# Improving the reliability and the performance of CAPE by using MPI for data exchange on network

Van Long Tran[1], Eric Renault[1], and Viet Hai Ha[2]

[1] Institut Mines-Telecom – Telecom SudParis, France,
{van_long.tran,eric.renault}@telecom-sudparis.eu
[2] Hue University, Vietnam,
haviethai@gmail.com

**Abstract.** CAPE - stands for Checkpointing Aided Parallel Execution - has demonstrated as a high performance and compliant OpenMP implementation on distributed memory systems. CAPE is based on the use of checkpoints to automatically distribute jobs of OpenMP parallel structs to distant machines and to automatically collect calculated results on these machines to the master machine. However, on the current version, the data exchange on networks use manual sockets that take time to establish connections between machines for each parallel construct. Furthermore, this technique is not very reliable due to the risk of conflict of ports and the problems of data exchange using stream. This paper aims at presenting the approach of use MPI to improve the reliability and the performance of CAPE. We will analyze, discuss two methods and experiment to evaluate the performance of them.

**Keywords:** CAPE, OpenMP, MPI, high performance computing, parallel programming

## 1 Introduction

In order to explore further the capabilities of parallel computing architectures such as grid, cluster, multi-processors, multi-cores, an easy-to-use parallel language is one of important factors.

MPI [1] (Message Passing Interface) is well-known that it is a standard for developing parallel application in distributed-memory architecture. It provides essential point-to-point communication, collective operation, synchronization, virtual topologies, and other communication facilities for set of processes in a language-independent way, with a language-specific syntax, plus a small set of language-specific features... Although it is capable of providing high performance, it is difficult to use. MPI require programmers to explicitly divide the program into blocks, one for the master process and the others for slave processes. Moreover, some tasks, like sending and receiving data or the synchronization of processes, must be explicitly specified in the program.

OpenMP [2] has become the standard for the development of parallel applications on shared-memory architectures. It is composed of a set of very simple and powerful directives and functions to generate parallel programs in C, C++ or Fortran. From the programmer's point of view, OpenMP is easy to use as it allows to incrementally express parallelism in sequential programs, the programmer can start with a sequential version of a program and step by step add OpenMP directives to change it into parallel versions. Moreover, the level of abstraction provided by OpenMP makes the expression of parallelism more implicit where the programmer specifies what is desired rather than how to do it. This has to be compared to message-passing libraries, like Message Passing Interface (MPI) [1], where the programmer specifies how things must be done using explicit send/receive and synchronization calls.

Because of these advantages of OpenMP, there are some effort to run OpenMP program on distributed-memory systems. In which, CAPE [3][4] is a tool to compile and provide an environment to execute concurrently OpenMP programs. This solution archive two requirements that are fully compatible with OpenMP standard and high performance.

In order to automatic distribute jobs to slaves on the distributed-memory system, CAPE has been developed within the following idea: when reaching a parallel section, the master thread is dumped and its checkpoint are sent to slaves; then, each slave executes a different thread; at the end of the parallel section, each slave thread extracts and returns to the master thread the list of all modifications that has been locally performed; this master includes these modifications and resumes its execution.

In the current version of CAPE, the data exchange between nodes works as DICKPT checkpoints [3][5], and they are transferred on network by using manual sockets. It is really waste time to initialize, connect and listen in socket at the run time. In addition, this approach is weak in the reliably due to the difficulty to manage the data exchange on the network.

This paper aims at presenting the approach focusing on reducing the checkpoints transfer time and increase the reliably of data transfer of CAPE on networks. For organization, some related works and its advantages that use MPI to transfer data on networks are listed in section 2, section 3 discuss and analyze about current version of CAPE using manual socket. Section 4 proposes a new method that use MPI instead of manual sockets. Section 5 shows evaluation and some result of experimentation to compare these methods. At the end, in the section 6, some conclusions and future works are shown.

## 2 Related Works

Using MPI framework to transfer data between nodes on the network have been researched and applicated widely today. They have achieved high reliability, security, portability, integrity, availability and high-performance, etc. of the transferred data.

The first research to mention is the combination of MPI and OpenMP. In this case, the researchers use MPI framework to send data and code from master node to all working nodes on network. At the other side, on each working node, OpenMP framework is used to execute the task assigned in parallel. Finally, the results on working nodes are sent back to master node by using explicit MPI codes. Although this way takes time and efforts of programmers, it take advantages on performance and integrity. Some typical cases such as the [6] shown that it can achieve high efficiency and good scalable performance, the [7][8] shown a reduction of communication needs and memory consummation, or improving the load balance ability.

The are also a lot of researches that use the advantages of MPI assume the data exchange between accelerators on cluster. For example, GPU-Aware MPI [9] and CUDA Inter-process Communication [10] used standard MPI to support data communication form GPU to GPU on clusters. This techniques have demonstrated high-performance, portability of the system using MPI. In addition, on cloud, Cloud Cluster Communication [11] and ECC-MPICH2 [12] using a modified MPI framework had shown the validation of security in terms of authentication, confidentiality, portability, data integrity and availability.

The result above is very important for the development orientation of CAPE using MPI. In this paper we use MPI framework to transfer checkpoints between nodes on CAPE. In the future, MPI will contribute more important role as we continue to develop CAPE that supports GPU and cloud computing.

It is necessary to note that the use of MPI on CAPE presented in this paper is completely different with the combinations of MPI and OpenMP mentioned above and this use does not change the essence of CAPE. CAPE is based on the use of checkpointing technique to implement OpenMP on distributed systems. This implementation is totally compliant with OpenMP standards and programmers have not to modify their application program source codes. MPI take only the role of transferring data on the network. At the other sides, in most cases, the other combination of MPI and OpenMP require programmers to modify their source codes and as a consequence, can not provide a totally OpenMP compliant implementation.

## 3 CAPE using manual sockets for transferring data

### 3.1 System organization

In CAPE, each node consists in two processes. The first one runs the application program. The second one plays two different roles: first as a DICKPT checkpointer and second as a communicator between the nodes. As a checkpointer, it catches signals from the application process and executes appropriate handles to create the DICKPT checkpoint. In the communicator role, it ensures the distribution of jobs and the exchange off data between nodes. Figure 1 shows the principle of CAPE's organization.

In the current version, the master node is in charge of managing slave nodes and does not execute any application job in the parallel sections.

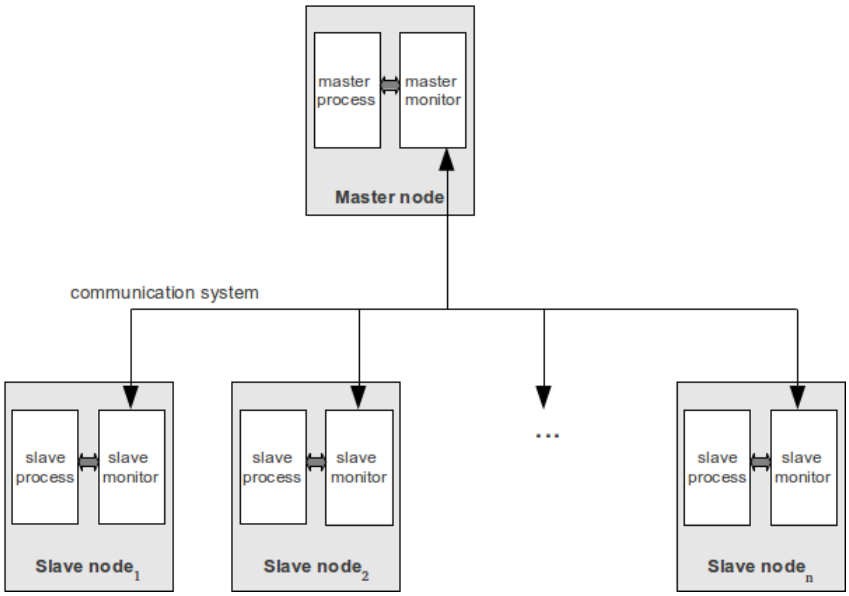



**Fig. 1.** System organization

### 3.2 Execution model

CAPE is the alternative approach to allow the execution of OpenMP programs on distributed-memory systems. CAPE is based on a process as a parallel unit, which is different from the traditional implementations of OpenMP where the parallel unit is a thread. All the important tasks of the fork-join model are automatically generated by CAPE based on checkpointing techniques, such as task division, reception of results, updating results into the main process, etc. In its first version, CAPE used complete checkpoints so as to prove the concept. However, as the size of complete checkpoints is very large, it takes a lot of traffic on the network to transfer data between processes and involves a high cost for the comparison of the data from the different complete checkpoints to extract the modifications. These factors have significantly reduced the performance and the scalability of our solution. Fortunately, these drawbacks have been overcome in the second version of CAPE based on DICKPT.

Figure 2 describes the execution model of the second version of CAPE using three nodes. At the beginning, the program is initialized on all nodes and the same sequential code block is executed on all nodes. When reaching an OpenMP parallel structure, the master process divides the tasks into several parts and send them to slave processes using DICKPT. Note that these checkpoints are very small in size, typically very few bytes, as they only contain the results of some very simple instructions to make the difference between the threads, which do not change the memory space that much. At each slave node, after

receiving a checkpoint, it is injected into the local memory space and initialized for resuming. Then, the slave process executes the assigned task, extracts the result, and creates a resulting checkpoint. This last checkpoint is sent back to the master process. The master process then combines them altogether and injects the result into its memory space and send it to all the other slave processes to synchronize the memory space of all processes and prepare for the execution of the next instruction of the program.

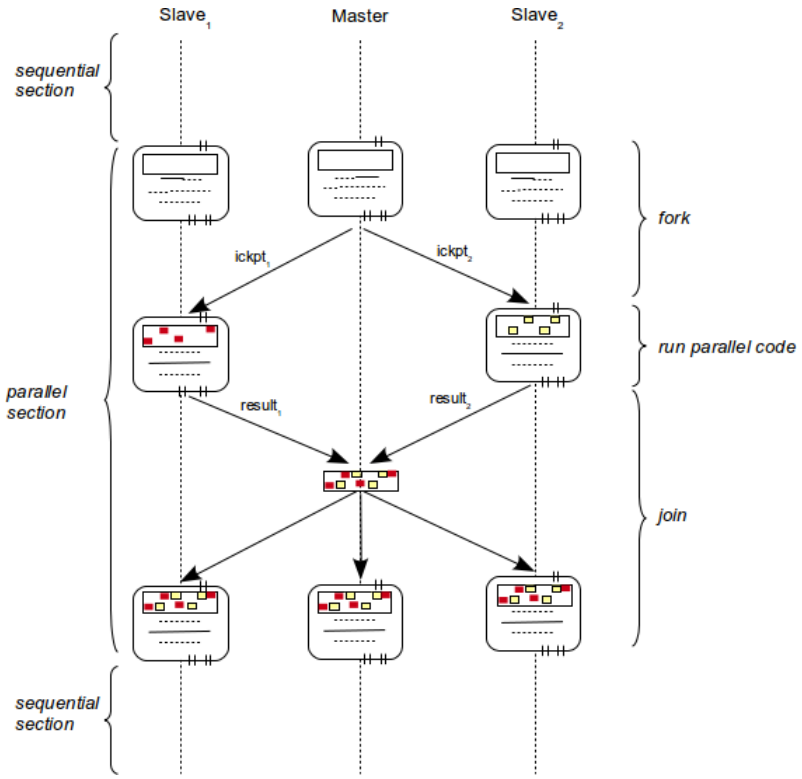



**Fig. 2.** Transferring data between nodes on CAPE

### 3.3 Data transferring

To distribute checkpoints to slave nodes, the master node initialize a master socket to listen the connection requirements from slaves. At a moment, master socket accept a connection request and sends a checkpoint through the established connection. At the slave nodes, to send a checkpoint to the master, it initialize a client socket and try to connect to the master, if the connection is ac-

cepted, the data will be sent. While waiting for the acceptance, the client always maintain a loop to request for the connection.

Sending checkpoints form the master to slave nodes and vice versa is implemented as the following code:

```
if(node == MASTER){  //master node
        initialize a server socket
        waiting and accept connection from 1 slave at each time
        send a DICKPT checkpoint  to slave respectively
}else{ //slave node
        initialize a slave socket
        do{
                request to master for a connection
        }while (connected); //if timeout, retry to connect
        send a DICKPT checkpoint to master node
}
```

To receive DICKPT checkpoints,the master initializes a server socket, accepts connections, receives data from the slaves one by one. At the other side, each slave always maintains a loop to request a connection to server before receiving data. Receiving data job at the master and slave nodes is implemented as the following code:

```
if(node == MASTER){ //master node
        for(i = 0; i < num_nodes; i++){
                wait and accept a connection from slave i
                receive a checkpoint through the socket
                   and inject it into memory space
        }
}else{ //slave node
        initialize a client socket
        do{
                request to the master for a connection
        }while (connected); //if timeout, retry to connect
        receive a checkpoint through the socket
               and inject it into memory space
}
```

From the algorithm presented above, it is easy to see that the use manual sockets to send and receive data wastes time to initialize and establish the connections between nodes for each data exchange requirement. Furthermore, in order to request a connection to the master, the slave always remains a loop, this may take resources of it and of the network. In addition, transferring data by stream using manual socket is not reliable because the risk of conflicts on port numbers, and data are not packaged.

## 4 CAPE using MPI for transferring data

### 4.1 System organization using MPI

Nowadays, parallel programming on clusters have been dominated by message passing, and using MPI [13]. MPI have demonstrated the advantages in different systems in section 2. Moreover, for MPI, data are moved from the address space of one process to the one of another process through cooperative operations on each process. Simply stated, the goal of the MPI is to provide a widely used standard for writing message passing programs. The interface attempts to be practical, portable, efficient and flexible.

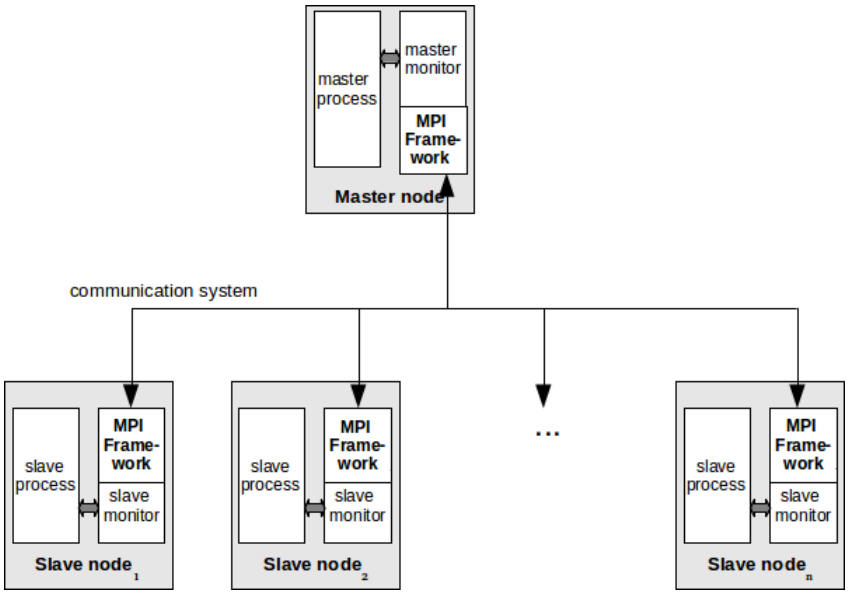



**Fig. 3.** System organization using MPI

In order to use the advantages of MPI, we have changed the organization of CAPE. We add MPI framework into CAPE system and use it as the communicator role. The new organization of CAPE will be shown in the figure 3. In this system, the monitor process use MPI framework to send and receive DICKPT checkpoint. In addition, it also use the MPI routines, that will be reduce time and improve the reliably of the system.

### 4.2 Data transferring

In order to use MPI in CAPE, we implemented the sending and receiving data at master and slaves nodes as the following code:

Sending a DICKPT checkpoint using MPI:

```
if (node == MASTER)
{
        current_slave_node++;
        MPI_Send(current_slave_node,..., DICKPT,.....)
}
else
{
         MPI_Send(0,..., DICKPT,.....)
}
```

Receiving a DICKPT checkpoint using MPI:

```
if (node == MASTER ){   //Master node
        for(i = 1; i< num_nodes; i++){
                MPI_Recv(i,....., DICKPT,.....)
                inject DICKPT checkpoint into memory space
        }
}
else{ //slave node
         MPI_Recv(0,....., DICKPT,.....);
         inject DICKPT checkpoint into memory space
}
```

For this case, MPI libraries are loaded at the beginning of execute time at all nodes, so it is not necessary to initialize at the moment when the nodes prepare to send or receive data. Therefore the execute time can be reduced while comparing with the method using manual sockets. In addition, MPI automatically setup an environment to connect between nodes of a system, that means it is not need to maintain a loop to request the connections from slaves to the master and vice versa, thus the occupation of CPU and other resources can be reduced at this time.

Furthermore, to transfer data by using manual sockets, it is necessary to implement the routines serving to send and receive data on network, especially the routines that are very important to distribute and collect data such as bcast, allreduce, etc. [14]. This requires a lot of programming time and effort and the developed routines, at the first time, are not reliable. Meanwhile, these routines are already available in MPI with high reliability and performance [13]. Moreover, as MPI features, vendor implementations should be able to exploit native hardware features to optimize performance [1]. By all these reasons, exploiting the available abilities of MPI for sending and receiving data on network will be a better choice than using manual sockets while considering the reliability and performance.

## 5 Evaluation and Experimentation

As CAPE's execution model presented in Section 3, the execution time of a parallel section is calculated by equation 1.

$$t = t_{comm} + t_{comp} \tag{1}$$

where $t_{comm}$ is the time to exchange data between nodes, it is the total time of phases sending and receiving DICKPT checkpoints from master node to slave nodes and vice versa. $t_{comp}$ is the time to execute application codes at master and slave nodes. For the methods mentioned in Section 3 and Section 4, $t_{comp}$ in these methods are equal.

In the method using manual sockets presented in Section 3, the time consumed in phases sending and receiving DICKPT checkpoints is calculated by equation 2.

$$t_{comm_i} = p(t_{startup} + t_{data}) \tag{2}$$

where $p$ is the number of slave nodes, $t_{startup}$ is time of startup step, it is the time to initialize, connect and prepare to send and receive data of each time when a checkpoint has to be exchanged, $t_{data}$ is the time to send and received data.

For using MPI, it works as `scatter` mechanism, the startup step will execute in the same time at all nodes, thus the communication time of phase sending or receiving is calculated by equation 3.

$$t_{comm_i} = t_{startup} + p.t_{data} \tag{3}$$

From equation 2 and 3, it is easy to see that at each phase sending or receiving DICKPT checkpoints, the communication time of using MPI method is less than using manual sockets. Thus, the total execution time is reduced.

In order to practically verify the above argues, some performance measurements have been conducted on a cluster. This test is composed of nodes including Intel(R) Pentium(R) 4 CPU 3.00GHz and 2 GB RAM, operated by Linux kernel 3.13.0 with the Ubuntu 14.04, and connected by a standard Ethernet at 100 MB/s. The cluster consists of three nodes - one master and two slaves. In order to avoid as much as possible external influences, the entire system was dedicated to the tests during performance measurements.

The program used for tests is a matrix-matrix product for which the size varies from 3,000×3,000 to 9,000×9,000. Matrices are supposed to be dense and no specific algorithm has been implemented to take into account sparse matrices. Each experiment has been performed at least 10 times and a confidence interval of at least 90% has always been achieved for the measures. Data reported here are the means of the 10 measures.

Fig. 4 shows the total execution time (in second) using MPI and manual sockets that resemble each other. Since the major parts of the program serve for computing works, the time of transferring data between nodes takes a very small scale. Therefore, although there is little improvement in the time to send and receive results, but in overall it does not to affect the total execution time of the program.

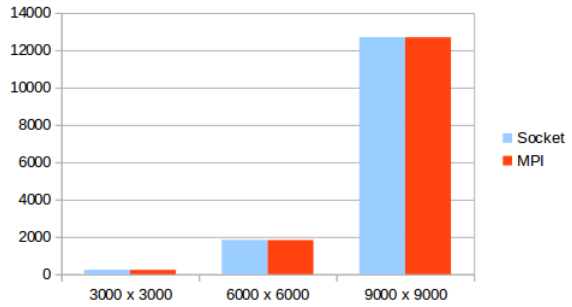
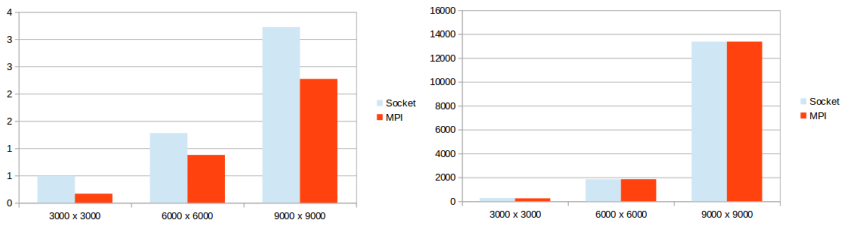



**Fig. 4.** Total execution time (in seconds) of CAPE using MPI and Socket.

The details are shown in Fig. 5. In that, `Init` step creates and sends DICKPT checkpoints to slave nodes, `Update` step waits and receives calculated results from slave nodes and injects them into memory space.

The `Init` step creates DICKPT checkpoints with some bites of data, then sends them to slave nodes on the system. At this step, Time used primarily to send data to the node, so we can see the difference between the methods used MPI and manual socket.

For the `Update` step, it take almost of the time to wait for computation at slaves, so the communication time is not significant while comparing with the overall time of the program. That why in Fig. 5 the time of two methods is similar.




(a) Init (b) Update

**Fig. 5.** Execution time (in seconds) of Init and Update steps on CAPE

From the result above, it is clear that using MPI consumes less time than using manual sockets, but the difference is not significant while comparing with the overall time of the program. However, if the number of nodes is increasing, using MPI is decreasing when compared with manual socket.

## 6 Conclusion and future work

From the analysis and experiments above, we found that it is better to replace the use of manual sockets by use MPI for data exchange. This helps CAPE achieve higher stability, security and tend to improve performance for the programs using functions supported by MPI, such as bcast, allreduce, etc.

In the current version, CAPE does not support for problems using functions bcast and reduce, so we can not do experiment to compare the results between the two methods in these cases. In near future, we will continue to develop CAPE to supports these ones, and CAPE using MPI will be able to demonstrate its strengths. Besides, in the next time, we will develop CAPE to support GPU and cloud systems.

## References


1. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum (2012)
2. OpenMP specification 4.0. OpenMP Architecture Review Board. (2013)
3. Viet Hai Ha, Eric Renault: Design and Performance Analysis of CAPE based on Discontinuous Incremental Checkpoints. Proceedings of the IEEE Conference on Communications, Computers and Signal Processing. Victoria, Canada, August 2011 (2011)
4. Viet Hai Ha, Eric Renault: Improving Performance of CAPE using Discontinuous Incremental Checkpointing. Proceedings of the IEEE International Conference on High Performance and Communications 2011 (HPCC-2011). Banff, Canada, September 2011 (2011)
5. Viet Hai Ha, Eric Renault: Discontinuous Incremental: A New Approach Towards Extremely Checkpoint. Proceedings of IEEE International Symposium on Computer Networks and Distributed System (CNDS2011), Tehran, Iran, February 2011 (2011)
6. Yongjin Li, Weichang Shen, Anlei Shi: MPI and OpenMP Paradigms on Cluster with multicores and its application on FFT. Proceedings of the conference on Computer Design and Application (ICCDA 2010) (2010)
7. Rolf Rabenseifner, Georg Hager, Gabriele Jost: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. Proceedings of Euromicro International Conference on Parallel, Distributed and Network-based Processing 17th (2009)
8. Wong H.J. , Rendell A.P. : The design of MPI based distributed shared memory systems to support OpenMP on clusters. Proceedings of IEEE International Conference on Cluster Computing, (2007)
9. Hao Wang, Potluri S., Bureddy D., Rosales C., Panda D.K. : GPU-Aware MPI on RDMA-Enabled Clusters: Design, Implementation and Evaluation. IEEE Transactions on Parallel and Distributed Systems (Volume:25 , Issue: 10), October 2014 (2014)
10. Potluri S.,Wang H., Bureddy D., Singh A.K., Rosales C., Panda, D.K. : Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication. Proceedings of the IEEE International Conference on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW) (2012)

11. Balamurugan B., Krishna P.V. , Rajya Lakshmi G.V. , Kumar, N.S. : Cloud cluster communication for critical applications accessing C-MPICH. Proceedings of the International Conference on Embedded Systems (ICES 2014) (2014)
12. Shivaramakrishnan S., Babar S.D.: Rolling curve ECC for Centralized Key Management System used in ECC-MPICH2 Proceedings of the IEEE Global Conference on Wireless Computing and Networking (GCWCN 2014) (2014)
13. Motohiko Matsuda, Tomohiro Kudoh, Yuetsu Kodama, Ryousei Takano, and Yutaka Ishikawa: Efficient MPI Collective Operations for Clusters in Long-and-Fast Networks. Proceedings of the IEEE International Conference on Cluster Computing. (2006)
14. Rabenseifner R.: Automatic MPI Counter Profiling of All Users: First Result on a CRAY T3E 900-512. Proceedings of the Message Passing Interface Developers and Users Conference 1999 (MPIDC99), (1999)