# Improving the Reliability and the Performance of CAPE by Using MPI for Data Exchange on Network

Van Long Tran[1(✉)], Éric Renault[1], and Viet Hai Ha[2]

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

**Abstract.** CAPE — which stands for Checkpointing Aided Parallel Execution — has demonstrated to be a high-performance and compliant OpenMP implementation for distributed memory systems. CAPE is based on the use of checkpoints to automatically distribute jobs of OpenMP parallel constructs to distant machines and to automatically collect the calculated results on these machines to the master machine. However, on the current version, the data exchange on networks use manual sockets that require time to establish connections between machines for each parallel construct. Furthermore, this technique is not really reliable due to the risk of conflicts on ports and the problem of data exchange using stream. This paper aims at presenting the impact of using MPI to improve the reliability and the performance of CAPE. Both socket and MPI implementations are analyzed and discussed, and performance evaluations are provided.

**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 and multi-cores, an easy-to-use parallel programming language is an important factor.

MPI [1] (which stands for Message Passing Interface) is the de-facto standard for developing parallel applications on distributed-memory architectures. Essentially, it provides point-to-point communications, collective operations, synchronization, virtual topologies, and other communication facilities for a 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 requires the programmers to explicitly distribute the program onto the nodes. Moreover, some operations, like sending and receiving data or the synchronization of processes, must be explicitly specified in the program.


S. Boumerdassi et al. (Eds.): MSPN 2015, LNCS 9395, pp. 90–100, 2015.
DOI: 10.1007/978-3-319-25744-0_8

OpenMP [2] also has become a standard for the development of parallel applications but 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, i.e. the programmer can start with a sequential version of a program and step by step add OpenMP directives to change it into a parallel version. 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 have been some efforts to run OpenMP programs on distributed-memory systems. Among them, CAPE [3,4] is a tool to compile and provide an environment to execute OpenMP programs on distributed-memory architectures. This solution provides both high performance and a compiler that is fully-compatible with the OpenMP standard.

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

In the current version of CAPE, data exchanged between nodes are computed using DICKPT [3,5] (which stands for Discontinuous Incremental Checkpoints), and are transferred over the network using manual sockets. However, initializing, connecting and listening to sockets at runtime is clearly a waste of time. In addition, this approach is weak in terms of reliability, due to the difficulty to manage the data exchanged over the network.

This paper aims at presenting the approach focusing on the reduction of the checkpoint's transfer time and increasing the reliability of data transfers of CAPE over the network. The remainder of the paper is as follows: first, some related works and the advantages of using MPI to transfer data over the network are listed in Sect. 2. Section 3 discusses and analyzes the current version of CAPE using manual sockets. Section 4 proposes a new method that use MPI instead of manual sockets. Section 5 compares the two methods by presenting an evaluation and some experimental results. At the end, Sect. 6 draws some conclusions and future works.

## 2 Related Works

Using the MPI framework to transfer data between nodes over the network has been developed and widely applied today. This allows to achieve high reliability, security, portability, integrity, availability and high-performance of the transferred data.

A typical example is the combination of MPI and OpenMP. In this case, the MPI framework is used to send data and code from the master node to all working nodes in the network. At the working node side, the OpenMP framework is used to execute the assigned task in parallel. Finally, the results from the working nodes are sent back to the master node by using explicit MPI codes. Although this way takes time and efforts from the programmer, it takes advantages of the performance and the integrity. In [6], authors show that this method can achieve high efficiency and scalable performance. In [7,8], authors show a reduction of the communication needs and memory consummation, or an improvement of the load-balancing ability.

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

The result above is very important for the orientation of the future development of CAPE using MPI. In this paper, the MPI framework is used by CAPE to transfer checkpoints between nodes. In the future, MPI will bring an even more important contribution to CAPE as the latter aims at supporting GPU and cloud computing infrastructures in the near future.

Note that the use of MPI by CAPE as presented in this paper is completely different from the combination of MPI and OpenMP as mentioned above or from the translation of OpenMP constructs into MPI function calls. In fact, the use of MPI as a support for CAPE 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 fully compliant with the OpenMP standard and programmers do not need to modify their application program source codes. With CAPE, the role of MPI only consists in transferring checkpoints over the network, while for most other cases programmers need to modify their source codes and, as a consequence, cannot provide a fully-compliant implementation of OpenMP.

## 3 CAPE Based on Manual Sockets

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

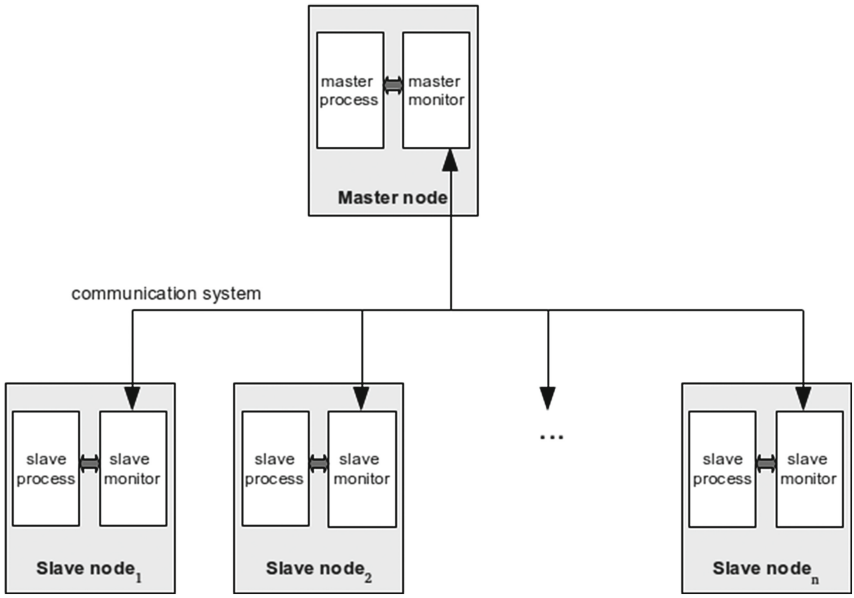



**Fig. 1.** CAPE organization.

### 3.1 Execution Model

CAPE is an 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

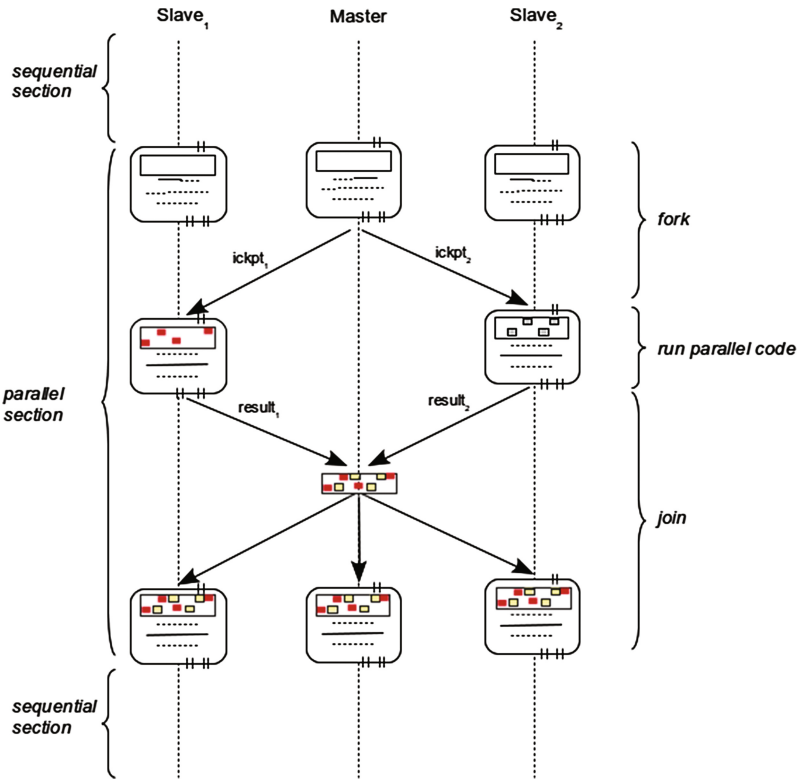



**Fig. 2.** Data transfer between nodes in CAPE.

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 all resulting checkpoints together, injects the result into its memory space and sends 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.

### 3.2 Data Transfer

In order to distribute checkpoints to slave nodes, the master node initializes a socket to listen to the connection requests from slaves. After the master accepts a connection request, it sends a checkpoint to the slave node through the established connection. Figure 3 presents the algorithm used to send checkpoints from the master to all slaves.

At the slave node side, a checkpoint must be returned to the master after the execution of the parallel part. The slave node initializes a client socket and tries to connect to the master. After the connection is accepted, the checkpoint is sent to the master.

```
if ( node == MASTER ) {
        initialize a server socket
        foreach ( slave ) {
                wait and accept a connection from a slave
                send a checkpoint to the slave
        }
} else {
        initialize a slave socket
        do {
                request the master for a connection
        } while ( ! connected )
        receive the checkpoint from the master node
        inject the checkpoint into the memory space
}
```

**Fig. 3.** Master-to-slave transfer using manual sockets.

```
if ( node == MASTER ) {
        foreach ( slave ) {
                wait and accept a connection from a slave
                receive a checkpoint through the socket
                inject the checkpoint into the memory space
        }
} else {
        initialize a client socket
        do {
                request the master for a connection
        } while ( ! connected )
        send the checkpoint through the socket
}
```

**Fig. 4.** Slave-to-master transfer using manual sockets.

To receive DICKPT checkpoints from the salves, the master initializes a server socket, accepts connections and receives data from the slaves the one after the one. At the other side, each slave always maintains a loop to request a connection to server before receiving data. The algorithm is summarized in Fig. 4.

From the two algorithms presented above, one can see that the use of manual sockets to send and receive data involves a waste of time to initialize and establish the connections between the nodes for each data exchange requirement. Furthermore, in order to request a connection to the master, the slave always performs a polling. This requires resources both on the node and over the network. In addition, transferring data by means of a stream using manual sockets is not reliable as the risk of conflicts on port numbers and data is not packaged.

## 4 CAPE Based on MPI

Nowadays, parallel programming on clusters have been dominated by message passing, and using MPI [13] has become a de-facto standard. MPI has demonstrated advantages over other systems (see Sect. 2). Moreover, for the case of MPI, data are transfered from the address space of one process to the one of another process through cooperative operations on each process. Simply stated, the goal of MPI is to provide a widely used standard for writing message-passing programs. The interface aims at being practical, portable, efficient and flexible.

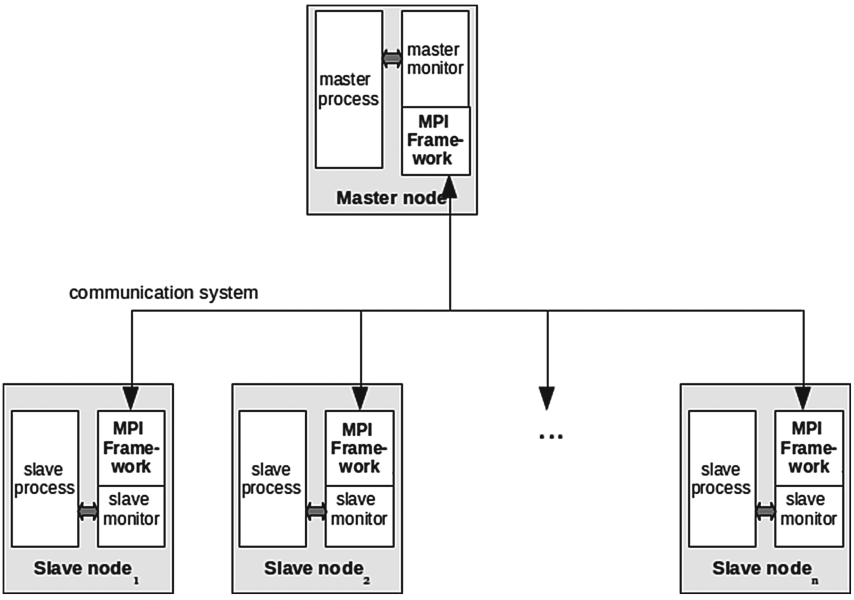



**Fig. 5.** MPI-based CAPE organization.

In order to take advantage of the MPI benefits, the organization of CAPE has been moved from a socket-based communication system to the MPI framework. The new organization of CAPE is shown in Fig. 5. With this new organization, the monitor process uses the MPI framework to send and receive DICKPT checkpoints. In addition, it also uses MPI routine to reduce the time overhead and improve the global reliably of the system.

### 4.1 Data Transfer

In order to provide a new version of CAPE on top of MPI, the sending and the receiving of data at both the master and the slaves nodes have been implemented as presented in pseudo-code on Figs. 6 and 7.

For this implementation, the MPI library is loaded by each node at the beginning of the execution, so that it is not necessary to initialize it when the nodes

```
if ( node == MASTER ) {
        current_slave_node ++
        MPI_Send ( current_slave_node, ... , DICKPT, ... )
} else {
         MPI_Recv ( 0, ..., DICKPT, ... )
        inject the checkpoint into the memory space
}
```

**Fig. 6.** Master-to-slave transfer using MPI.

```
if ( node == MASTER ) {
       foreach ( slave ) {
               MPI_Recv ( i, ..., DICKPT, ... )
               inject the checkpoint into the memory space
       }
} else {
         MPI_Send ( 0, ..., DICKPT, ... )
}
```

**Fig. 7.** Slave-to-master transfer using MPI.

need to send or receive data. Therefore, the execution time is reduced when compared with the manual-socket implementation. In addition, MPI automatically setup connections between nodes to perform data transfers which means that there is no need for maintaining a loop to request a connection from the slaves to the master. As a result, the use of the CPU and other resources is considerably reduced at this time.

Furthermore, the transfer of data using manual sockets requires the implementation of routines to send and receive data over the network, especially those routines that are very important to distribute and collect data, such as broadcast and reductions [14]. This requires a huge effort in terms of development and ensuring the reliability of such an implementation is not easy. Meanwhile, all these routines have been made available in the MPI framework and after many years of customization they are regarded as highly reliable and efficient [13]. Moreover, for the case of MPI, vendor implementations usually exploit native hardware features to optimize the performance [1]. For all these reasons, using MPI for sending and receiving data over the network is better than using manual sockets, especially when considering reliability and performance.

## 5 Experimentation and Evaluation

Let $t_{comm}$ be the time to exchange data between the nodes, i.e. the total time for sending and receiving DICKPT checkpoints from the master to all slave nodes and vice versa. Let $t_{comp}$ be the time to execute the application code at both the master and slave nodes. For the two methods mentioned in Sects. 3 and 4, $t_{comp}$ in the same.

According to the execution model of CAPE as presented in Sect. 3, the execution time of a parallel section can be computed using Eq. (1).

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

Let $p$ be the number of slave nodes, $t_{startup}$ be the time to set up a socket, i.e. the time to initialize, connect and prepare to send and receive data of each time when a checkpoint has to be exchanged, and $t_{data}$ be the time to send and received data.

When using manual sockets as presented in Sect. 3, the time required to send and receive DICKPT checkpoints can be computed using Eq. (2).

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

With MPI, the `scatter` operation has been used so that the startup step is executed at the same time on all nodes. As a result, the communication time for the sending or receiving phase can be computed using Eq. (3).

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

From Eqs. (2) and (3), one can see that each time a DICKPT checkpoint has to be sent or received, the communication time when using the MPI method is always more efficient than using manual sockets.

In order to verify the above arguments, some performance measurements have been conducted on a real cluster. The plateform is composed of nodes including four 3-GHz Intel(R) Pentium(R) CPUs with 2 GB of RAM, operated by Linux kernel 3.13.0 with the Ubuntu 14.04 flavour and connected by a standard 100 Mbits/s Ethernet. The cluster consists of three nodes, i.e. one master and two slaves. In order to avoid as much as possible any external influences, the entire system was dedicated to the tests during performance measurements.

The program used for tests is the 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.

Figure 8 shows the total execution time (in seconds) for both MPI and the manual socket implementation. Since the major parts of the program serve for computing works, the time for transferring data between nodes takes a very small scale. Therefore, although there is a significant improvement in the time to send and receive results, the overall execution time of the program remains almost the same.

The details are shown in Fig. 9. During the `Init` step, the DICKPT checkpoints are created and sents to the slave nodes, while during the `Update` step the master waits for the reception of the computed results from the slave nodes and injects them into its memory space.

The DICKPT checkpoints created during the `Init` step are composed of very few bytes of data, so that the time to send these checkpoints is very short. For

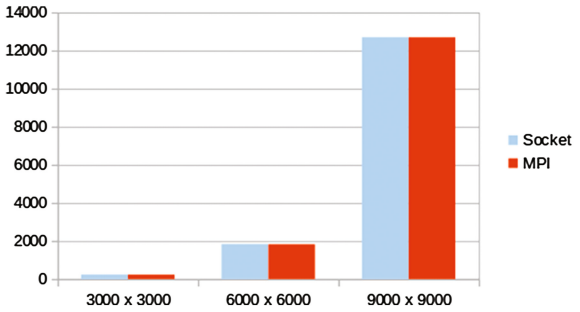
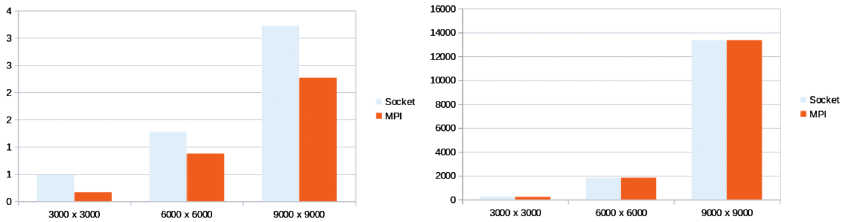



**Fig. 8.** Total execution time (in seconds) of CAPE using MPI and Socket.

the `Update` step, it takes almost the same amount of time to wait for the result of the computations from the slaves, so that the communication time is not really significant as compared with the overall time of the program. This results in very similar overall times for both methods as shown in Fig. 9.




(a) Init (b) Update

**Fig. 9.** Execution time (in seconds) for both Init and Update steps.

From the result above, it is clear that using MPI consumes less time than using manual sockets. However, the difference is not significant while comparing the overall time of the program.

## 6 Conclusion and Future Work

From the analysis and the experiments above, we found that it is interesting to replace the use of manual sockets by use of MPI for data exchange. This helps CAPE achieves higher stability, security and tends to improve performance for the programs using functions supported by MPI, such as broadcast and reductions.

In the near future, we will keep on developing CAPE to support other constructs of OpenMP in order to allow a larger set of algorithms to run on distributed-memory architectures. Moreover, it is also planed to port CAPE on top of other architectures like GPU-based clusters for example.

## References


1. MPI: A Message-Passing Interface Standard. Message Passing Interface Forum (2012)
2. OpenMP specification 4.0. OpenMP Architecture Review Board (2013)
3. Ha, V.H., Renault, E.: Design and performance analysis of CAPE based on discontinuous incremental checkpoints. In: Proceedings of the IEEE Conference on Communications, Computers and Signal Processing. Victoria, Canada, August 2011
4. Ha, V.H., Renault, E.: Improving performance of CAPE using discontinuous incremental checkpointing. In: Proceedings of the IEEE International Conference on High Performance and Communications 2011 (HPCC-2011), Banff, Canada, September 2011
5. Ha, V.H., Renault, E.: Discontinuous incremental: a new approach towards extremely checkpoint. In: Proceedings of IEEE International Symposium on Computer Networks and Distributed System (CNDS 2011), Tehran, Iran, February 2011
6. Li, Y., Shen, W., Shi, A.: MPI and OpenMP paradigms on cluster with multicores and its application on FFT. In: Proceedings of the Conference on Computer Design and Application (ICCDA 2010) (2010)
7. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In: Proceedings of 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing (2009)
8. Wong, H.J., Rendell, A.P. : The design of MPI based distributed shared memory systems to support OpenMP on clusters. In: Proceedings of IEEE International Conference on Cluster Computing (2007)
9. Wang, H., Potluri, S., Bureddy, D., Rosales, C., Panda, D.K.: GPU-aware MPI on RDMA-enabled clusters: design, implementation and evaluation. IEEE Trans. Parallel Distrib. Syst. **25**(10), 2595–2605 (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. In: Proceedings of the IEEE International Conference on Parallel and Distributed Processing Symposium Workshops & Ph.D. 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. In: 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. In: Proceedings of the IEEE Global Conference on Wireless Computing and Networking (GCWCN 2014) (2014)
13. Matsuda, M., Kudoh, T., Kodama, Y., Takano, R., Ishikawa, Y.: Efficient MPI collective operations for clusters in long-and-fast networks. In: 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. In: Proceedings of the Message Passing Interface Developers and Users Conference 1999 (MPIDC 1999) (1999)