Implementation of MPI for apeNEXT

Dmitry Sokolov
Moscow State University
sokolov@forecsys.ru

Our aim was to provide a high-level message passing protocol to make migration to APE more easy. We generally follow the MPI-1.1 Standard. However, since the MPI interface assumes a very general communication model, it is very hard to support the full MPI functionality on a special architecture like apeNEXT. Some functions would become prohibitively inefficient. So, we implemented a set of the major functions to allow users a more easy use of the power of apeNEXT using the MPI mechanism.

1 INTRODUCTION

The APE (Array Processor Experiment) project was started in the mid eighties by the Istituto Nazionale di Fisica Nucleare (INFN) with the aim of developing massively parallel computers optimized for LQCD (Lattice QCD) simulations in theoretical particle physics. Within the framework of an European collaboration with DESY and the University of Paris Sud, currently a new generation of LQCD machines, apeNEXT, is being developed.

The Message Passing Interface (MPI), is a widely used standard for writing parallel programs following the message-passing communication model. The interface should establish a practical, portable, and flexible standard for message passing.

To make code migration from other supercomputers to APE easier we implemented a subset of the MPI library for apeNEXT. Our aim was not to implement the full MPI-1.1 standard, but rather to provide some key-functionality, to allow the user to run typical numerical code using MPI on APE.

The user should, however, be aware that due to the generic purpose of the MPI interface, it is not possible to make full use of the specialized hardware features of apeNEXT. Using MPI on APE will therefore typically not be the most efficient way of parallel programming.

2 APE PROJECT

APE is one of several projects in the theoretical physics community that have developed massively parallel, high-performance computer architectures. The driving force why physicists develop and build computers by themselves is the success of numerical simulations in understanding the interactions of elementary particles, in particular their strong interactions described by quantum chromodynamics (QCD) (see [2] for details). The formulation of QCD which can be simulated on computers is called Lattice QCD (LQCD).

These simulations require a lot of computer power which can only be provided by massively parallel computers. To keep costs low it is (still) useful to custom design special purpose computers like apeNEXT.

2.1 The Family of APE Machines

The evolution over more than one decade of APE systems is briefly recollected in Table 1.

The first generation of APE computers dates to the mid eighties. APE100, the second generation of APE supercomputers, had been the leading workhorse of the European lattice community since the middle of the 1990s.

Commissioning of APEmille, the third generation of APE systems, started in the year 2000. These machines make currently 2 TFlops of computing power available to the LGT community.

APEmille machines are installed at several sites all over Europe, as detailed in table 2. They provide a very stable and reliable computing environment, with typical up-times of the order of 85 %.

In order to keep up with future and growing

	APE 1 1988	APE100 1993	APEmille 1999	apeNEXT 2003
Architecture	SIMD	SIMD	SIMD	SPMD
# nodes	16	2048	2048	4096
Topology	flexible 1D	rigid 3D	flexible 3D	flexible 3D
Memory per CPU	256 MB total	4-64 MB	32 MB	256-1024 MB
# registers (width)	64 (32 bit)	128 (32 bit)	512 (32 bit)	512 (64 bit)
clock speed	8 MHz	25 MHz	66 MHz	200 MHz
Total Computing Power of all	1.5 GFlops	250 GFlops	2 TFlops	8-20 TFlops

Tab. 1: *Key parameter comparison of the APEfamily of supercomputer.*

requirements, the development of a new generation of a multi-TFlops computer for LGT, apeNEXT, is in progress.

2.2 *apeNEXT Processor and Global Design*

For apeNEXT all processor functionalities, including the network devices, are integrated into one single custom chip running at a clock frequency about 200 MHz. Unlike in the former APE machines, the nodes run asynchronously.

This gives rise for two new key features of apeNEXT. First, apeNEXT follows the single program multiple data (SPMD) programming model (as opposed to SIMD). Each processing node is a fully independent processor, with a full-fledged flow-control unit and, of course, a number-crunching unit. The node has access to its own memory bank, where both program and data are stored. It executes its own copy of the program at its own pace. Nodes need to be synchronized only when a data-exchange operation is performed. This architecture may be labeled as a distributed-memory parallel computer, in which nodes exchange data through some sort

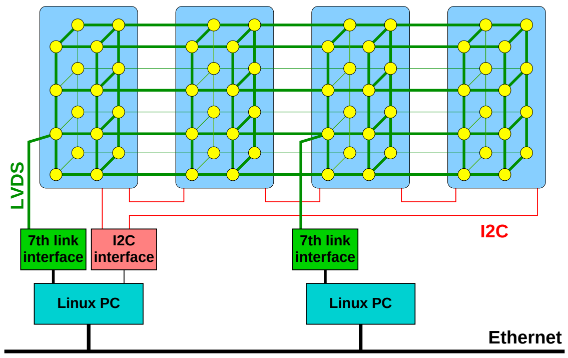

Fig. 1: *A possible apeNEXT configuration with 4 boards, 2 external LVDS-links for I/O, and a chained I2C-link for slow-control.*

of “message-passing” scheme.

The latency associated to a “message” is extremely short, of the order of 2 to 3 times the latency associated to an access to local memory.

A second important architectural enhancement lays on the possibility of routing all read memory accesses (to local or remote nodes) through a receiving queue, which can be later accessed by the processor with zero latency.

The complete processing element is contained in just one custom-designed integrated circuit, connected to a memory bank of 256–1024 MBytes with Double Data Rate (DDR) Dynamic RAM chips.

The block diagram of the processor chip is shown in Fig. 2. It is a 64-bit architecture, optimized for floating point performance.

We would like to highlight some selected details of the processor shown in Fig. 2:

- A large register file of 256 registers each containing a pair of 64 bit words. All operands for the arithmetic unit arrive from

Bielefeld	130 GFlops	(2 crates)
Zeuthen	520 GFlops	(8 crates)
Milan	130 GFlops	(2 crates)
Bari	65 GFlops	(1 crates)
Trento	65 GFlops	(1 crates)
Pisa	325 GFlops	(5 crates)
Rome 1	520 GFlops	(8 crates)
Rome 2	130 GFlops	(2 crates)
Orsay	16 GFlops	(1/4 crates)
Swansea	65 GFlops	(1 crates)

Tab. 2: *The APEmille installations.*

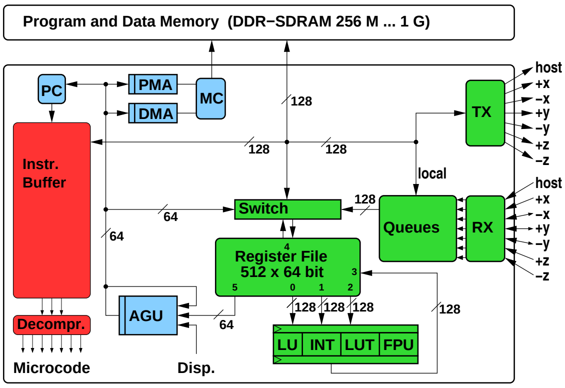

Fig. 2: *Schematics of the apeNEXT processor.*

the register file and all results are written back here.

- An arithmetic unit which performs floating point as well as integer operations. All floating-point data is represented in the 64-bit double-precision format of the IEEE standard.
- An address-generation unit (AGU) which computes addresses for memory access independently and concurrently with the main arithmetic unit.
- A memory controller (MC) supporting a memory bank of 256–1024 MBytes based on standard DDR-SDRAM. The memory is used to store both data and program instructions.
- A flow-control unit that executes programs specified as a sequence of compiler prepared microcode words, using the VLIW control style.
- A network interface which contains seven LVDS link interfaces. Each link is bi-directional allowing send and receive operations to run concurrently.
- A slow serial interface based on the I2C standard, used for system initialization, debugging and exception handling.

2.3 apeNEXT Network

We have a three-dimensional torus of processors (see Fig. 3). Although all nodes are connected to their nearest neighbors only, the hardware allows routing 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 $|\Delta_i| \le 1$.

Typically we use the SIMD operation model, therefore the communications are *homogeneous*, i.e., each node $(x,\ y,\ z)$ communicate with the corresponded $(x+\Delta_x,\ y+\Delta_y,\ z+\Delta_z)$ with $|\Delta_i| \le 1$.

Since apeNEXT also allows non-SIMD operation (due to the possible asynchronous operation of each node), we can also use non-homogeneous communications. Non-homogeneous communications are restricted to the rule, that from any node with coordinates $(x,\ y,\ z)$ one can send data only to the node with coordinates $(x+\Delta_x,\ y+\Delta_y,\ z+\Delta_z)$ with $(|\Delta_x|+|\Delta_y|+|\Delta_z|) \le 1$. For instance, we can send data only from the node with coordinates (1, 1, 1) to the node (1, 1, 2). The first one must sends this data into $Z+$ direction and the second node reads it from $Z-$ direction. Note, that the receiver on node (1, 1, 2) needs to be activated explicitly by instructions executed on this node.

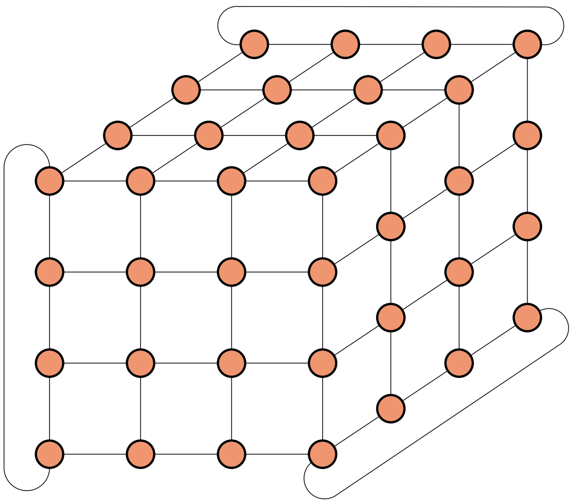

Fig. 3: *3D-Torus. Each node connected with 6 neighbors. Some communicational links not showed.*

For homogeneous communications, the hardware supports direct data transfer along 2 (or 3) links, e.g. through the path shown in Fig. 4. The directions of these 2 (or 3) links have to be orthogonal. One possible communication could be from each node $(x,\ y,\ z)$ to $(x,\ y+1,\ z+1)$.

In practice, these restrictions are not a problem, because in LQCD equations the nodes often need to communicate only with their nearest neighbors.

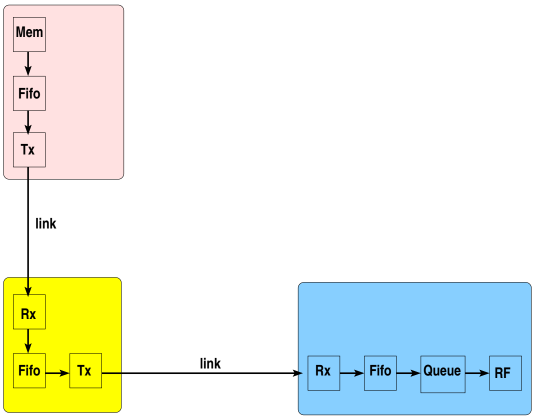

Fig. 4: *Two nodes can transmit data using homogeneous communications.*

2.4 Data types on apeNEXT

Since all data types on apeNEXT have a size of either 64 or 128 bits, we can restrict ourself to support the following data types:

- integer
- unsigned integer
- double
- complex

Most MPI data types can be mapped to these types according to Table. 3. Our implementation does not support data types `MPI_PACKED` and the family of `MPI_CHAR`, because they are not relevant for LQCD simulation codes.

2.5 *"Hello world!"* on apeNEXT

It's time to write a *"Hello world!"* program on APE. Our program will be executed on a supercomputer optimized for Lattice QCD, so, we will also create a small array, fill it with some digits and transmit them in $X-$ direction. The program does not require a specific topology, so it can run on any APE configuration. The standard `printf()` command would output data only from the *default node*, therefore, we will use the apeNEXT specific function `mputi()` for output. As arguments `mputi()` takes a *slice* of nodes which should write their data. The command `mputi(0, 0, 0, i)` will print the integer value `i` from node (0, 0, 0), command `mputi(0, 0, 1, i)` will

MPI name	C equivalent

`MPI_INT`
`MPI_LONG`
`MPI_SHORT`
`MPI_BYTE`
`MPI_LOGICAL`
	integer,
	long,
	short int

`MPI_UNSIGNED`
`MPI_UNSIGNED_SHORT`
`MPI_UNSIGNED_LONG`
	unsigned integer,
	unsigned short int,
	unsigned long,

`MPI_DOUBLE`
`MPI_FLOAT`
`MPI_LONG_DOUBLE`
`MPI_DOUBLE_PRECISION`
	double,
	float,
	long double

`MPI_COMPLEX` `MPI_DOUBLE_COMPLEX`	
`MPI_CHAR` (*) `MPI_UNSIGNED_CHAR` (*) `MPI_CHARACTER` (*)	char unsigned char
`MPI_PACKED` (*)	

Tab. 3: *MPI data types. (*) – means that this feature is not supported by MPI on APE*

do the same from node (0, 0, 1), command `mputi(0, 0, 0, 1, 1, 0, i)` will generate output from nodes (0, 0, 0), (0, 1, 0), (1, 0, 0) and (1, 1, 0).

In our program we will use global variables defined by the operating system, like `machine_size_x` which stores the number of nodes in X dimension and `node_abs_id`, which contains a unique *node id*.

```
// Example 1: Hello world!
#include <stdio.h>
#include <queue.h>
#include <sysvars.h>
#define MAX 4 // length of array

int main()
{
   int i;
   double a[MAX];
   // This wil be produced only by
   // default node
   printf("Hello world!\n");

   // Initialization of array
   for (i=0; i<MAX; i++)
        a[i] = node_abs_id + i;

   // Sending array in X+
   for (i=0; i<MAX; i++) {
        prefetch(a[i+X_PLUS]);
        fetch(a[i]);
```

```
    }
    // Output of array
    for (i=0; i<MAX; i++) {
         printf("a[\%d]\n", i);
         mputd(0, 0, 0,
             machine_size_x - 1,
             machine_size_y - 1,
             machine_size_z - 1,
             a[i]);
     }
     return 0;
}
```

In this code we declare two variables (array of double `a[]` and integer `i`). On each node of apeNEXT an instantiation of these variables will be created in the local memory, but of course on each node these variables can hold a different value. When elements of an array are accessed with the *magic offset* `X_PLUS` in the index, the CPU of each node recognizes that it must take the value of `a[]` not from its local memory, but from the memory of the corresponding remote node. The memory read access can be split into two parts by using the macros `prefetch()` and `fetch()`:

`prefetch(a[i+X_PLUS])`: fetches the value of `a[i]` from the local memory and sends it to $X-$ and it also starts receiving data from the remote node in $X+$ direction. This data is automatically stored in a receive queue.

`fetch(a[i])`: loads the data from the receive queue into the register file if the receive operation is completed, otherwise the processor has to wait.

At software level no further instructions are needed, the network operation is automatically controlled and executed by the hardware.

2.6 Software and Application Benchmarks

For apeNEXT both a TAO and a C compiler is provided. The latter is based on the freely available *lcc* compiler and supports most of the ANSI 89 standard with a few language extensions required for a parallel machine [3]. Both compilers generate a high-level assembly. An assembler pre-processor (*mpp*) is used to translated this into a low-level assembly. For machine specific optimizations at this assembly level, e.g. address arithmetics and register move operations, the software package *sofan* is under development. Finally, the microcode generator (*shaker*) optimizes instruction scheduling, which for APE machines is completely done in software (see Fig. 5).

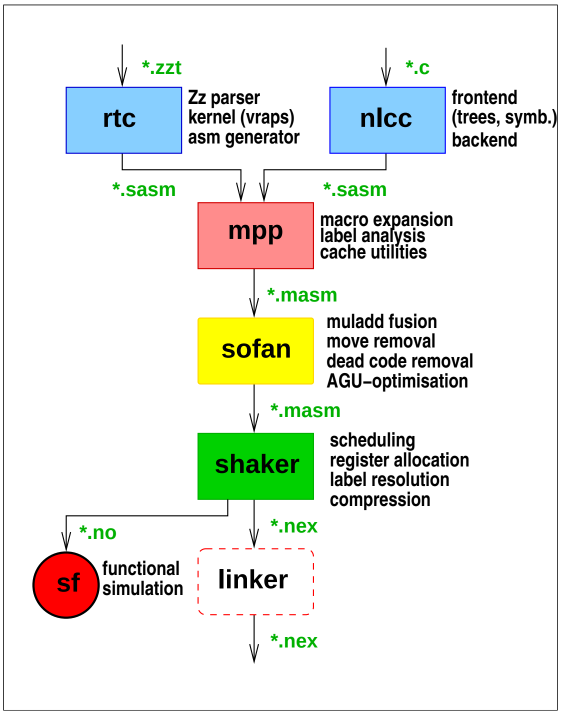

Fig. 5: *An overview on the compilation procedure.*

In QCD simulations most of the time is spent applying the discretized Dirac operator, i.e. a matrix-vector multiplication (see. [2]). This operation involves remote communications and therefore depends on the number of processors used. The maximum number of processors is limited by the size of the problem, i.e. the lattice volume. Considering the worst case where the problem is distributed over the maximum number of processors, a sustained performance of 56% has been found. This eventually indicates an excellent scaling behavior of the apeNEXT architecture.

3 MPI INTERFACE

3.1 What is MPI?

MPI was an effort to produce a message-passing interface standard across the whole parallel processing community. Sixty people representing forty different organizations – users and vendors of parallel systems from both the US and Europe – collectively formed the "MPI Forum". After two years of discussion they released a document specifying a standard Message Passing Interface (MPI).

MPI provides source-code portability of message-passing programs written on C or Fortran across a variety of architectures.

MPI name	Function
`MPI_SUM`	Sum
`MPI_PROD`	Production
`MPI_MIN`	Minimum
`MPI_MAX`	Maximin
`MPI_LAND`(*)	Logical AND
`MPI_BAND`(*)	Bitwise AND
`MPI_LOR` (*)	Logical OR
`MPI_BXOR`(*)	Bitwise OR
`MPI_LXOR`(*)	Logical XOR
`MPI_BXOR`(*)	Bitwise XOR
`MPI_MAXLOC`(*)	Max. & location
`MPI_MINLOC`(*)	Min. & location

Tab. 4: *MPI operations used by `MPI_Allreduce()`. (*) – is not supported.*

3.2 MPI Communication modes

MPI comprises a library. An MPI process consists of a C or Fortran 77 program which communicates with other MPI processes by calling MPI routines.

MPI supports four communication modes:

Standard mode: Must be mapped to *buffer* or *ready*.

Buffered mode: Send operations store data to a memory buffer and communication will be finished once the receive function has completed.

Synchronous mode: Sender waits for posting from receiver and exits from subroutine only after the receiver has received the full message.

Ready mode: Sender immediately starts to transfer data, but the operation is guaranteed to complete correctly only if the receiver is ready to receive data, otherwise the result of the operation is undefined.

Each message in MPI must have a *tag* that allows the receiver to take any message with a matching tag.

4 IMPLEMENTATION DETAILS

4.1 APE-Limitations for MPI

As 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. We therefore decided to introduce the following restrictions:

1. MPI allows users to divide the whole machine into processing groups, called *communication groups* identified by a *communicator*. Users can execute an operation on all nodes of one communication group. We prohibit this – only `MPI_COMM_WORLD` communicator can be used, that means that always the whole machine partition takes part in group operations.

2. apeNEXT doesn't have any system services that run independently from the main program and could, e.g., interrupt execution of user programs. Therefore we don't implement *error handlers*. Users can't cancel network operations.

3. Since on apeNEXT data transfer is handled by the network hardware in a completely independent and automatic way (after a communication request has been issued), all send operation are always non blocking. Special *non-blocking operations* like `MPI_ISend()`, `MPI_IBsend()`, `MPI_ISSend()` and `MPI_IRecv()` are not supported.

4. The apeNEXT network has a 3D-torus topology and non-homogeneous communication are allowed only if sender and receiver are connected by a direct link. For homogeneous network operations (when all nodes execute the same instructions, sending/receiving data in/from the same direction) the hardware allows routing across up to three orthogonal links to all nodes on a cube, i.e. connecting nodes at distance $(\Delta_x, \Delta_y, \Delta_z)$ with $|\Delta_i| \leq 1$. Other communications (like sending data from node (0, 0, 0) to node (2, 0, 0) are not allowed.

5. There is no posting mechanism implemented, *synchronous mode* and *ready mode* can't be supported.

MPI name	Description
MPI_SUCCESS	Successful return code
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_UNKNOWN	Unknown error
MPI_COMM_WORLD	All nodes involved into operation
MPI_COMM_SELF	Only local node involved
MPI_MAX_PROCESSOR_NAME	Len of buffer for storing node name
MPI_ANY_TAG	Don't care about TAG of message
MPI_PROC_NULL	Empty source/destination
MPI_VERSION	Major version, (=1)
MPI_SUBVERSION	Minor version, (=1)

Tab. 5: *Main MPI constants.*

6. As mentioned in section "Data types on apeNEXT" [2.4], we do not support `MPI_PACKED` and `MPI_CHAR` data types.

7. There are two send functions implemented: `MPI_Bsend()` and `MPI_APE_Send()`. Standard `MPI_Send()` defaults to `MPI_Bsend()`. `MPI_APE_Send()` provides an apeNEXT specific send operation.

Roughly speaking, we expect the user of our MPI implementation to use the SIMD programming model.

4.2 APE Mode Send/Receive

To allow users to make more efficient use of the network or to send data in non-homogeneous mode between two, directly connected nodes we provide two new functions:

`MPI_APE_Send(void *d, int cnt, int dst)`
: This function sends data `d` to the FIFO of hardware link for the corresponding direction and hardware starts to transmit it to `dst`. `MPI_APE_Send()` can finish before data was send.

`MPI_APE_Recv(void *d, int cnt, int src)`
: This initiates receive operation and stores data to memory. `MPI_APE_Recv()` operation will be finished only after successefull data transmission.

The length of the data is limited by the maximum message size, supported by the hardware, i.e. 64 words[1]. The user has to take care that the receive the queues are not been used between call of `MPI_APE_Send()` and `MPI_APE_Recv()`.

4.3 Implemented Functions

- MPI Standard functions implemented for APE:

`MPI_Init()` Initialize MPI system

`MPI_Initialized()` Checks, if MPI is initialized or not.

`MPI_Finalize()` Switch off MPI.

`MPI_Abort()` Immediately stop execution of program.

`MPI_Comm_size()` Get number of nodes.

`MPI_Comm_rank()` Get unique node id.

`MPI_Get_processor_name()` Get string containing name of node.

`MPI_Barrier()` Wait until all nodes go to this barrier.

`MPI_Wtime()` Get timer value to measure execution time.

`MPI_BCast()` Broadcast some data to all nodes from root node.

`MPI_Allreduce()` Makes group operation on all nodes, like sum or product.

`MPI_Send()` Send data from one node to another.

`MPI_Bsend()` Send data from one node to another in buffered mode.

`MPI_Recv()` Receive data in buffered mode.

[1] Size of each word is 128 bit.

`MPI_Sendrecv()` Send and receive data (shift operation).

`MPI_Sendrecv_replace()` Same like `MPI_Sendrecv()`.

- Non MPI Standard functions implemented for APE:

`MPI_APE_Send()` Send data to neighbor node in APE mode.

`MPI_APE_Recv()` Receive data in APE mode.

`MPI_GetNodeAbsX()` Returns X coordinate of node.

`MPI_GetNodeAbsY()` Returns Y coordinate of node.

`MPI_GetNodeAbsZ()` Returns Z coordinate of node.

`MPI_GetNextNodeX()` Returns *node id* of next node in $X+$ direction.

`MPI_GetPrevNodeX()` Returns *node id* of next node in $X-$ direction.

`MPI_GetNextNodeY()` Returns *node id* of next node in $Y+$ direction.

`MPI_GetPrevNodeY()` Returns *node id* of next node in $Y-$ direction.

`MPI_GetNextNodeZ()` Returns *node id* of next node in $Z+$ direction.

`MPI_GetPrevNodeZ()` Returns *node id* of next node in $Z-$ direction.

4.4 MPI Examples

Here is example of how to execute the `MPI_Allreduce()` function. We initialize an array on each node by some bulk data and make complete the global sum of it.

```
// Example 2: Using MPI_Allreduce()
#include "mpi.h"
#define MAX 5

int main(){
  int i, res, source, dest;
  int i_in[MAX], i_res[MAX];
  double time;

  MPI_Init(0, 0);
  time = MPI_Wtime();

  // Initialize array
  for (i=0; i<MAX; i++) {
    i_in[i] = i+node_abs_id;
    i_res[i] = 0;
  }

  // Calling Allreduce()
  res = MPI_Allreduce(i_in, i_res,
    MAX, MPI_INT, MPI_SUM,
    MPI_COMM_WORLD);

  // Output results
  printf("-- Allreduce(%d) --\n", res);
  for (i=0; i<MAX; i++) {
    printf("i[%d]=%d\n", i, i_res[i]);
    mfputi(0, 0, 0, machine_size_x-1,
      machine_size_y-1,
      machine_size_z-1, i_res[i]);
  }

  // Measure execution time
  time = MPI_Wtime() - time;
  printf("Seconds : %f.\n", time);

  MPI_Finalize();
  return 0;
}
```

In the next example we execute `MPI_Sendrecv()` to shift data in $X+$ direction.

```
// Example 3: Using MPI_Sendrecv()
#include "mpi.h"
#define MAX 5

int main(int *argc, char *argv[]){
  int i, rank, size, res;
  int source, dest;
  double i_in[MAX], i_res[MAX];
  double time;
  char c[MPI_MAX_PROCESSOR_NAME];
  MPI_Status status;

  MPI_Init(argc, argv);
  time = MPI_Wtime();

  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(c, &i);
  printf("Node: %s\n", c);

  // Initialize array
  for (i=0; i<MAX; i++) {
    i_in[i] = i * time;
    i_res[i] = 0;
  }

  // Calculating source and dest
  dest = GetNextNodeX(rank);
  source = GetPrevNodeX(rank);

  // Calling Sendrecv()
  res = MPI_Sendrecv(i_in, MAX,
```

```
    MPI_DOUBLE, dest, 0, i_in,
    MAX, MPI_DOUBLE, source, 0,
    MPI_COMM_WORLD, &status);

  // Output
  printf("-- Sendrecv(%d) --\n", res);
  for (i=0; i<MAX; i++) {
    printf("i[%d]=%d\n", i, i_res[i]);
    mputi(0, 0, 0, machine_size_x-1,
      machine_size_y-1,
      machine_size_z-1, i_in[i]);

    printf("r[%d]=%d\n", i, i_res[i]);
    mputi(0, 0, 0, machine_size_x-1,
      machine_size_y-1,
      machine_size_z-1, i_res[i]);
  }

  time = MPI_Wtime() - time;
  printf("Seconds : %f.\n", time);

  MPI_Finalize();
  return 0;
}
```

5 BENCHMARKS

To check the efficiency of MPI for apeNEXT we executed the following benchmarks:

global sum: Perform a global reduce operation (sum, max, min, product) across all the nodes using `MPI_Allreduce()`. For example if we have N nodes and each node k stores an array $a_k[i]$. After executing `MPI_Allreduce()` using `MPI_PROD`, each node will contain an array $b[i]$, where $b[i] = \prod_{j=0}^{N} a_j[i]$.

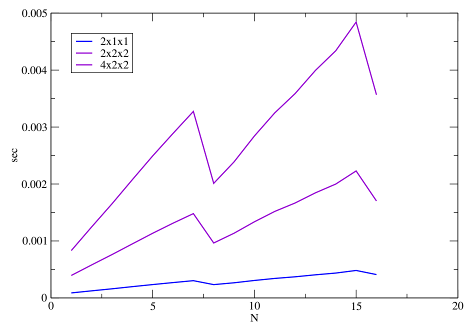

Fig. 6: *Bandwidth of `MPI_Allreduce()` depending from length of array. Each line corresponds to different number of processors involved into operation.*

ping-pong: Perform network operations where all nodes send data in one direction by executing `MPI_Send()` and receive data using `MPI_Recv()`, followed by a data transfer in the opposite direction.

sendrecv: Perform network operations where all nodes send and receive by calling `MPI_Sendrecv()`.

All benchmarks test the efficiency of the network (including memory access), *global sum* (see Fig. 6) also involves the mathematical unit of apeNEXT. We did run all benchmarks several times with different data length and compared bandwidths. small, MPI lose a lot of time for of memory operation, but increasing the

We executed our benchmark code on two prototype nodes of apeNEXT. The measured bandwidth is shown in Fig. 7. The processors were running at a clock frequency of 120 MHz. The gross bandwidth was therefore 120 MBytes per second. Due to the hardware protocol overhead we can expect to achieve only about 75% of the gross bandwidth, i.e. about 90 MBytes per second. As can been seen from Fig. 7 we get already close to this limit.

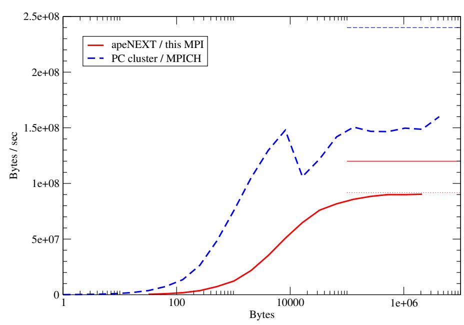

Fig. 7: *Bandwidth of `MPI_Sendrecv()` depending from length of array. The lower two horizontal lines show the maximum effective and gross network bandwidth for apeNEXT, the upper the maximum bandwidth of the Myrinett network.*

For comparison we also plotted the numbers obtained on a PC cluster of the John from Neumann Institute (NIC). This cluster has a Myrinett network for internode communication. For a comparison of both results some comments should be taken into account. The PC cluster was installed in 2001 and therefore does not provide up-to-date technology. On the other hand, the apeNEXT processors are not yet running at full speed. The expected final clock speed is 180-200 MHz, i.e. the bandwidth will increase by about 50-66%. The total bandwidth per node is up to 3 times larger, as an apeNEXT node can communicate through up to 3 bi-directional network links concurrently. The PC cluster provides only one link per node (i.e. two processors).

6 CONCLUSION AND OUTLOOK

My work in the APE project included several steps:

- Understanding the apeNEXT architecture: both software and hardware details.
- Learning the MPI Standard.
- Understanding and discussing, what parts of MPI can be implemented for apeNEXT.
- Implementation of MPI.
- Testing, debugging, optimizing functions.
- Making benchmarks. Comparing with other architectures.
- Writing student report, technical documentation for implementation and documentation for future developers of MPI for apeNEXT.

I have completed the implementation, debugging, testing, benchmarking and documentation of a relevant sub-set of the MPI Standard. Future development activities might add a few functions and certainly should spend further effort on optimization of the library to improve performance.

Working in APE Group was very interesting for me. I never before worked so close with supercomputers, which required to analyze assembler and microcode and to think of optimization on the hardware level. I executed my code on prototype hardware at DESY Zeuthen and at INFN in Ferrara. Discussing many implementation details, algorithms and hardware features with people from the APE group allowed me to apply and increase my knowledge in computer science. Writing my code I also found a number of bugs in the C-compiler(*nlcc*), which have been fixed on the fly. Thus I found my research work in the APE project very interesting and I extended my knowledge on computer architectures, processor design and supercomputers and on computer science at all.

7 ACKNOWLEDGMENTS

I want to thank all people from APE: Hubert, Dirk, Norbert, Max and Guillaume for sharing with me their research work and for the interesting discussions we had. I'd like to express a special gratitude to Hubert Simma and Dirk Pleiter for there patient to answer any of my questions, showing me small tricks in big supercomputer, having always creative ideas made my working time with you very interesting and enjoyable. And again my special thanks to Dirk for the guidance in writing the documentation, fixing bugs in my code, help with the benchmarks and for the good sense of humor in everyday work.

References

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

[2] F. Bodin et al., "The apeNEXT Project", CHEP'03, La Jolla, California, March 25-28, 2003

[3] C.W. Fraser, D.R. Hanson, D. Hansen, "A Retargetable C Compiler: Design and Implementation", 1995.

[4] D. Pleiter, D. Sokolov, "MPI for apeNEXT", DESY, 2004

[5] APE Group DEZY Zeuthen Web-site, http://www-zeuthen.desy.de/ape

[6] MPI Forum Community, http://www.mpi-forum.org

[7] apeNEXT: a Multi-TFlops Computer for Elementary Particle Physics, APE-Collaboration

[8] Neil MacDonald, Elspeth Minty, Tim Harding, Simon Brown, "Writing Message-Passing Parallel Programs with MPI", Course Notes, Edinburgh Parallel Computer Center, The University of Edinburgh