AUTOMAP/AUTOLINK USERS' MANUAL

Olivier NICOLAS
Guest Researcher at NIST
E-mail : olivier.nicolas@nist.gov

May 15, 2000

Contents

Chapter 1

Downloads

Go to `http://math.nist.gov/savg/parallel/auto/v3.00/download.html`

The latest version of this users' manual, can be found at :
`http://math.nist.gov/savg/parallel/auto/papers/AutoMapAutoLinkUsersManual.ps.gz`

You can also get the latest version of the examples at :
`http://math.nist.gov/savg/parallel/auto/Download/Examples_Latest.tar.gz`

Chapter 2

Installing the software

To install the version of AutoMap and Autolink you downloaded, you just need to uncompress the .tar.gz files in a directory of yours (Using “gunzip -dc *.tar.gz | tar xvf -”) rexpecting the kind of following architecture :

MPIdtt_v3.00b1
MPIdtt_v3.00b1/AutoLink
MPIdtt_v3.00b1/AutoLink/Src
MPIdtt_v3.00b1/AutoMap
MPIdtt_v3.00b1/AutoMap/Src
MPIdtt_v3.00b1/AutoMap/Exec
MPIdtt_v3.00b1/Examples
MPIdtt_v3.00b1/Examples/commonFiles
MPIdtt_v3.00b1/Examples/AutoLink
MPIdtt_v3.00b1/Examples/AutoMap
MPIdtt_v3.00b1/Examples/commonFiles
MPIdtt_v3.00b1/Examples/Docs

NOTES

1. AutoMap

   The *AutoMap* executable file will be in *AutoMap/Exec/*, in the directory corresponding to your architecture (SGI, Solaris, Linux, etc.).

2. AutoLink

   When using AutoLink, you need to set a *define* called *ALHOME*, in the Makefile of your program, to the directory where the *autolink.inc* file is (*AutoLink/Src/*).

TROUBLESHOOTING

Some versions of MPI happen to have what we called the *Padding problem* (a bug fixed in the latest versions of the MPI software). AutoMap/AutoLink can work correctly with this problem anyway, if you inform them your version of MPI has the problem.

After installing the software, you will have to perform a test on each machine you intend to use :
for each machine, compile and run the test in *Examples/AutoMap/paddingBug* with 2 processes.

If the result of the test is :

1. **padding compliant**
   You don’t need to do anything.

2. **padding problem present**
   The problem can be fixed as shown with the example in *Examples/AutoMap/paddingTest*

   (a) Compile and run the example with 2 processes.

   (b) You should have as a result :

   ```
   [ch12]
   []
   ```

   saying that the string "*ch12*" sent hasn't been correctly received.

   (c) Now, edit the `Makefile` to uncomment the "*FLAG define*" line, re-compile and re-run.

   (d) You should then have as a result :

   ```
   [ch12]
   [ch12]
   ```

   saying that the string "*ch12*" sent has been correctly received.

   Summary : include the "*FLAG = -DAM_PC*" in your Makefiles (on the systems tested as non compliants) to avoid the *Padding problem*.

3. **padding problem persistant**
   Contact martial.michel@nist.gov

Chapter 3

Using the software

3.1 Usage on the Web

You can also use AutoMap on the web.

Go to `http://math.nist.gov/savg/parallelauto/v3.00/automap_web.html` and fill the box with the content of your data-structure file.

It will generate the files you need to include into your source code.

3.2 The installed version

3.2.1 AutoMap stand alone

What does AutoMap do?

AutoMap automatizes the process of data-type creation for MPI; it parses the C header-file given as a parameter to generate the MPI-data-types corresponding to the *typedef struct* found in the file.

The command line

Type *AutoMap -help* to have a list of the options available :

```
Location :  AutoMap
Identification :  AutoMap 3.00 [beta 1]

AutoMap [-help] [-v] [-log] [-noAL] filename
-help :  Will print this help menu
-v :  Verbose mode
-log :  Will generate the "logbook.txt" for this run
-noAL : Will not generate the entries for use with AutoLink
filename :  name of the C typedef definition file to analyze
```

Let's learn it with a minimal example!

1. **The header file for AutoMap**

   Let's say you want to use, with MPI, the following structure :

   ```
   struct element
   {
   int value;
   };
   ```

Then, you can transform your *.h* file to look like this :

```
/*~ AM_Begin*/
typedef struct element element /*~ AM*/ ;

struct element
{
int value;
};
/*~ AM_End*/
```

Everything between `/*~ AM_Begin */` and `/*~ AM_End */` will be analyzed by AutoMap. The `/*~ AM */` flag will tell AutoMap to map this structure for MPI, by generating a new MPI-type that will be named after the name of the structure by adding the *AM_* prefix (here, the name will then be `AM_element`).

2. **AutoMap processing**

   Type "*AutoMap -noAL short.h*" where *short.h* is the file containing the above code. The "*-noAL*" option makes AutoMap generate no code for AutoLink for in this example we want to use AutoMap stand alone only. You can generate the code for AutoLink, it's just not needed in this example.

```
AutoMap -noAL short.h

--> Generated output for :
----> element
--> done
AutoMap finished without error

Output files can be found in current directory.
0 Errors, 0 Warnings, Highest Severity 0
```

   2 files have then been generated (*mpitypes.h* and *mpitypes.inc*) in the current directory.

3. **The *main* function**

   In your ".c" main file you will need to add the following :

   Headers :

```
#include <mpi.h>
#include "mpitypes.inc"
```

   Beginning of your main function :

```
MPI_Init(&argc, &argv);
Build_MPI_Types(); /* to create the new MPI data-types */
```

   End of your main function :

```
Free_MPI_Types(); /*for cleaning the MPI data-types */
MPI_Finalize();
```

   The only thing needed to send your data is to provide the name of the type you wish to transfer to the send and receive functions; in this case : "*AM_element*".

To transfer "`element myElement;`" use :

```
MPI_Send(&myElement, 1, AM_element, sendTo, 0, MPI_COMM_WORLD)
```
and
```
MPI_Recv(&myElement, 1, AM_element, recvFrom, 0, MPI_COMM_WORLD, &status)
```

according to the syntax of the MPI functions *MPI_Send* and *MPI_Recv*.

Want to have a look at the source code?

You can find the source code of this example in your "AutoMap-Example" directory
(Ex : *Examples/AutoMap/shortExample*).

Summary

1. **Have a file containing your AutoMap specifications**

   Your file will look like this :

```
/*~ AM_Begin */

/* first the typedefs */

typedef struct struct1 struct1 /*~ AM */;
typedef struct struct2 struct2 /*~ AM */;
...
typedef struct structN structN /*~ AM */;

struct struct1
{
...
};

struct struct2
{
...
};

...

struct structN
{
...
};

/*~ AM_End */
```

2. **Use AutoMap to generate the code to include in your program** (*AutoMap -noAL short.h*)
3. **In your code, include** *mpitypes.inc*, **and calls to** *Build_MPI_Types()* **and** *Free_MPI_Types()*
4. **Use the MPI standard functions to send and receive data with your new types (prefixed by AM_)**
5. **You're ready.**

Want to see more examples of AutoMap stand alone?

You can find several examples of AutoMap stand alone in your "AutoMap-Examples" directory.
(Ex : *Examples/AutoMap/Examples*).

3.2.2 AutoMap with AutoLink

What does AutoLink do?

AutoLink is a tool/library that allows you to transfer, via MPI, data structures containing pointers. AutoLink, being an extension to AutoMap, works exclusively on *typedef structs*.

Let's learn it with a minimal example!

1. **The header file for AutoMap**

   AutoLink needs the output of AutoMap in order to work.

   Let's say you want to send, via MPI, the following structure :

   ```
   struct element
   {
   int *value;
   };
   ```

   Then, you will have to modify your code to pass to AutoMap a C header file that looks like this :

   ```
   /*~ AM_Begin */

   typedef struct element element /*~ AM */;
   typedef struct data data /*~ AM */;

   struct data
   {
   int value;
   };

   struct element
   {
   data *dataElement;
   };

   /*~ AM_End */
   }
   ```

   Why do you have to modify your code this way? Because AutoMap and AutoLink work exclusively on *typedef structs* and AutoLink requires its data to point to user defined data-types.

2. **AutoMap processing**

   So, you type "*AutoMap short.h*" to generate the necessary files :

   mpitypes.h
   mpitypes.inc
   al_routines.h
   al_routines.inc
   autolink.h

3. **The *main* function**

   In your ".c" main file you will have the following :

   Headers :

   ```
   #include <mpi.h>
   #include "autolink.inc" /* will include mpitypes.inc */
   ```

Beginning of your main function :
```
MPI_Init(&argc, &argv);
AL_Init(-1); /* also calls Build_MPI_Types */
```

End of your main function :
```
AL_Finalize(); /* also calls Free_MPI_Types */
MPI_Finalize();
```

To transfer your data, you'll need to use AutoLink's specific functions :

(a) **For a blocking usage :**
```
AL_Send
AL_Recv
```

(b) **For a non-blocking usage :**
```
AL_ISend
AL_ISendTest
AL_ISendWait
AL_ISendComplete
AL_IRecv
AL_IRecvTest
AL_IRecvWait
AL_IRecvComplete
```

You have to declare the data you want to receive as a pointer. This is because the "AutoLink receive functions" require a pointer to a pointer. Do not allocate any memory for the data you will receive, for AutoLink will do it for you; you would waste memory (that you couldn't free) by doing so!!

It means you have to declare "*element *myElement*"; not "*element myElement*";

The blocking version functions

1. **In general**

   Two functions are at your disposal : *AL_Send* and *AL_Recv*.

   (a) **AL_Send**

   int AL_Send(void* buf, int datatype, int dest, int tag, MPI_Comm comm);

buf	Initial address of send buffer
datatype	Data-type to send
dest	Rank of destination
tag	Message tag
comm	Communicator

   Use this function to send your data-type.

(b) **AL_Recv**

int AL_Recv(void** buf, int* datatype, int source, int tag, MPI_Comm comm, MPI_Status* status, AL_ReqList** rqlist);

buf	Pointer to the initial address of receive buffer
datatype	Pointer to the data-type to receive (will be filled for you)
source	Rank of source
tag	Message tag
comm	Communicator
status	Return status
rqlist	Specify the memory allocation method

Use this function to receive your data-type.

2. **With our example**

   AL_Send(myElement, AL_element, sendTo, 0, MPI_COMM_WORLD);
   AL_Recv((void *) &myElement, &type, recvFrom, 0, MPI_COMM_WORLD, &status, NULL);

The non-blocking version functions

1. **AL_ISend**

   int AL_ISend(void* buf, int datatype, int dest, int tag, MPI_Comm comm, AL_ISendRL** rlist);

buf	Initial address of send buffer
datatype	Data-type to send
dest	Rank of destination
tag	Message tag
comm	Communicator
rlist	Specific information for transfer completion

2. **AL_ISendTest**

   int AL_ISendTest(AL_ISendRL **rlist);

   rlist Specific information for transfer completion

3. **AL_ISendWait**

   int AL_ISendWait(AL_ISendRL **rlist);

   rlist Specific information for transfer completion

4. **AL_ISendComplete**

   int AL_ISendComplete(AL_ISendRL **rlist);

   rlist Specific information for transfer completion

5. **AL_IRecv**

   int AL_IRecv(void** buf, int* datatype, int source, int tag, MPI_Comm comm, AL_IRecvRL** rlist, AL_ReqList** rqlist);

buf	Pointer to the initial address of receive buffer
datatype	Pointer to the data-type to receive (will be filled for you)
source	Rank of source
tag	Message tag
comm	Communicator
status	Return status
rlist	Specific information for transfer completion
rqlist	Specify the memory allocation method

6. **AL_IRecvTest**

   int AL_IRecvTest(AL_IRecvRL **rlist);

   rlist Specific information for transfer completion

7. **AL_IRecvWait**

   int AL_IRecvWait(AL_IRecvRL **rlist);

   rlist Specific information for transfer completion

8. **AL_IRecvComplete**

   int AL_IRecvComplete(AL_IRecvRL **rlist);

   rlist Specific information for transfer completion

Other AutoLink functions

1. **AL_Init**

   void AL_Init(int packetSize);

   packetSize size of the packets in bytes

   If packetSize<0, AutoLink uses the default size for the packets to send (see "*al_common.h*" definitions.

2. **AL_Finalize**

   void AL_Finalize();

3. **AL_LogEntry**

   int AL_LogEntry(char *entry);

   entry string you want to add to the log file (cf LogFile chapter)

4. **AL_SetPacketSize**

   int AL_SetPacketSize(int size);

   size size of the packets in bytes

5. **AL_Free**

   void AL_Free(AL_ReqList** rlist);

   rlist specific information for transfer completion

Note : the rlist parameter seen in all those functions needs to point to the same list for one session!

Compiling an AutoLink program

See the Makefile in the given examples.

Want to have a look at the source code?

You can find the source code of this example in your AutoLink-Examples directory
(Ex : *Examples/AutoMap/Examples/shortExample*).

Summary

1. **Have a file containing your AutoMap specifications (Ex :*short.h*)**
2. **Use AutoMap to generate code to include in your program (*AutoMap short.h*)**
3. **In your code, include *autolink.inc*, and calls to *AL_Init()* and *AL_Finalize()***
4. **Use the AutoLink functions to send and receive data with your new types (prefixed by AL_)**
5. **You're ready**

Want to see more examples of AutoLink?

You can find several examples of AutoLink used with AutoMap in your AutoLink-Examples directory
(Ex : *Examples/AutoMap/Examples*).

Chapter 4

More things you need to know

1. It is not possible to map every type via AutoMap, due to the technical specifications of AutoMap/AutoLink; AutoLink needs fixed sized data-types to work.

   For example, you cannot use *char**. Use *char[n]* instead.

2. AutoMap doesn't read files included with the *#include* mechanism.

3. AutoMap doesn't perform macro text replacement.

4. AutoMap doesn't perform any computation

   Ex :

   ```
   char b[2*3];
   ```

   won't be recognized by AutoMap's grammar.

5. Mapping of pointers of arrays are not implemented yet.
   (Ex : *int (array *) [2]*)

6. Be sure, when using AutoLink to put every non-used pointer to NULL before sending any data.

7. AutoLink needs fully defined fixed size entries to work!

8. Also, don't map a data-type architecture containing 2nd level (any level>1 in fact) pointers; it's not yet supported.
   Ex of a 2nd level pointer :
   A points to B
   B contains C
   C points to D

   AutoLink won't traverse the C type and so won't access D.