A Macro Communication Package for Master-Slave models on DMPPs

René M. Rehmann

October 27, 1992

Abstract

In this report we describe a communication package which is designed to support regular process topologies, e.g. master-slave models, on a MPP system. The package allows to address a neighboring process with a relative address and includes functions like broadcasting and combine. The communication package has been implemented as C preprocessor macro package using ANSI-C syntax to preserve efficiency and portability among compilers.

1 Introduction

Computational problems in natural science and engineering are characterized by the application of local functions to large and homogeneous data structures. To run these types of applications on multiprocessor systems with distributed memory, they are often mapped to homogeneous process topologies representing the regular structure of the underlying model. One process topology model which has proven to be efficient for these types of applications is the decomposition model which is also known as the master-slave model. In the master-slave model the large and homogeneous data structure is decomposed by the master-process and each slice of the decomposed structure is distributed to a process which is called slave-process. The local functions are applied on each part of the distributed data structure with data exchange at the boundaries. To calculate global values of the distributed data structure, every slave-process calculates local values of its part of the data structure and communicates the value to the master-process which calculates the global value from the local values of the slave processes.

To realize the master-slave computational model we only need communication paths from one slave-process to its neighboring slave-processes and from each slave-process to the master-process and vice versa. But most of the vendor-supplied communication packages allow for arbitrary point-to-point communication which is more powerful but often more complicated for programming as well. Therefore we developed a communication package specifically designed for regular, grid-like process- and communication structures which allows easy nearest-neighbor communication with relative direction names (e.g. north, south, east, west, etc.) for the recipient or sending process. Additionally the package provides functions which are often missing in vendor-supplied packages and which are useful for regular topologies. The function `broadcast` for example allows sending a message from the master-process to all slave-processes. This is often useful for sending synchronization signals and for distributing global parameter values.

The communication package is implemented as C preprocessor macro function package to fully guarantee performance and flexibility. Portability among different C compiler is guaranteed by using the ANSI-C preprocessor directives and syntax.

The macro package consists of function definitions for three different areas of usage. Two macro functions for usage within the declaration or definition part of a program, one function for the initialization part of the program and a set of functions for the communication. The communication types supported in this package are synchronous, which means that both sender and receiver process are blocked until they are both ready to exchange data, and buffered synchronous transmission which means that one process can proceed if the other process is not ready to receive a message or if the other process has not yet sent the message. To verify that the message has arrived you have to test with a separate function for completeness. Additionally you can choose whether you want to exchange messages with XDR[Sun90] conversion. This is especially useful in heterogenous networks.

2 Macro Functions

2.1 Directives for Macros

There are several macro variables which influence the behaviour of the macro functions during compilation and execution. Some of these directives are mandatory and others are optional.

Mandatory:

`DIMENSION`	`this variable defines the number of dimensions of the process topology.`
`IS_MASTER`	`this variable defines that this module belongs to a process which acts as a master process. Only one of IS_MASTER or IS_SLAVE can be used for a module.`
`IS_SLAVE`	`this variable defines that this module belongs to a process which acts as part of a slave process. Only one of IS_MASTER or IS_SLAVE can be used for a module.`

Optional:

`COMM`	`this variable is for debugging purposes only and defines, if a message should be printed for each call of a communication function. All parameters of a communication function and the module location is printed if this variable is set to 1.`
`PROC_NAME`	`this variable is used by COMM to identify the location of a communication call. It should be set once for each process module.`
`TASK_NAME`	`this variable is used by COMM to identify the location of a communication call. It should be set once for each task module.`

`TIMEOUT`	`this variable is set by default to -1 and sets the timeout for a communication to infinity. The only purpose to change the value is to find deadlocks in programs. The number is given in milliseconds.`

2.2 Declaration

There are two functions which either define or declare the needed variables for the communication functions. The function `declare_transports_and_names` defines the necessary variable names and channel names[1] for the communication functions. The function `extern_declare_transports_and_names` is used in separatly compiled modules and only declares the needed variable names as external. Both functions are used without any parameters.

Example:

```
  .
  .
#include "COMMLIB.h"
extern_declare_transports_and_names;
  .
  .

my_function_definition_xyz() {
  ...
}
```

2.3 Initialization

Only one function is needed to allocate and initialize the variables and buffers and to establish the communication network from the master- to all slave-processes and vice-versa and between all slave-processes in the desired topology. Additionally this functions checks if the correct number and values of command line arguments is present. The command line arguments are described in the next section. This function call should be the first statement of the `main`- or `driver`-program after the declaration of the variables.

The function `open_and_register_transports` initializes the variables and buffers and opens and establishes the communication links between all processes. This functions initializes the process topology according to the value of the macro variable `DIMENSION`.

Useful C-variables which are declared by `declare_transports_and_names` and initialized by `open_and_register_transports`:

`num_slaves`	`number of slave-processes for this execution. Is set by the first argument of the master-process argument-list.`
`proc_number`	`number of this slave-process in the range of 0 to num_slaves-1. This is set to MAXINT if this is the master-process.`

[1]The channels are called transports in the Meiko CSTools communication package which is the basis of the macro package

2.4 Communication Functions

Relative Addresses

As mentioned earlier, all communication functions use relative addresses for sending messages to other processes. The name of these addresses are specifically tailored for the master-slave model. All addresses can be used for either sending or receiving a message. Note that not all addresses are valid for both types of processes and for all dimensions of the slave topology.

Addresses valid on master process:

`SLAVE(i)`	`send (receive) a message to (from) slave i, where i is between 0 and num_slaves-1`
`SLAVES`	`send (receive) a message to (from) all slaves, e.g. broadcast a message`

Addresses valid on slave processes:

The maximum number of dimensions supported are four dimensions. We assign each of the four dimensions a name: `row`, `column`, `plane`, and `cube` to depict the underlying geometric structure.

`MASTER`	`send (receive) a message to (from) master-process`
`NORTH`	`send (receive) a message to (from) slave-process column+1`
`SOUTH`	`send (receive) a message to (from) slave-process column-1`
`EAST`	`send (receive) a message to (from) slave-process row+1`
`WEST`	`send (receive) a message to (from) slave-process row-1`
`UP`	`send (receive) a message to (from) slave-process plane+1`
`DOWN`	`send (receive) a message to (from) slave-process plane-1`
`FRONT`	`send (receive) a message to (from) slave-process cube+1`
`BACK`	`send (receive) a message to (from) slave-process cube-1`

Functions

send(to,buffer,buffer_size)

```
int                 to          /* destination address */
char *              buffer      /* transfer buffer */
int                 buffer_size /* size of transfer buffer */
```

Macro function for sending a message in blocking, synchronous mode. The destination address is one of the relative addresses described above. The message buffer is given by a pointer to a character string, other pointers must be casted. The length of the message buffer must always be given as integer in order to allow sending of other data structures than character strings.
The special destination name `SLAVES`, which is only valid on the master-process, is used to imitate a broadcast, i.e. to send the same message to all slave-processes.

xdr_send(to,buffer,buffer_size,elsize,xdr_func)

```
int                 to          /* destination address */
char *              buffer      /* transfer buffer */
int                 buffer_size /* size of transfer buffer */
int                 elsize      /* size of an data element of the data */
bool_t (*xdr_func)(XDR *,...)   /* function to convert one element to XDR */
```

Macro function for sending a message, converted with XDR to a machine independent form, in blocking synchronous mode. `To`, `buffer`, and `buffer_size` correspond to the arguments used for the `send` function. The size of a single data element is used to determine how many elements of an array must be converted. If `elsize` is not equal to `buffer_size`, the `xdr_array` function is used to convert the buffer. `xdr_func` is the pointer to a function which converts one single data element. Standard types can be converted using the function provided by the XDR-library. For data structures and newly defined data types, the programmer must provide the function (see [Sun90] for an example).
The special destination name `SLAVES`, which is only valid on the master-process, is used to imitate a broadcast, i.e. to send the same message to all slave-processes.

queue_send(to,buffer,buffer_size)

```
int                 to          /* destination address */
char *              buffer      /* transfer buffer */
int                 buffer_size /* size of transfer buffer */
```

Macro function for sending a message in non-blocking, buffered mode. `To`, `buffer`, and `buffer_size` correspond to the arguments used for the `send` function. The arrival of a message sent with `queue_send` must be checked with `test` (see below) to insure correct behaviour of the communication system.
The special destination name `SLAVES`, which is only valid on the master-process, is used to imitate a broadcast, i.e. to send the same message to all slave-processes.

receive(from,buffer,buffer_size)

```
int                 from        /* address of sender */
char *              buffer      /* transfer buffer */
int                 buffer_size /* size of transfer buffer */
```

Macro function to receive a message in unbuffered, blocking mode from a specified address. The sender address must be one of the relative addresses described above. The message buffer is given by a pointer to a character string. The space of this buffer in `buffer_size` bytes must be preallocated by the programmer.
The special sender address `SLAVES` is used to receive a message from all slaves. In this case the user must provide a buffer which is a large as the sum of all transfer buffers of the clients (NUM_SLAVES * buffer_size). The messages are written to this buffer `in order of arrival`. Please note that there is no relationship between the order in the receiving buffer and the numbering of the slave processes.

xdr_receive(from,buffer,buffer_size,elsize,xdr_func)

```
int                  from        /* address of sender */
char *               buffer      /* transfer buffer */
int                  buffer_size /* size of transfer buffer */
int                  elsize      /* size of an data element of the data */
bool_t (*xdr_func)(XDR *,...)    /* function to convert one data to XDR */
```

Macro function for receiving a message in unbuffered, blocking mode with additional XDR-conversion. `From`, `buffer`, and `buffer_size` correspond to the arguments used in function `receive`. `elsize` is the size of a single data element of the transferred array (if any). `xdr_func` is a pointer to the function to convert one single element from XDR representation to machine representation. Standard types can be converted using the function provided by the XDR-library. For data structures and newly defined data types, the programmer must provide this function (see [Sun90] for an example).
The special sender address `SLAVES` is used to receive a message from all slaves. In this case the user must provide a buffer which is a large as the sum of all transfer buffers of the clients (NUM_SLAVES * buffer_size). The messages are written to this buffer `in order of arrival`. Please note that there is no relationship between the order in the receiving buffer and the numbering of the slave processes.

queue_receive(from,buffer,buffer_size)

```
int                  from        /* address of sender */
char *               buffer      /* transfer buffer */
int                  buffer_size /* size of transfer buffer */
```

Macro function fro receiving a message in buffered, non-blocking mode. `From`, `buffer`, and `buffer_size` correspond to the arguments used in function `receive`. The arrival of a message sent with `queue_receive` must be checked with `test` (see below) to insure correct behaviour of the communication system.
The special sender address `SLAVES` is used to receive a message from all slaves. In this case the user must provide a buffer which is a large as the sum of all transfer buffers of the clients (NUM_SLAVES * buffer_size). The messages are written to this buffer `in order of arrival`. Please note that there is no relationship between the order in the receiving buffer and the numbering of the slave processes.

test(type,address,buffer)

```
int           type      /* type of communication (either send or receive) */
int           address   /* address of sender or recipient */
char *        buffer    /* size of transfer buffer */
```

Macro function to test whether a message sent with `queue_send` has been received at the destination address or a message queued up with `queue_receive` has already been received. The function blocks until the message is transferred or until timeout (see `TIMEOUT`).

3 Command Line Arguments

There are a number of command line arguments which finally influence the size of the process topology and the number of slave-processes to be used. Whereas the master-process only needs to know how many slaves-processes are available during the execution, the slave-processes also need to know the number of processes in each dimension. The following list presents the command line arguments which must be specified by the user.

Master-Process

`ARGV[1]`	`number of slave processes during current execution`

Slave-Process

`ARGV[1]`	`logical number of slave process during current execution, should be numbered from 0 to num_slaves-1`
`ARGV[2] to ARGV[DIMENSION+1]`	`size of dimension in north/south, east/west, up/down, and front/back direction in this order. If the topology is less than four dimensions only the equivalent number of command line arguments need to be specified.`

4 Restrictions and Implementation Details

There are some restrictions in the current implementation of the macro comunication package which cannot be raised due to the nature of implementing the package as macro functions.

The functions `xdr_queue_send` and `xdr_queue_receive` are completely missing because the memory management routines could not be implemented as macros. Whenever the `send`-buffer should be converted to XDR, the macro function allocate memory to hold the converted bytes and, in case of the synchronous communication, it frees the buffer after the communication has taken place. In case of the buffered communication only the `test`-function can free the space, as only this function can check whether the communication is already finished or not. This means that the pointer of the temporary allocated space for this communication must be known to the `test`-function. This would imply to implement a linear list of pointers to temporary buffers, because other `xdr_queue_send`-functions can be called in the meantime and can allocate space for temporary buffers before freeing the first one. This implementation of a linear list is very hard to do in a macro package.

Another problem which arises when implementing a macro package is the problem of errors resulting of wrong usage of the macro functions. These errors are often detected by the compiler and result in strange error messages pointing to the line where the macro function is called. Most of the time, these errors are the results of using a wrong parameter type or a false keyword in a parameter list of a macro function.

The current implementation is restricted for usage with the CSTools communication library of Meiko [Mei90]. We used version 1.19 of the CSTools library for testing the implementation. It should be easy to port the package to other communication libraries because the package only depends on having functions for buffered and non-buffered send and receive, a test function for testing if buffered functions completed, and the XDR-conversion library.

5 Conclusion

Implementations of applications and the feedback of users using the packages have shown that the macro package is easy to use and for this type of applications very handy. It has also shown that the package is not only useful for proptotyping of application but having the same efficiency as programming the communication calls directly, it can also be used in full-scale applications. Even with that small number of functions, the package is powerful enough to fit the needs of users programming applications in the master-slave computational model on distributed memory parallel machines.

References

[Mei90] Meiko Ltd., Bristol. *CSTools - Communicating Sequential Tools*, 1990.

[Sun90] Sun Microsystems, Inc. *Network Programmering Guide*, 1990.