AutoMap & AutoLink
Tools for Communicating Complex and Dynamic Data-Structures using MPI

Delphine Stéphanie GOUJON[1,4], Martial MICHEL[2,4], Jasper PEETERS[3], and Judith Ellen DEVANEY[4]

[1] Télécom INT, France
[2] RÉSÉDAS, France
[3] University of Twente, Netherland
[4] NIST, USA
National Institute of Standards and Technology
Web page : http://www.itl.nist.gov/div895/sasg/
AutoMap & AutoLink Project Leader : Judith Ellen DEVANEY
AutoMap & AutoLink Project Contact : martial.michel@nist.gov

Abstract. This article describes two software tools, AutoMap and AutoLink, that facilitate the use of data-structures in MPI. AutoMap is a program that parses a file of user-defined data-structures and generates new MPI types out of basic and previously defined MPI data-types. Our software tool automatically handles specialized error checking related to memory mapping. AutoLink is an MPI library that allows the transfer of complex, dynamically linked, and possibly heterogeneous structures through MPI. AutoLink uses files generated by AutoMap to automatically define the needed MPI data-types. We describe each of these tools, and give an example of their use. Finally we discuss the internals of AutoLink design, and focus on the performance rationale behind them.

Key Words
MPI — AutoMap — AutoLink — Data-Structure — Library

Introduction

Many applications, business and scientific, require intensive and complex processing of data along with computing power. Yet moving these applications to parallel computers has been hindered so far by programming difficulties. The Message Passing Interface standard (MPI) [1] makes development of message passing programs easier through its portability and interface with common high level languages.

However, the MPI standard functions support complex data-types only indirectly. Users may create new MPI data-types out of basic data-types for use with MPI functions, but the process is tedious. Hence it is desirable to have a high level tool that automates the creation of MPI data-types from user-defined data-structures. Likewise, it is useful to simplify the sending and receiving of dynamically linked structures, by having the details handled by a library.

With these tools, compute intensive, data-structure rich application domains are easier to manage in parallel programs. These application domains include speech recognition, data mining, genetic programming, and complex modeling.

The remainder of the paper is organized as follows. In section 1, we present AutoMap, and explain complex and dynamic data-types. We show how to create an MPI data-type from a user defined type, and how AutoMap does it automatically. In section 2, we explain how AutoLink enables sending and receiving dynamically linked data-structures. Performance considerations are discussed in section 3.

1 AutoMap : Generating New MPI Types Out of Data-Structures

1.1 Data-Types

The MPI library can only transfer types that it knows about. The C implementation of MPI knows about the basic types in C. There are basic data-types such as `int`, `char`, `long` and `double` for which an MPI type exists, like `MPI_INT`, `MPI_CHAR`, `MPI_LONG` and `MPI_DOUBLE`. MPI functions can transmit basic C data-types. The MPI standard permits the creation of user defined MPI data-types. But creating an MPI data-type is complicated.

Complex data-types (i.e. struct) can only be sent and received if they are described to the MPI library. AutoMap performs this service for the programmer.

Complex Data-Type. Users may want to use a composition of basic data-types. In C, such a composition may be created using the `struct` operand. A complex data-type is such a user-defined structure, as long as the user doesn't use pointers to other structures or components inside the structure.

So, an example of a complex data-structure may be :

```
struct {
  char      display[50];
  int       maxiter;
  double    xmin, ymin;
  double    xmax, ymax;
  int       width;
  int       height;
} cmdline;
```

Dynamic Data-Type. Dynamic data-types are an extension of complex data-types. Dynamic data-types can handle structures containing pointer fields. Examples of such data-structures [2] are *linked lists*, *trees*, and *graphs*.

It is possible to send such structures with MPI but the pointer memory references will be invalid on the receiving processor. So the transfer of such data-structures is an operation left entirely to the user, and resolved by the use of the library AutoLink.

1.2 An Example of Complex Data-Type Creation with MPI

Here is an example showing how to create a complex data-type using MPI. Understanding these steps will make the design and use of AutoMap clearer.

Initial C Structure. The C structure used here is the one described in section 1.1. It contains 50 chars, 3 integers (1 and then 2 more), and 4 doubles.

Creation of the MPI Data-Type. The process of creating an MPI data-type involves specifying the layout in memory of the data in the C structure [3]. It is done in six operations :

1. Set up an array defining the number of data of each kind that will be used (in the same order as the structure definition).

   ```
   int          blockcounts[4] = {50,1,4,2};
   ```

   Which corresponds to : 50 `char`, 1 `int`, 4 `double`, 2 `int`,
2. Set up an array that will contain the type specification for each element contained in the structure. There are four fields in the `struct`, thus :

   ```
   MPI_Datatype types[4];
   ```

   Even if there are only 3 different data-types, one has to follow the `struct` *type order, meaning char, int, double, int.*
   Set the data-type for each element of the data-type to be created :

   ```
   types[0] = MPI_CHAR;
   types[1] = MPI_INT;
   types[2] = MPI_DOUBLE;
   types[3] = MPI_INT;
   ```

3. Set up an internal displacement array containing the memory offset of each field in the `struct`,

   ```
   MPI_Aint     displs[4];
   ```

   Map onto the displacement array, the MPI data-type on the C structure (by linking it to the very first memory element).

   ```
   MPI_Address(&cmdline.display, &displs[0]);
   MPI_Address(&cmdline.maxiter, &displs[1]);
   MPI_Address(&cmdline.xmin,    &displs[2]);
   MPI_Address(&cmdline.width,   &displs[3]);
   ```

Adjust the displacement array so that the displacements are offsets from the beginning of the structure.

```
for (i=3; i>=0; i--)
     displs[i] -= displs[0];
```

4. Give a name to the MPI data-type.

```
MPI_Datatype cmdtype;
```

5. Build the new MPI type. Set the container of the MPI data-type .

```
MPI_Type_struct(4, blockcounts, displs, types,
                &cmdtype);
```

6. Validate the type existence to be used with MPI.

```
MPI_Type_commit( &cmdtype );
```

1.3 Developed Software

AutoMap is a source-to-source compiler that automatically translates C structures into MPI data-types. To be more specific, AutoMap works as a lexer and a parser to translate C data-structures into MPI data-structures.

AutoMap was initially designed for sending and receiving complex data-types. Then it was extended to dynamic data-structures [4].

Currently two versions of AutoMap are available : a stand-alone version which operates only on complex data-types and a version coupled with AutoLink which is designed for dynamically linked data-structures.

AutoMap Process. The AutoMap compiler implements a grammar, which reads C structures into an input file and outputs the corresponding MPI data-types. The compiler generator used here is Yacc++ [5].The parsing of the input file is done by generating an Abstract Syntax Tree (AST) that is used in the type recognition process.

In the case of the stand alone version, AutoMap generates only the file containing the MPI data-types from the C structures `mpitypes.c`. When used with AutoLink, two other files are generated:

- `logbook.txt` is a log file,
- `al_routines.c` contains all the functions used internally by AutoLink.

MPI Data-Type Issues. In the case of structures, there are possible interactions between MPI and the compiler [1]. These interactions can affect the way a compiler does padding between one structure and the next.

[1] The authors acknowledge the work of Raja Daoud on this section.

When sending more than one structure, this can be an issue. There are two possible ways to deal with this. One way is to assume default padding and to create the structure type maps based on it. The other way is to explicitly include the upper bound of the structure with `MPI_UB`.

It is useful to assume the default case, but have a test that would indicate if things did not match up. The test case could create both data-types, with and without `MPI_UB`, get their extents, and compare. The current version of AutoMap implements the default, but a future version will provide both options, with a test to ensure correctness if the default is not used.

1.4 Integrated Use of AutoMap

Overview. Whatever version of AutoMap the programmer uses, one will have to run AutoMap on a file that contains its data-structures (`struct.h` for example). In the stand alone version, the generated file containing the MPI data-types (`mpitypes.c`) is directly included in the user program (`UserProg.c` for example). When AutoMap is coupled with the AutoLink library, its output files (`mpitypes.c` and `al_routines.c`) are included in the AutoLink source file `autolink.c`. When writing code, the programmer includes `autolink.c`.

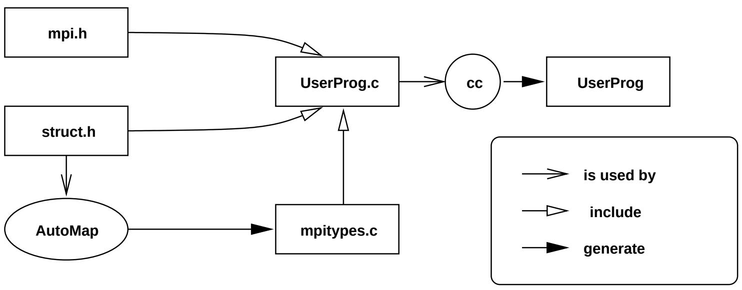

Fig. 1. AutoMap generation of files, Stand Alone version

Figures 1 and 2 illustrate AutoMap in stand alone version and coupled with AutoLink.

Preparing the User-Defined File. The user must modify the source file before running AutoMap. AutoMap recognizes the structures to be converted to MPI types by directives in the C code. In order to read a user's code directly, the directives were designed as modified C comments. The modification consists of a '~' just after the usual beginning of a comment "/*". Additionally, this is followed by an `AutoMap_Begin` for the start directive or an `AutoMap_End` for the end directive. Thus a structure is surrounded by :

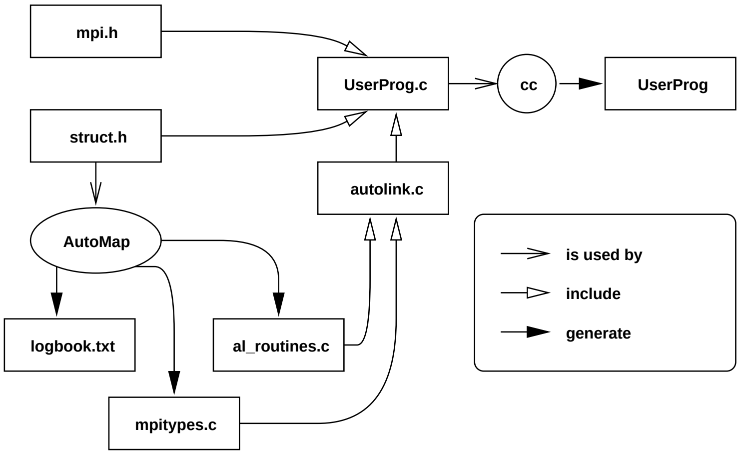

Fig. 2. AutoMap generation of files, AutoLink version

```
/*~ AutoMap_Begin */

/*~ AutoMap_End */
```

and the data-type to be 'AutoMap recognized' by :

```
/*~ AutoMap_TpUsed */
```

Here is an example of file `struct.h` defining the data-structure(e.g. a linked list) after modification by the user :

```
/*~ AutoMap_Begin */

#define SIZE 1000

typedef struct LList *LL;
typedef struct LList
{
  float  data[SIZE];
  LL     next;
} LinkedList /*~ AutoMap_TpUsed */;

/*~ AutoMap_End */
```

1.5 Practical Use of AutoMap in the User Code

Stand Alone Version. In the stand alone version, the user will send complex data-types almost in the same way as standard MPI data-types. First, at the beginning of the main program, the user will have to initialize the new MPI data-types by calling the function `Build_MPI_Types()` defined in the generated file `mpitypes.c`. The actual name of each generated MPI data-type is made up of the initial name as written in the `struct.h` file, to which the prefix `AutoMap_` is added. For instance, for the complex data-structure `cmdline` described in section 1.1, the MPI data-type name would be `AutoMap_cmdline`. Then one will use the MPI data-type name in the usual MPI communication function calls. In our example, to send one variable `var` whose type is `cmdline` :

```
MPI_Send(&var,1,AutoMap_cmdline,next_rank,
         MPI_ANY_TAG,MPI_COMM_WORLD);
```

AutoMap Coupled with AutoLink. When AutoMap is coupled with AutoLink, users will call AutoLink routines to send or receive their dynamic data-structures.

AutoLink's routines consists of two pairs of functions: one for initializing/finalizing AutoLink, one for communicating.

- `AL_Init()` initializes MPI layer, loads the newly created MPI data-types and allocates memory for AutoLink internal structures.
- `AL_Finalize()` finalizes MPI layer and de-allocates memory for AutoLink internal structures.
- `AL_Send()` takes care of the traversing and sending of the dynamic data-structure.
- `AL_Recv()` takes care of the receiving and restoring of the dynamic data-structure.

2 AutoLink : Handling Dynamic Data-Structures

2.1 General Overview of AutoLink

AutoLink is a library that allows the sending and receiving of dynamically linked structures through MPI (a case not handled by 'AutoMap stand alone' version).

In a previous version of AutoLink we had chosen to buffer the whole data-structure before sending or receiving. This was too memory intensive in the case of large dynamic data-structures. On the other hand, sending data without any buffering saves memory but will induce large communication time latency. We choose to send packets of data, which is a good trade-off between memory use and communication time. Additionally, the packet size is configurable.

Furthermore, we manage to overlap communication (sending or receiving of packets) with computation (traversing the data-structure or allocating memory for the data received).

Packet Transmission. From a communication point of view, the general algorithm consists of sending (resp. receiving) the content of the data packet by packet and then the information describing how the data are linked.

- In the sending processor, simultaneously to the traversal of the dynamic data-structure, packets are sent when the packet buffer is full. When the traversal of the data-structure is finished, the information about the links between the data is sent.
- In the receiving processor, as packets arrive, memory space is allocated and the data are copied into memory. When data transmission is over, the data-structure link information is received and the dynamic links restored.

2.2 Storage Structures for Handling the Link Information

Refer to figure 3 for understanding the structures mentioned hereafter.

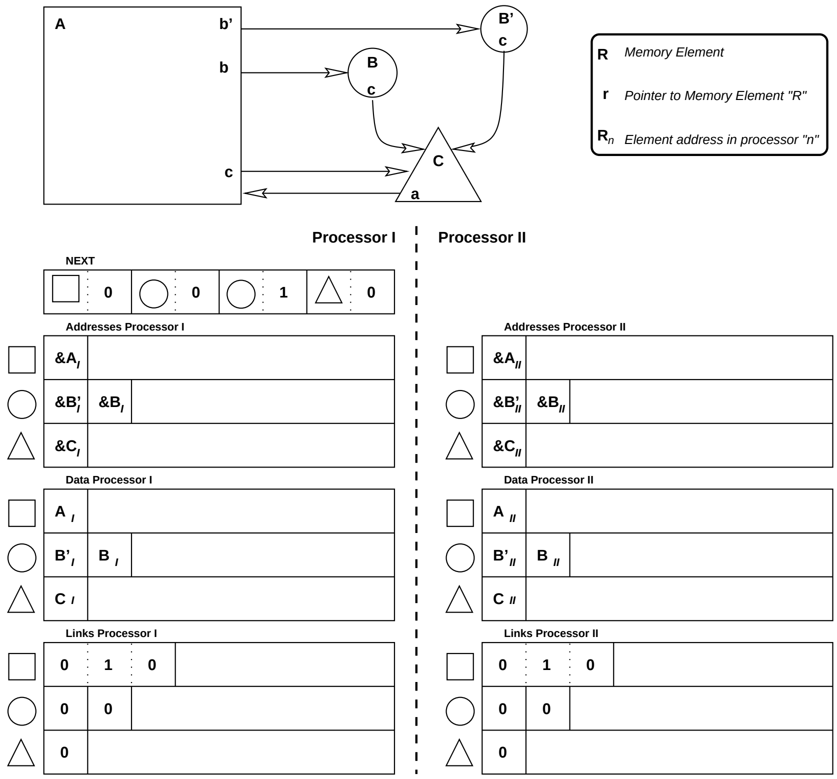

Fig. 3. AutoLink transfer concept. Each shape represents a different type of node. Here there are three types; square, circle, and triangle.

The dynamic pointer structure based on addresses is no longer valid on a remote processor. Therefore a logical representation of the dynamic structure is sent to the remote processor (LINKS array). The logical representation is based on labeling each node of the dynamic structure with a number. Actually there is one labeling per node type. The links between nodes are then represented by labels (integers) instead of pointers. For each node, the label of its children will be stored in the LINKS array.

The role of the traversal function is to construct the array of links sent to the remote processor; on the receiving end, the physical links will be rebuilt from the logical label representation. An iterative breadth-first search is used for traversing the data-structure.

Here are the storage structures used by AutoLink to build the equivalence between the physical and logical representation of the dynamic structure.

NEXT array is useful for the traversing stage. The NEXT array stores the next elements to be visited in the breadth-first search. At each step of the breadth-first search, the children of the current node are stored in NEXT array if they were not marked. In our example (figure 3), in the first step of the breadth-first search, . The children of node A, which are nodes B, B' and C are stored in NEXT. In the second step of the search, node B' is the current node and node C is the child of node B'. Node C won't be stored because it has already been marked during the first step of the search and so forth. Each element of NEXT array is composed of one integer for the data-type (i) and one integer for the label (j). Based on these two indexes, one accesses directly the addresses of a node contained at the position (i, j) in the ADDRESS array. In figure 3, integers representing data-types have been drawn as geometrical shapes. Data-type 0 is represented as a square, data-type 1 as a circle and data-type 2 as a triangle.

Labels are given during the breadth-first search. Let N be the current point of the breadth-first search. For each child of node N, if the child (of data-type i) has not been visited yet, the breadth-first search gives a label j to the child, stores the child data-type and label in NEXT array, and its address in ADDRESS at the position (i, j). There is one labeling per data-type and the label is given in the order of the breadth-first search. For instance, the order of the breadth-first search in our example is : node A, then node B', node B, node C. Node A is the first node of the data-type square, therefore it gets the label 0. In the order of the breadth-first search, node B' is the first node of data-type circle and node B the second one. Thus node B' gets label 0 and node B gets label 1. Node C gets label 0 as first node of data-type triangle.

ADDRESS is a two-dimensional array that stores the address of an element. The address of the j^{th} element of data-type i will be stored in the i^{th} line and j^{th} column. On the sending processor, ADDRESS stores the addresses of the nodes that have already been visited by the breadth-first search, and the addresses of the children of the current node. These addresses are useful for accessing the next node in the next step of the search. On the receiving processor, the ADDRESS array is used to store the address of the newly allocated nodes. These addresses are useful to restore the links.

LINKS is a two-dimensional array that carries the logical representation of the dynamic data-structure. Coupled with ADDRESS, LINKS enables the reconstruction of the data-structure. Each line of LINKS corresponds to one data-type and is decomposed into slots containing as many integers as there are pointers for a given data-type. These slots store the labels given to the children of each node of that data-type.

From AutoMap, AutoLink knows the number of pointers for a given data-type –thus the size of the slot– and their types. For instance, the data-type square contains three pointers : the first two pointers are pointers to data-type circle and the third one points to triangle data-type. Since the nodes of the same data-type are stored together in the same line of LINKS, the reading of a line is completed by a step equal to the size of the slot for this data-type. That way, AutoLink knows the data-type of the element e whose label is currently read. For instance, in figure 3, AutoLink knows that the data-type of the element corresponding to the value read in the second position of the first line of array LINKS is a circle. The value is the label of the element (here the value is 1). Therefore the element that is actually referred to is node B. With the label value, AutoLink knows how to fetch element e. AutoLink reads the address of element e in ADDRESS in the line corresponding to the data-type of element e and in the column given by the label. In our example, it reads the address of node B in the second line, second column of ADDRESS.

TAG or MARK : the library needs a tag (also called mark) to know if an object has already been traversed during the breadth first search. A hash table implements the tag.

2.3 Algorithms

Sending Dynamic Data-Types. Follows a high level algorithm;

```
|Add entry node in NEXT
|Add address of entry node in ADDRESS
|Mark entry node
|While there are elements to visit in NEXT
|  |Reach current node in NEXT
|  |Add node data to PACKET
|  |If PACKET is full, send PACKET
|  |For all children of current node
|  |  |If child does not exist
|  |  |Then Add NO CHILD in LINKS
|  |  |Else |If child has not been marked
|  |  |     |Then |Add child in NEXT
|  |  |     |     |Add address of child in ADDRESS
|  |  |     |     |Mark child node
|  |  |     |     |Add child label in LINKS
```

```
|  |  |     |Else Add child label in LINKS
|  |Go to next in NEXT
|Send last PACKET
|Send LINKS
|Send references of Initial Object
```

Receiving Dynamic Data-Types. High level algorithm for receiving;

```
|While more PACKET to receive
|  |Receive PACKET
|  |For each element in PACKET
|  |  |Create element in memory
|  |  |Add address of created element in ADDRESS
|Receive LINKS
|Receive references of Initial Object
|For each element of LINKS
|  |If value of LINKS is NO CHILD
|  |Then child referred by LINKS refers to no element
|  |Else child refers to correct element in ADDRESS
|Result is Initial Object with recreated links
```

3 Performances Study

There are no performance results yet. Data will be available from the NIST Scientific Applications Support Project Web page (`http://www.itl.nist.gov/div895/sasg/`). The following explains how the design ideas are optimized.

3.1 Memory Performance

Use of a Link Buffer. Because AutoLink was written in C, the memory overhead generated by AutoLink could be decreased by getting rid of LINKS. Currently, a node is copied into the data buffer with its actual data-type made up of data and pointers. MPI data-types generated by AutoMap out of the actual data-types are used as a parameter in the sending and receiving process, so that in the receiving end the data-type of the structures reallocated correspond to the MPI data-types received.

Since C allows to cast a pointer into an integer, the idea would be to use the space occupied by the pointers in the data for storing the link information as integers. The pointer values would be used as labels. One would force C pointers to be recognized by MPI as `MPI_UNSIGNED_LONG`. Yet this cast is not allowed in strong typing languages. That's why we have chosen to keep the method as general as possible.

Tag. A tag is simulated by the use of a Hash Table to avoid having the user add one field to their data-structures [2]. A tag included in the user structure would have been memory consuming, since it is used only once.

The pure Hash Table performance is mostly determined by its H-function, and even more by its size, which is configurable.

Use of Packets. The sending process employs packets to utilize memory more efficiently. The size of the packets is configurable.

3.2 Time Performance

Non Recursive Traversal. The Algorithm developed to improve the traversal part is based on an Iterative Breadth First Search Algorithm.

Overlap Communication & Computation. Non blocking communication is used for the sending, so that the traversal of the dynamic data-structure can be carried out while MPI handles the sending of packets.

Conclusion

AutoMap and AutoLink are user-friendly tools that make the development of MPI-based applications easier. AutoMap dramatically simplifies the creation of MPI data-types. It simply needs to read the user-defined C data-structure and therefore minimizes the intervention from the user. AutoLink handles the transfer of these data-types in a straightforward way.

AutoMap and AutoLink are flexible and portable tools that can run on any MPI-enabled platform. By removing the bothersome complexity of MPI programming from the user, these tools enable the design of more complex applications.

References

1. Message Passing Interface Forum http://www.mpi-forum.org/docs/docs.html
2. Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein : Data Structures Using C. Prentice Hall (1990)
3. William Gropp, Ewing Lusk and Anthony Skjellum : Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press, Cambridge, MA (1994)
4. K. H. J. Vrielink, E. C. Baland and J. E. Devaney : AutoLink: An MPI Library for Sending and Receiving Dynamic Data Structures. International Conference on Parallel Computing, University of Minnesota Supercomputer Institute (October 3-4 1996)

[2] Note also that the addition of an initialized value to a struct is impossible in C, so the use of a tag inside the C struct would have to be initialized by the user.

5. Yacc++ and the Language Objects Library Reference Guide. Compiler Resources, Hopkinton, MA (1996)

Organizations

- **NIST**
  National Institute of Standards and Technology
  Web page : http://www.nist.gov/
- **RÉSÉDAS**
  Web page : http://www.loria.fr/equipes/resedas/
- **Télécom INT**
  Télécom Institut National des Télécommunications
  Web page : http://www.int-evry.fr/
- **University of Twente**
  Web page : http://www.utwnte.nl/

Disclaimer

Certain commercial products may be identified in order to adequately specify or describe the subject matter of this work. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available for the purpose.