MPI for apeNEXT

02.09.2004

In this document we specify the implementation of a subset of MPI functionality on apeNEXT and describe the incompatibilities with respect to the MPI-1.1 standard[1].

Contents

1 Overview **5**
- 1.1 Incompatibilities with respect to MPI-1.1 . . . 5
- 1.2 Processes . . . 5
- 1.3 Communication modes . . . 6
- 1.4 Supported data types . . . 7
- 1.5 Send/receive operations . . . 7

2 Implemented functions **9**
- 2.1 Standard MPI functions . . . 9
  - 2.1.1 MPI_Init . . . 9
  - 2.1.2 MPI_Initialized . . . 9
  - 2.1.3 MPI_Finalize . . . 10
  - 2.1.4 MPI_Abort . . . 10
  - 2.1.5 MPI_Comm_size . . . 10
  - 2.1.6 MPI_Comm_rank . . . 10
  - 2.1.7 MPI_Get_processor_name . . . 11
  - 2.1.8 MPI_Barrier . . . 11
  - 2.1.9 MPI_Wtime . . . 11
  - 2.1.10 MPI_BCast . . . 12
  - 2.1.11 MPI_Allreduce . . . 12
  - 2.1.12 MPI_Send . . . 13
  - 2.1.13 MPI_BSend . . . 13
  - 2.1.14 MPI_Recv . . . 14
  - 2.1.15 MPI_APE_Send . . . 14
  - 2.1.16 MPI_APE_Recv . . . 14
  - 2.1.17 MPI_Sendrecv . . . 15
  - 2.1.18 MPI_Sendrecv_replace . . . 15
- 2.2 Non standard MPI functions . . . 16
  - 2.2.1 MPI_GetNodeAbsX . . . 16
  - 2.2.2 MPI_GetNodeAbsY . . . 16
  - 2.2.3 MPI_GetNodeAbsZ . . . 16
  - 2.2.4 MPI_GetNextNodeX . . . 16
  - 2.2.5 MPI_GetPrevNodeX . . . 16
  - 2.2.6 MPI_GetNextNodeY . . . 16
  - 2.2.7 MPI_GetPrevNodeY . . . 17

2.2.8 MPI_GetNextNodeZ 17
2.2.9 MPI_GetPrevNodeZ 17
2.3 Library internal functions 17
2.3.1 _MPI_Topology 17
2.3.2 _MPI_Neighbours 17

3 Appendix **18**
3.1 MPI communication modes 18

1 Overview

We generally follow the MPI-1.1 Standard. But, since the MPI interface assumes a very general communication models, it is very hard to support the full MPI functionality on a special architecture like apeNEXT. Some functions would become prohibitively inefficient.

1.1 Incompatibilities with respect to MPI-1.1

This implementation of MPI for apeNEXT does not have the goal to support the full MPI standard. This would be hard to achieve as MPI assumes a MIMD programming model.[1]

- Only **communicator** `MPI_COMM_WORLD` is supported.
- No support for **request handles** is implemented. Functions providing pointers to request handles will not change this pointer, except `MPI_Sendrecv()`.
- The following **data types** are not supported:
  - `MPI_CHAR`
  - `MPI_UNSIGNED_CHAR`
  - `MPI_BYTE`
  - `MPI_PACKED`
- Not implemented/supported functions:
  - `MPI_Buffer_attach`, `MPI_Buffer_detach`[2], `MPI_Get_count`, `MPI_Probe`, `MPI_RSend`, `MPI_SSend`, `MPI_Test`, `MPI_Testany`, `MPI_Testall`, `MPI_Testsome`, `MPI_Wait`, `MPI_Waitany`, `MPI_Waitall`, `MPI_Waitsome`, `MPI_Probe`, `MPI_IProbe`, `MPI_Cancel`, `MPI_Test_cancelled`
  - Functions related to persistent send requests

1.2 Processes

Since on APE machines exactly one process is executed at a time on each node, there is a *one-to-one* correspondence between nodes and processes, and we user the two notions as synonymous in the following.

[1] [1], p. 12.

[2] We plan to support `MPI_Buffer_attach` and `MPI_Buffer_detach` in future.

1.3 Communication modes

The following rules impose restrictions to the supported communication modes:

1. Each node can communicate only across up to three orthogonal links to all nodes on a cube, i.e. connecting nodes at relative distance $(\Delta_x,\ \Delta_y,\ \Delta_z)$ with $|\Delta_i| \leq 1$.
2. Communication between two nodes have to be /em homogeneous if the distance between both nodes is > 1, this means that all nodes my make communication at the same time in the same (relevant) direction.
3. Receive operations are always blocking. No mechanism for posting receive operations and no support for request handles is foreseen.
4. Send operations are always non-blocking.
5. Communications can not be canceled.

MPI provide four communicational modes, but we support only *buffered mode*:

- Standard mode is mapped to buffered mode.
- Buffered mode:
  - Send function triggers copy of send data into send buffer in memory.
  - Receive operation triggers network transfer.
  - Buffer released after (blocking) receive operation completed
  - Message size limited by memory space.
  - Only homogeneous communication allowed.
  - Only 1-, 2- and 3-Step communications are allowed.
- Synchronous mode is not supported since there is no mechanism to post receive operations foreseen.
- Ready mode is not supported since there is no mechanism foreseen to post receive operations[3].

We also support APE (hardware oriented) send mode:

- Send function triggers send operation.
- Send data is "buffered" in `TXFIFO`, i.e. message size is limited to `MPI_FIFO_SIZE`.
- Receive function triggers receive operation.
- Inhomogeneous communication are allowed.
- Sender and receiver must be directly connected (1-link communication).

[3]Although hardware would allow to "post" a single receive operation up to `MPI_FIFO_SIZE`

1.4 Supported data types

apeNEXT is a 64-bit architecture and all data types have the same length. We don't support `MPI_CHAR`. All MPI data types are mapped to four groups of *C*-types implemented on apeNEXT (see Table 1.1). Most functions do not care about the data type, except `MPI_Allreduce()`.

MPI name	C equivalent	Size on APE

`MPI_INT`
`MPI_LONG`
`MPI_SHORT`
`MPI_BYTE`
`MPI_LOGICAL`
	integer
	long
	short int
		$2 * 64$ bit
		mirrored

`MPI_UNSIGNED`
`MPI_UNSIGNED_SHORT`
`MPI_UNSIGNED_LONG`
	unsigned integer
	unsigned short int
	unsigned long
		$2 * 64$ bit
		mirrored

`MPI_DOUBLE`
`MPI_FLOAT`
`MPI_LONG_DOUBLE`
`MPI_DOUBLE_PRECISION`
	double
	float
	long double
		64 bit +
		64 bit zero

`MPI_COMPLEX` `MPI_DOUBLE_COMPLEX`	—	$2 * 64$ bit
`MPI_CHAR` (*) `MPI_UNSIGNED_CHAR` (*) `MPI_CHARACTER` (*)	char unsigned char	—
`MPI_PACKED` (*)	—	—

Table 1.1: MPI data types. (*) – means that this feature is not supported on APE

The internal *FIFO* of the apeNEXT network hardware has space for a total of `MPI_FIFO_SIZE`= 64 words, with a wordlength of 128 bits. Since, each of the supported data typs occupies one word of 128 bits, the limitation of `MPI_FIFO_SIZE`= 64 elements is independent of the data type.

1.5 Send/receive operations

To make send/receive operation we provide to the user several functions:

- Send-receive operation:

  `MPI_Sendrecv()`: Basic function to perform homogeneous communications. We strongly recommend to use this function for data transmission. See description (2.1.17) for details.

`MPI_Sendrecv_replace():` Macro to `MPI_Sendrecv()`.

- Send and receive separately:

  `MPI_Send():` Macro for `MPI_BSend()`.

  `MPI_BSend():` Send in buffered mode. See description (2.1.13) for details.

  `MPI_Recv():` Receive in buffered mode. See description (2.1.14) for details.

- APE specific functions:

  `MPI_APE_Send():` Allows to send data in non-homogeneous mode between nearest neighbors. See description (2.1.15) for details.

  `MPI_APE_Send():` See description (2.1.16) for details.

Function `MPI_BSend()`[4] allows user to send huge amounts of data and allows routing across up to three orthogonal links to all nodes on a $3*3*3$ cube, i.e. to any node at relative distance $(\Delta_x,\ \Delta_y,\ \Delta_z)$ with $|\Delta_i| \leq 1$. The communication must be homogeneous. The user must specify a unique `tag` (assertion that tag not equal to `MPI_ANY_TAG`). This operations stores the data to the local memory, until `MPI_Recv()` is called with the corresponding `tag`.

When the user calls `MPI_Recv()`, the `tag` argument is checked. If tag equal to `MPI_ANY_TAG`, function finds first not empty buffer, otherwise, the function finds the buffer with the same tag. If such buffer not found, function return `MPI_ERROR_TAG`. Then the function starts a send-receive operation from the buffer in the local memory to the destination.

[4]The same for `MPI_Send()`.

2 Implemented functions

2.1 Standard MPI functions

Most functions have an assertion that the communicator equals to `MPI_COMM_WORLD`.

2.1.1 MPI_Init

`void MPI_Init(int *argc, char *argv[])`

All MPI programs must contain a call to `MPI_Init()`; before any other MPI routine is called. It must be called at most once; subsequent calls are erroneous. The version for ANSI C accepts the `argc` and `argv` that are provided by the arguments to the main function:

Internals:

- checks topology,
- creates map of neighbors,
- marks local buffers as empty,
- initialize MPI mechanism `MPI_Initialized()`.

APE-Limitations: `MPI_Init()` currently ignores the arguments.

2.1.2 MPI_Initialized

`int MPI_Initialized(int *flag)`

This routine may be used to determine whether `MPI_Init()` has been called. It is the only routine that may be called before `MPI_Init()` is called.

Internals: Checks global variable.

APE-Limitations: None.

2.1.3 MPI_Finalize

`void MPI_Finalize(void)`

This routines cleans up all MPI states. Once this routine is called, no MPI routine (even `MPI_Init()`) may be called. The user must ensure that all pending communications involving a given process complete before that process calls `MPI_Finalize`.

Internals: Doing nothing.

APE-Limitations: None.

2.1.4 MPI_Abort

`void MPI_Abort(MPI_Comm comm, int errno)` This routine makes a 'best attempt' to abort all tasks in the group of `comm`. This function does not require that the invoking environment takes any action with the error code.

Internals: Output `errno` and exit. As it is allowed by the MPI standardm function ignores `comm` parameter.

APE-Limitations: None.

2.1.5 MPI_Comm_size

`void MPI_Comm_size(MPI_Comm comm, int *size)`

Get the number of processes in the group of `comm`.

Internals: Returns `MPI_block_size`, calculated by the internal library function `_MPI_Topology()`, called from `MPI_Init()`.

APE-Limitations: Assertion that `comm` is `MPI_COMM_WORLD`.

2.1.6 MPI_Comm_rank

`void MPI_Comm_rank(MPI_Comm comm, int *size)`

Indicates the rank of the process that calls it in the range `0, 1, 2, ..., (size-1)`, where size is the return value of `MPI_Comm_size()`.

Internals: Returns the value of the OS-initialized variable `node_abs_id`.

APE-Limitations: Assertion that `comm` is `MPI_COMM_WORLD`.

2.1.7 MPI_Get_processor_name

`void MPI_Get_processor_name(char *name, int *len)`

This routine returns the name of the processor on which it was called. The name is a character string for maximum flexibility. It's value must allow to identify a specific piece of hardware; possible values include "`processor 9 in rack 4 of mpp.cs.org`" and "`231`" (where 231 is the actual processor number in the running homogeneous system). The argument name must represent storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long. `MPI_Get_processor_name` may write up to this many characters into name.

Internals: Returns "`node_XXX`", where "XXX" ≡ `node_abs_id`.

APE-Limitations: From "`node_XXX`" it's not possible to identify the system of the machine ("slice") on which program is executed.

2.1.8 MPI_Barrier

`void MPI_Barrier(MPI_Comm comm)`

Blocks the caller until all group members have called it. The call returns on all processes after all group members have entered the call.

Internals: The global tree of the apeNEXT root-loginc network is used to implement a barrier by executing instruction `if(all())`.

APE-Limitations: Assertion that `comm` is `MPI_COMM_WORLD`.

2.1.9 MPI_Wtime

`double MPI_Wtime()`

Returns a floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The 'time in the past' is guaranteed not to change during the life of the process. The user is responsible for converting large numbers of seconds to other units if they are preferred.

Internals: Access the configuration register, storing number of run-mode clock cycles, multiply this value by clock frequency.

APE-Limitations: Does not count the time for which the machine is in *12C-mode* (e.g. during system service via OS1).

2.1.10 MPI_BCast

`int MPI_Bcast(const void* in, const int sendcount, const MPI_Datatype dtype, int root, MPI_Comm comm)`

Broadcasts a message from the process with rank `root` to all processes of the group, including the `root` processor itself. It is called by all members of group using the same arguments for `comm`, `root`. On return, the contents of root's communication buffer has been copied to all processes.

Internals: Send the data from `root` node to all other nodes by repeating 1-step communications along the X, Y and Z directions.

APE-Limitations: Assertion that `comm` is `MPI_COMM_WORLD`.

2.1.11 MPI_Allreduce

`int MPI_Allreduce(const void* sendbuf, void* recvbuf, const int count, const MPI_Datatype dtype, const MPI_Op op, MPI_Comm comm)`

Combines the elements provided in the input buffer of each process in the group, using the operation `op`, and returns the combined value in the output buffer on all processors. The input buffer is defined by the arguments `sendbuf`, `count` and `dtype`; the output buffer is defined by the arguments `recvbuf`, `count` and `dtype`; (`count` and `dtype` specify the number of elements and their size for both buffers). The routine is called by all group members using the same arguments for `count`, `dtype`, `op` and `comm`. Thus, all processes provide input buffers and output buffers of the same length, with elements of the same type. Each process can provide one element, or a sequence of elements, in which case the combine operation is executed element-wise on each entry of the sequence. For example, if the operation is `MPI_MAX` and the send buffer contains two elements that are floating point numbers (`count = 2` and `dtype = MPI_FLOAT`), then `recvbuf(1) = global max ( sendbuf(1) )` and `recvbuf(2) = global max ( sendbuf(2) )`.

Internals: First, all nodes exchange data, making operation `Op` on them. Then, if operation is `MPI_SUM` or `MPI_PROD` and operands contains floating point numbers, `default` node (0, 0, 0) broadcasts result to all nodes, to ensure that all nodes have the same result of operation (because it may depend on the operand order). `MPI_MIN` and `MPI_MAX` is not supported for `MPI_COMPLEX` data, as it described in the MPI standard.

APE-Limitations:

- Assertion that `group` is `MPI_COMM_WORLD`.
- Function supports only `MPI_MIN`, `MPI_MAX`, `MPI_SUM` and `MPI_PROD` operations.

2.1.12 MPI_Send

`MPI_Send(void *data, int count, MPI_Datatype dtype, int dest,`
`int tag, MPI_Comm comm)` Send some data though the network.

Internals: Macro to `MPI_BSend()`

APE-Limitations: See `MPI_BSend()`.

2.1.13 MPI_BSend

`int MPI_BSend(void *data, int count, MPI_Datatype dtype,`
`int dest, int tag, MPI_Comm comm)`

A buffered mode send operation can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. However, unlike the standard send, this operation is local, and its completion does not depend on the occurrence of a matching receive. Thus, if a send is executed and no matching receive is posted, then MPI must buffer the outgoing message, so as to allow the send call to complete. An error will occur if there is insufficient buffer space.

Internals: Function copy user data into `MPI_LOCAL_BUFFER` and store it here, until `MPI_Recv()` function will be called with the corresponding `tag`. If you will try to send several data will the same `tag` nobody check, but if you will start to `MPI_Recv()` you may get an exception, thanks to that fact that data will be mixture and you will try to receive another one.

APE-Limitations:

- User must specify unique `tag`. `MPI_ANY_TAG` is prohibited.
- Operation must be `homogeneous`. So, it's possible to have 2- and 3-Step communications in different dimensions.
- Now operation using static buffer, except static ATTACH/DETACH mechanism. So, only `MPI_LOCAL_BUFFERS_COUNT` buffers available. Maximum size of massage `MPI_LOCAL_BUFFERS_LEN`. If there are no free space in buffer, function returns `MPI_ERR_TAG`.
- Function ignores argument `dest`. Since only the corresponding receive operation will trigger the communication.
- Assertion that `group` is `MPI_COMM_WORLD`.

2.1.14 MPI_Recv

`int MPI_Recv(void *recvbuf, int count, MPI_Datatype dtype,`
`int source, int tag, MPI_Comm comm)`

The receive buffer (`recvbuf`) must provide storage for `count` consecutive elements of the type specified by `dtype`, starting at address `recvbuf`. The length of the received message must be less than or equal to the length of the receive buffer. An overflow error occurs if all incoming data does not fit, without truncation, into the receive buffer.

Internals: Buffered receive. The function initiates the communication after checking the parameter `tag`: if it is equal to `MPI_ANY_TAG` function gets first message from buffer it founds. Otherwise, the function tries to find the first message in the buffer with the same `tag`.

APE-Limitations:

- Assertion that `comm` is `MPI_COMM_WORLD`.
- Only 1-, 2- and 3-step communications are possible.
- Message must be smaller then `MPI_LOCAL_BUFFER_LEN`.
- Operations must be homogeneous, therefore it's better to use `MPI_Sendrecv()`.

2.1.15 MPI_APE_Send

`int MPI_APE_Send(void *data, int count, int dest)`

Start immediately send operation to destination node. Function allows user to send data in non-homogeneous mode between two nodes that are directly connected (i.e. nearest neighbors).

Internals: Issues only one MTQ/RTQ (tx only). Function checks that `dest` is nearest neighbor and performs `prefetch()`. The function has less arguments than `MPI_Send()`) because they are irrelevant for `MPI_APE_Send()`.

APE-Limitations:

- Only 1-step communications.
- Message length must be Size $\leq$ `MPI_FIFO_SIZE` elements.

2.1.16 MPI_APE_Recv

`int MPI_APE_Recv(void *recvbuf, int count, int source)`

Starts immediately receive operation from given `source` node.

Internals: Generates RTQ (*rx* only) and QTR instructions. The function checks if `dest` is nearest neighbor. The function has less arguments than `MPI_Send()`) because they are irrelevant for `MPI_APE_Send()`.

APE-Limitations:

- Only 1-Step communications are possible.
- Message length must be $\leq$ `MPI_FIFO_SIZE` elements.

2.1.17 MPI_Sendrecv

`int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype send type, int dest, int sendtag, void *recvbuf, int recvcount, MPI_Datatype recvtype, int source, MPI_Datatype recvtag, MPI_Comm comm, MPI_Status *status)`

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes.

Internals: Function checks that distance from `source` to current node and from current node to `dest` node are the same. Then starts the communication. Warning: This function does not check that the distance is the same on all nodes, and that communication is actually homogeneous.

APE-Limitations:

- Assertion that `group` is `MPI_COMM_WORLD`.
- Only homogeneous operations allowed.
- 1-, 2- and 3- step communications are allowed.
- The distance from `source` to current node and from current to `dest` must be the same on all nodes.
- Functions doesn't take care about `tag` and `status` arguments.

2.1.18 MPI_Sendrecv_replace

`int MPI_Sendrecv_replace(void *buf, int count, MPI_Datatype datatype, int dest, int sendtag, int source, int recvtag, MPI_Comm comm, MPI_Status *status)`

Execute a blocking send and receive. The same buffer is used both for the send and for the receive, so that the message sent is replaced by the message received. The semantics of a send-receive operation is what would be obtained if the caller forked two concurrent threads, one to execute the send, and one to execute the receive, followed by a join of these two threads.

Internals: Macro to `MPI_Sendrecv()`.

APE-Limitations: The same like `MPI_Sendrecv()`.

2.2 Non standard MPI functions

We also implement several non-standard function to allow users to write optimal send-receive requests without caring about hardware details.

All this functions have no limitations and are based on global variables like `mpi_block_size_x` and `mpi_block_number_x`, generated by the `_MPI_Topology()` function.

2.2.1 MPI_GetNodeAbsX

`int MPI_GetNodeAbsX(int node_id)`
Returns the X-coordinate of the node with rank `node_id`.

2.2.2 MPI_GetNodeAbsY

`int MPI_GetNodeAbsY(int node_id)`
Returns the Y-coordinate of the node with rank `node_id`.

2.2.3 MPI_GetNodeAbsZ

`int MPI_GetNodeAbsZ(int node_id)`
Returns the Z-coordinate of the node with rank `node_id`.

2.2.4 MPI_GetNextNodeX

`int MPI_GetNextNodeX(int node_id)`
Returns the rank of the neighbor in direction $X+$ of the node with rank `node_id`. For example, if you have a machine with topology $4x4x4$ command
`int MPI_GetNextNodeX(0)` will return 1
`int MPI_GetNextNodeX(1)` will return 2
`int MPI_GetNextNodeX(2)` will return 3
`int MPI_GetNextNodeX(3)` will return 0

2.2.5 MPI_GetPrevNodeX

`int MPI_GetPrevNodeX(int node_id)`
Returns the rank of the neighbor in direction $X-$ of the node with rank `node_id`.

2.2.6 MPI_GetNextNodeY

`int MPI_GetNextNodeY(int node_id)`
Returns the rank of the neighbor in direction $X-$ of the node with rank `node_id`.

2.2.7 MPI_GetPrevNodeY

`int MPI_GetPrevNodeY(int node_id)`

Returns the rank of the neighbor in direction $Y-$ of the node with rank `node_id`.

2.2.8 MPI_GetNextNodeZ

`int MPI_GetNextNodeZ(int node_id)`

Returns the rank of the neighbor in direction $Z+$ of the node with rank `node_id`.

2.2.9 MPI_GetPrevNodeZ

`int MPI_GetPrevNodeZ(int node_id)`

Returns the rank of the neighbor in direction $Z-$ of the node with rank `node_id`.

2.3 Library internal functions

A library contains some internal functions that run are called automatically from `MPI_Init()`. The user is not supposed to execute them. But, we describe them here, because they are mentioned in the description of some other functions.

2.3.1 _MPI_Topology

Check machine topology (divided into block or not) and generate global variables, used by other functions.

2.3.2 _MPI_Neighbours

Generate `map[]` of neighbors for each node. It is used to check if it is possible to perform send-receive operations or not, and to determine the actual (relative) distances, which are used by the communication hardware.

3 Appendix

3.1 MPI communication modes

The MPI-1.1 standard foresees the following communication modes:

- Standard communication mode:

  The MPI Standard specify that the standard communication modde must be implemented either as *buffered* or *ready* mode (i.e. the choice between the two is implementation dependent). Therefore the standard send can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. Successful completion of a send *may* depend on the occurrence of a matching receive.

- Buffered mode:

  Send can be started whether or not a matching receive has been posted. It may complete before a matching receive is posted. Completion does *not* depend on matching receive.

- Synchronous mode:

  Send can be started whether or not a matching receive was posted. However, he send will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message sent by the synchronous send.

- Ready communication mode:

  Send may be started *only* if the matching receive is already posted. Otherwise the outcome is undefined.

Bibliography

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," 1995.

[2] APE Zeuthen Web-page, www-zeuthen.desy.de/ape

[3] D. Sokolov, "Implementation MPI for apeNEXT", Summer Student Report, DESY Zeuthen 2004