datatype rectangle_datatype;
   .
   .
   .
   // Array definition
63 MPI_Type_contiguous(100,struct_datatype,&
   rectangle_datatype);
64 MPI_Type_commit(&rectangle_datatype);
```

Listing #4

Note that lines 65 and 66 have been added to ease the development and involves no performance penalty at execution when using the new MPI derived datatype. Listing #5 presents the definition of function `MPI_Typedef_rectangle`.

```
 1 #include<mpi.h>
 2
 3 void MPI_Typedef_rectangle(MPI_Datatype *t
   ype)
 4 {
        .
        .
        .
65      MPI_Type_contiguous(1,rectangle_dataty
   pe,type);
66      MPI_Type_commit(type);
67 }
```

Listing #5

Finally, note that even if new C datatypes are already defined in the source code, they are all redefined inside the body of the function created to define the MPI derived datatype. This is performed in order to make sure there is no conflict between variable names used in the body of the function and other global variables or user defined datatypes. However, this has no impact on performance, except (may be) at compilation time.

5 Tricky cases

At present, only two cases have not been solved. However, it seems that solutions are very application dependent and generating MPI derived datatypes including these cases cannot be performed without extra information.

The first case is the use of pointers. What does it mean to send a pointer from one process to another one? In the MPI programming model, each process is associated a specific virtual address space; thus, the address of an object inside this virtual address space should have no meaning in the virtual address space of the other processes. One could argue this address should be sent as an `MPI_INT` or a set of `MPI_BYTE`; another one could argue that the user may expect the value pointed to to be sent: this seems to be application dependent and no formal choice have been done already.

The second case is the use of unions. Regardless structures, MPI provides no specific function to deal with unions. A solution is provided in [3] but an extra integer is required to select in an array the good MPI derived datatype. Therefore, as this integer must be managed by the user, it is not possible to generate an MPI derived datatype from a C datatype automatically.

Thus, as shown in this section, these tricky cases are not due to the implementation of MPIPP but are involved by the intrinsic nature of both pointers and unions.

6 Related Works

Similar tools have been designed in order to provide the same kind of translation. However, none was able to match the requirements of AIPE. This section presents both AutoMap and C++2MPI and discusses advantages and drawbacks of each both in general and regarding AIPE requirements.

AutoMap [6] is a high level tool that facilitate the use of data-structures in MPI. This piece of software generates new MPI derived datatypes from a user-defined file provided by the user. This file is obtained by modifying another file containing the definition of the C data-structure. Modifications consist in inserting for each instance a pair of begin/end comments to embrace the C data structure the user wants to translate to the corresponding MPI derived datatype. These C comments are used by the dedicated parser to automatically generate the C code that enables the dynamic declaration of the MPI derived datatype. The insertion of these C comments is handmade and needs the end-user to add extra lines in the source code. Therefore, this solution is not suitable for AIPE. Moreover, considered structures are built from basic C datatypes, not from complex C datatypes like arrays, structures... In other words, this software tool does not work for any C datatype definition.

C++2MPI [7] is a software tool developed as part of PGMT (for Processing Graph Method Tool) to provide the MPI derived datatype associated to a C++ class. The end-user specifies classes to deal with by means of `pragma` directives inserted before the definition of the corresponding classes. These directives are used by the dedicated parser to automatically generate the MPI code that enables the dynamic declaration of the MPI derived datatypes. Following the example of AutoMap, the insertion of these directives is handmade, ie. extra lines must be added to the source code by the end-user. Thus, this solution does not match AIPE requirements either. It is interesting to note that regardless AutoMap, C++2MPI is able to handle arrays of classes; however, it is not able to handle nested classes.

Table 1 draws a comparison between AutoMap, C++2MPI and MPIPP. Finally, the `mpipp` software tool we have developed provides two main advantages regarding the other works. First, it does not require the original source code to be modified by the end-user. Second, it allows to take into account any kind of C datatype (including nests of arrays and structures) except the tricky cases discussed in Section 5.

Table 1. Comparison of functionalities.

	AutoMap	C++2MPI	MPIPP
□ Language	C	C, C++	C
□ *Datatypes*			
△ basic	✓	✓	✓
△ *array*			
◊ one dim.	✓	✓	✓
◊ multi-dim.			✓
△ structure	✓	✓	✓
△ *depth of nested*			
◊ structures	1	1	any
◊ arrays	1	1	any
□ *Translation*			
△ number	1	any	any
△ source modified	yes	yes	no
△ notification	comments	`# pragma`	N/A

7 Conclusion

This article presents MPIPP, a new tool to automatically generate MPI derived datatypes from C datatypes. Existing works about this subject highlights that users and applications are requesting such a tool. Inside the AIPE framework, the MPIPP tool has proven its efficiency rapidly. As shown in section 6, the main improvement of MPIPP stands in its ability to translate any kind of C datatype to MPI derived datatype with no restriction but the use of unions and pointers (as discussed in section 5).

An MPI binding is available for other languages like Fortran, C++, Java... Each of them is potentially concerned with MPIPP advantages. Therefore, the implementation of MPIPP for these languages is worth to do. Moreover, this shall be easy regarding the fact that the current work deals with most of the datatypes available in all these languages. However, implementations for object-oriented languages may require extra developments regarding procedural languages.

References

[1] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*, volume 1. Prentice Hall, 1999.

[2] G. Cooperman, H. Casanova, J. Hayes, and T. Witzel. Using TOP-C and AMPIC to Port Large Parallel Applications to the Computational Grid. *Future Generation Computer Systems*, 19(4):587–596, May 2003.

[3] M. P. I. Forum. MPI: A Message Passing Interface Standard, June 1995.

[4] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[5] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Architecture*. Morgan Kaufmann Publishers Inc., 1999.

[6] D. S. Goujon, M. Michel, J. Peeters, and J. E. Devaney. AutoMap and AutoLink: Tools for Communicating Complex and Dynamic Data-Structures Using MPI. In D. K. Panda and C. B. Stunkel, editors, *Second International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications*, volume 1362 of *Lecture Notes in Computer Sciences*, pages 98–109, Las Vegas, NV, January 1998. Springer-Verlag.

[7] R. Hillson and M. Iglewski. C++2MPI: A Software Tool for Automatically Generating MPI Datatypes from C++ Classes. In *International Conference on Parallel Computing in Electrical Engineering*, pages 13–17, Trois-Rivières, QC, August 2000. IEEE Computer Society.

[8] N. T. Karonis, B. R. Toonen, and I. T. Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, May 2003.

[9] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. Unix Programming Tools. O'Reilly, 2nd edition, October 1992.

[10] D. Millot, C. Parrot, and Éric Renault. Transparent Usage of Grids for Data Parallelism. In M. Oudshoorn and S. Rajasekaran, editors, *Proceedings of the ISCA 18th International Conference on Parallel and Distributed Computing Systems*, pages 241–246, Las Vegas, NV, September 2005. ISCA, The International Society for Computers and Their Applications.