New micro-benchmark to investigate the scalability of MPI point-to-point matching

Jia Song

August 22, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2007

Abstract

With the development of high performance techniques, the computational capability of supercomputers will be improved to an exascale level to process larger sets of data. It is anticipated in these future circumstances that the latency and overhead per message will be one of the most important factors of the machine's performance. This project develops a new way to investigate the performance of the MPI libraries of some the most advanced HPC facilities. By implementing the micro-benchmark suite on Blue Gene/Q, HECToR, INDY and ECDF, it was possible to obtain the time spent on forcing the matching order of the messages on each machine which provides theoretical references for judging the performance of MPI libraries on HPC systems.

Content

List of Figures

List of Tables

Acknowledgements

I sincerely thank my supervisor Dan Holmes for his guidance and support throughout the life time of the whole project. His feedback during our weekly meetings was highly informative and invaluable to the success of this project.

I want to thank Fiona Reid and Stephen Booth for guiding me while Dan was away for a brief period.

1. Introduction

In June this year, the International Supercomputing Conference 2013 released the newest Top 500 List. The Chinese Supercomputer Tianhe-2 won first with a performance of 33.86 petaflop/s [1]. Over the past few years, computing capabilities of the top supercomputers in the world has shown both an impressive and sustained growth. The performance has increased by approximately 10x every 3.6 years. The first generation of supercomputers started with 1 Gflop/s in 1985. The Intel ASCI Red later achieved 1 TFlop/s in 1997, the Roadrunner developed with 1 PFlop/s in 2009, and presently the exaflop machine with 1000 PFlop/s is expected to be implemented around 2018 [2].

Exascale computing is able to undertake large scale scientific computations with a huge data set which may consist of millions of messages, a large amount of which may be transferred at one time. Hence, the per message overheads and latency as well as the actual message sending time will play a more and more important role in the efficiency of the HPC systems. This project investigates the performance issues of MPI libraries on the four most advanced supercomputers by using a new micro-benchmark suite.

Various kinds of benchmark suites for MPI programs exist now, but at present they are not able to meet the need of measuring the overheads and latency for each message. They may contain communication overheads, time for invoking subroutine calls and other factors. The micro-benchmark suite aims to execute a multi-pingpong program with a number of messages (a data package) between two nodes in two orders, InOrder or ReverseOrder. By calculating the difference between the numbers of matching in InOrder or ReverseOrder communication patterns as well as the difference between the time of in order matching and out of order matching to compute the matching time and overheads of a message with a certain size.

The four HPC machines involved in this project are the Blue Gene/Q, HECToR, INDY and ECDF. Blue Gene/Q is the third generation product of the IBM Blue Gene project which aims to invent the fastest and most powerful computing facilities [3]. HECToR is a parallel supercomputer which represents for the UK's high-end computing resource, funded by the UK Research Councils [4]. INDY is an industry machine maintained by Edinburgh Parallel Computing Center [5]. ECDF is the Edinburgh Compute and Data Facility which belongs to the University of Edinburgh [6]. All these machines provide resources to compute large data MPI programs.

This dissertation is structured as following:

Chapter 2 sets out the background research of the project which is based on the thesis of *McMPI - a Managed-code Message Passing Interface Library for High Performance Communication in C#* [7]. The programs are written in C programming language with Massage Passing Interface model, furthermore this is explained in detail. Memory architecture is also a dominating factor of the efficiency of computers, some typical architecture of these machines will also be discussed. Finally, some other benchmark suites will be introduced.

Chapter 3 outlines the code design consideration, mathematic model, implementation details and data analysis strategy of the program. In order to satisfy the particular needs of the program and to guarantee accuracy, two algorithms are developed for the micro-benchmark code, Tag Algorithm and Comm Algorithm.

Chapter 4 introduced the major configuration information and performance information of four the HPC machines (Blue Gene/Q, HECToR, INDY and ECDF. Moreover, the programs output results are illustrated after the configuration of each machine.

Chapter 5 compares and evaluates the performance and computational abilities of these four machines.

Chapter 6 draws a conclusion of this project which includes the findings during the project life time, these four supercomputers' performance, their latency and matching time for a message of the HPC facilities. It also recommends some further work to the project.

2. Background

This chapter provides a summary of the principle background research behind this project. Firstly is provides an overview of the key literature underpinning this study. Secondly, it highlights some MPI communication mode and subroutines applied to build the benchmark code. Additionally, it reviews the memory architectures of the HPC systems. Finally, primary existing benchmark suites are considered and discussed.

2.1 Background Literatures

One of the most notable studies related to this project is Daniel Holmes's thesis, *McMPI - a Managed-code Message Passing Interface Library for High Performance Communication in C#* [7], which aims to combine and reinforce the best-practice academia on technological advancement in the sectors of high performance and the current commercial high productivity computing.

In this day and age, the HPC is prevalent in the academic arena designed to do large scale simulations and computations, especially in physics, chemistry and biology disciplines. One of the most important HPC technologies is the Massage-Passing model which taken a dominating role in the efficient parallel programming on distributed memory architectures. All the top supercomputers in the world are equipped with MPI to deal with big data parallel programming with different MPI libraries. However, there are only some existing universal MPI libraries for C and Fortran programming language not for C#. As one of the object-oriented computer languages C# is able to improve the programmers'' productivity and programs' portability. It is therefore necessary to enhance and extend the paradigm in C# by building with MPI libraries in the near future.

The highlight of the project is that it establishes a thread-to-thread delivery model and regards every thread as a rank rather than treating a rank as a separate process. It may result in a shorter time for message transferring between threads when we use the thread-as-rank model to test a communication pattern.

In summary, Holmes's work proves that pure C# (one of the .Net suite of computer languages) can be employed to build a reliable high performance MPI library with semantics and syntax following the MPI version 2.0 standards [8]. Though the code of the micro-benchmark project is

written in C programming language, Holmes's study also provides an initial idea and structure of the communication model and algorithm prototype for the benchmark program.

2.2 MPI Programming Model

The major MPI technic applied in the micro-benchmark suite is the MPI non-blocking communication. This constructs core algorithms. Another term is the MPI Message Passing Protocols, which effects performance of communication.

2.2.1 MPI Non-blocking Communication

Non-blocking communications allow the overlap computation with communication to deliver performance gains [8]. Typically, after initiating the communication the process can return to perform operations. Then, at some later time, it must test or wait for the completion of the non-blocking operation. There are four primary reasons to employ the non-blocking communication in the program. Above all, overlapping communication and other useful work is conducive to hiding the communication cost. The other advantages of non-blocking communication are avoiding: deadlocks, idle processors and unnecessary synchronization resulting in benefits for performance. However, limitations and weaknesses need to be noted. It is not safe to modify or operate on the buffers (e.g. send buffer) before completion of a non-blocking operation. Hence, the programmer should pay special attention to ensure the buffers are free for reuse so the data is sent and received correctly. Furthermore, there are four communication modes for the non-blocking communication: Standard send (MPI_Isend), Synchronous send (MPI_Issend), Buffered send (MPI_Ibsend) and Ready send (MPI_Irsend) and a non-blocking receive (MPI_Irecv).

The completion of each non-blocking communication is achieved by testing (MPI_Test or MPI_Testall) or waiting (MPI_Wait or MPI_waitall).Thus, the behaviors of the receiver cannot effect the operation on the sender.

2.2.2 MPI Message Passing Protocols

Typically MPI implementations utilize different underlying protocols depending on the size of the message, so the protocol may have an effect on the performance dependent on the size of the data package changes [8]. A noteworthy feature is that the protocols are not defined by the MPI standard, but are determined by implementers. A combination of protocols for the same MPI routine may also be applied to the MPI implementations. There are many variants of these basic protocols. The two most common protocols are the eager protocol and rendezvous protocol.

Eager protocol is an asynchronous protocol by which the sending buffer can send messages without acknowledgement receiving process. The performance can benefit from the eager protocol because it reduces synchronization delays. The weakness is that it is not scalable and may cause memory exhaustion. In general, the eager protocol used for small messages, and the limit message size can also be changed by the number of MPI tasks. The default eager protocol message size varies in different environments, which can be set with MPI_EAGER_LIMIT. For example, the IBM eager protocol message size for 33 to 64 MPI tasks is 1024 bytes. Rendezvous protocol is a synchronous protocol which requires a matching receive launched before the send operation to complete. The protocol is memory friendly because only the small message envelops need to be buffered however a downside is that it may cause a higher synchronization delays.

2.3 Architecture

This section introduces some very commonly used memory architectures for HPC facilities. These memory architectures include distributed memory architecture, shared memory cluster, Symmetric MultiProcessing (SMP) architecture, and cache-coherency Non Uniform Memory Access (cc-NUMA).

2.3.1 Distributed Memory Architecture

The majority of HPC facilities are distributed memory architectures (Figure 1). All processors have their own local memory separately. They are connected with each other by the interconnect mechanism and communicate with each other via explicit message passing. This is a highly scalable architecture as adding processors increases memory bandwidth. The disadvantage of distributed memory architecture cannot be ignored. Its scalability relies on a good interconnect and the system management overhead may be quite high.

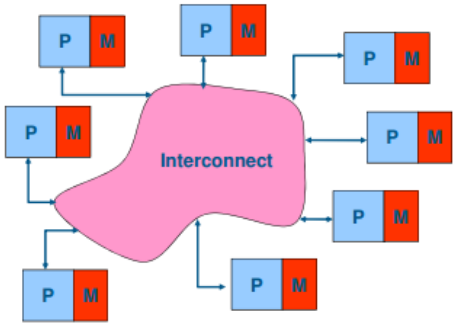

Figure 1: Distributed memory architecture

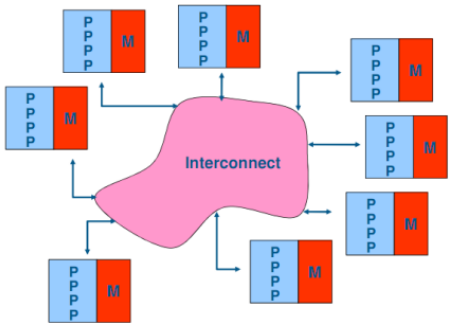

Figure 2: Shared memory cluster architecture

2.3.2 Shared Memory Cluster

The distributed machine can be configured with SMP (Symmetric MultiProcessing) node which contains multiple processors and each processor has equal access to all parts of memory. It will result in a new architecture called shared memory cluster (Figure 2). This new formation combines features of two architectures with shared memory within a node and distributed memory between nodes. It means it constructed as a standard distributed memory machine but with more powerful nodes. Thus, the scalability, availability and other computational capabilities can be enhanced by this cluster system. The memory bus bottleneck can be avoided, however the bandwidth of the interconnect could be too slow to sustain high performance gains.

2.3.3 cc-NUMA (Non Uniform Memory Access)

The cc-NUMA architecture (Figure 3) follows in scaling from SMP architectures. It refers to memory access time depends on the memory location of a processor. On this basis each processor has some fast local memory and slow remote memory, and the remote memory can be accessed via a global address space. Every process has a single address so the cache misses and conflicts can be decreased. It also shows a low latency and high bandwidth global memory. All the capabilities scale as the system grows. But the access remote memory latency is much greater than the local memory latency, therefore the time for sending messages between two nodes may fluctuate quite heavily.

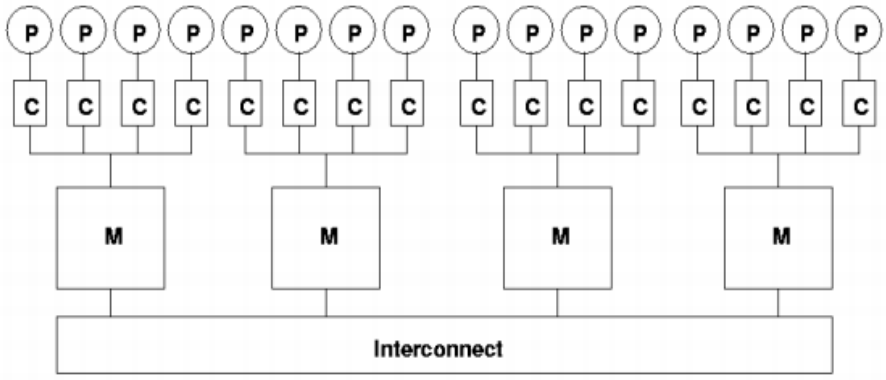

Figure 3: cc-NUMA architecture

2.4 Benchmark Classification

2.4.1 Pingpong and Pingping Benchmark

The pingpong or pingping benchmarks are point-to-point MPI benchmark programs in which two processes (rank 0 and rank 1) repeatedly pass a message back. They mainly aim to measure latency and bandwidth. MPI standard blocking communication is normally used. The pingpong or pingping patterns are done in a loop. To be more specific, rank 0 should send a message to rank 1, rank 1 should receive this message then send the same data back to rank 0, rank 0 should receive the message from rank 1 and then return it and so o. The timing calls are inserted before the iteration and after the last iteration to measure the time taken by all the communications.

2.4.2 Intel MPI Benchmarks

As a widely used set of benchmarks, Intel MPI Benchmarks provide an efficient way to measure the performance of some of the specific MPI sunroutines. It is comprised of three parts: IMB-MPI1, IMB-MPI2 and IMB-IO [9].

The objectives of the Intel® MPI Benchmarks are [9]:

- *Provide a concise set of benchmarks targeted at measuring the most important MPI functions.*
- *Set forth a precise benchmark methodology.*
- *Report bare timings rather than provide interpretation of the measured results. Show*

throughput values if and only if these values are well defined.

Intel® MPI Benchmarks is developed using ANSI C plus standard MPI.

Intel® MPI Benchmarks is distributed as an open source project to enable use of benchmarks across various cluster architectures and MPI implementations [9].

Intel MPI Benchmarks offer a set of performance measurements to both the MPI point-to-point and global communication operations. The outputs of the benchmark programs are able to measure the overall system performance by measurements such as network latency, node performance and throughput. Furthermore, the efficiency of the MPI implementation is another key element of Intel MPI Benchmarks [9]

3. Methodology

Two versions of the code, *Benchmark_Tag* and *Benchmark_Comm*, using different algorithms (Tag Algorithm and Comm Algorithm) are described in sections 3.1 and 3.2. The main distinguishing element of the two versions is the different approaches to forciing the ordering of matching the messages in each data package. More specifically, the Tag Algorithm uses different tags to force the matching order of the messages, whereas the Comm Algorithm uses different communicators to achieve the same goal. Thus, the communication patterns are labeled InOrderTags and ReverseOrderTags for the Tag Algorithm but InOrderComms and ReverseOrderComms for the Comm Algorithm. This chapter mainly explains the methodology for the two algorithms. For each algorithm, the methodology contains the design of the micro-benchmark code, the mathematic model, the implementation processes and the analysis strategy.

It should be noted that the In-Order means the first node sending messages from 1 to *n* according to the sequence of the messages while the receiver on the opposite side receives all the messages in same order by using different tags or communicators. The Reverse-Order refers to messages sent by one node in order but matched at the target node in reverse order.

3.1 Tag Algorithm

3.1.1 Code Design for Tag Algorithm

The micro-benchmark code works as a multi-pingpong program. The data package contains a number of messages (e.g. 10 or 45) which is sent from one node, whilst a node from another side receives all the messages then responds with the same amount of messages. The operation of the program can be changed by inputting three parameters: 1) the number of messages to be sent per data package, 2) the length of each message (note: each length is the number of 8-byte double-precision floating point values, the size of messages is 8*length bytes) and 3) the number of iterations between two nodes. It is quite flexible so that users can control the program according to the machine situation. The Figure 4 simulates the above described communication process.

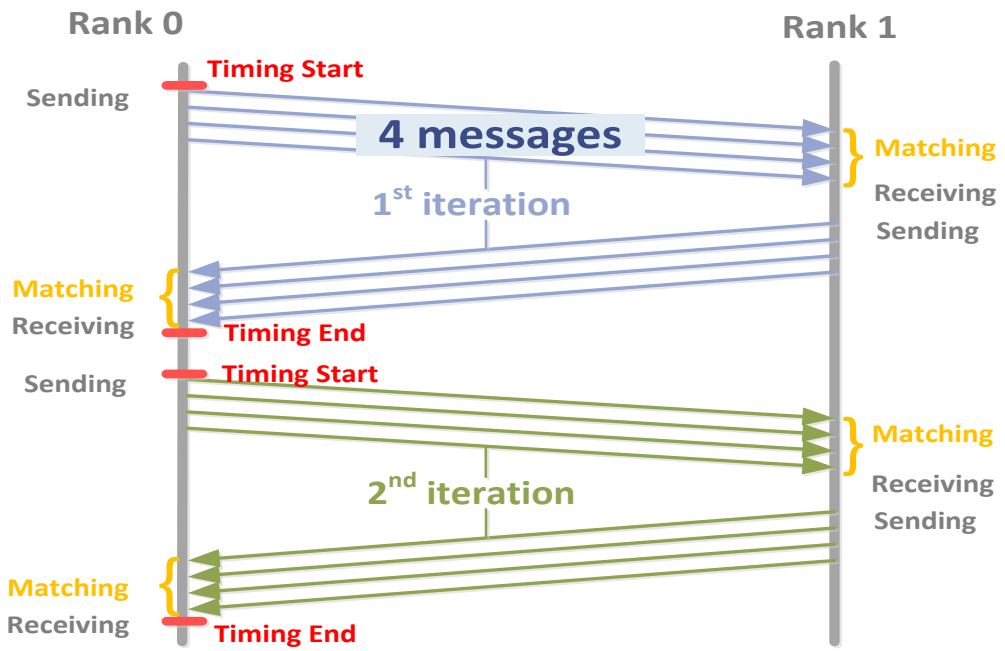

Figure 4: illustrates the In-Order algorithm showing two complete iterations with each data block containing 4 messages

The communication patterns in the Tag Algorithm apply the InOrderTags pattern and ReverseOrderTags pattern. In both patterns, point-to-point non-blocking communication standard mode is utilized to send and receive messages, i.e. MPI_Isend and MPI_Irecv. After each operation, MPI_Waitall is used to wait for all given MPI requests to complete.

The InOrderTags pattern communication can be described as follows: the send buffer copy is transmitted the entire data package which contains a number of messages from one node, and then sending these messages in sequence; the receive buffer on the opposite side receives all the messages in same order by using different tags; next, the second node copies the data package into its send buffer and sends this back to the first node; the first node then checks the returned data package against the outgoing messages to verify they are identical with each other. In contrast, the ReverseOrderTags patter refers to the messages sent by one node in order but matched at the target node in reverse order. The complete code is concluded in Appendix.

The sending algorithm for both InOrderTags and ReverseOrderTags pattern can be represented as the following pseude-code:

```
MPI_Barrier (MPI_COMM_WORLD)
FOR tag = 1 TO n
  MPI_Isend ( sbuf[tag], ..., tag, ...);
```

```
END FOR
MPI_Waitall (requests)
FOR tag = 1 TO n
   MPI_Irecv (rbuf[tag],..., tag, ...)
END FOR
MPI_Waitall (requests)
```

The receiving algorithm for the InOrderTags pattern can be represented as the following pseude-code:

```
IF (order == inorder)
   FOR tag = 1 TO n
      MPI_Irecv (rbuf[tag],..., tag, ...)
   END FOR
   MPI_Waitall (requests)
   FOR tag = 1 TO n
      MPI_Isend ( sbuf[tag], ..., tag, ...)
   END FOR
   MPI_Waitall (requests)
END IF
```

The receiving algorithm for the ReverseOrderTags pattern can be represented as the following pseude-code:

```
IF (order == reverseorder)
   FOR tag = n-1 TO 0
      MPI_Irecv (rbuf[tag],..., tag, ...)
   END FOR
   MPI_Waitall (requests)
   FOR tag = n-1 TO 0
      MPI_Isend ( sbuf[tag], ..., tag, ...)
   END FOR
   MPI_Waitall (requests)
END IF
```

In the Benchmark_Tag program, important variables should be defined before communication and computation. Firstly, this benchmark program requests to run on two nodes, otherwise it will abort. Secondly, three parameters within limits should be supplied as command-line arguments to run the program. Thirdly, a send buffer and a receive buffer should be created on each node to store the messages. The buffers are allocated dynamically by using malloc and their size depends on the number of messages, length of message and the size of the double type in C programming language. Fourthly, in order to store the timing results and other outputs, some arrays are created with sizes that depend on the number of iterations and the number of repetitions (defined as a constant at the beginning of program).

A timer is specified to measure the parallel execution after a barrier which guarantees that all the processes are ready to do the communication and computation. Another timer is specified just after the completion of a whole iteration which means the time for a one round-trip of the ping-pong will be measured per iteration. The outputs are passed and written to the defined array which will be printed at the end of the program. This guarantees that no extra time is spent on printing or other operations recorded between the two timers. The core part of InOrderTags code that are the "for" loops are illustrated as an example below to show the implementation of the pseudo-code. The complete code is concluded in Appendix.

```
if (order == 0){
   for(iter=0;iter<numiter;iter++) {
      Int1[rep*numiter+iter]= MPI_Wtime();
      if(rank == 0){
         for(i=0; i<nummess; i++){
            MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
         }
         MPI_Waitall((int)nummess,r,statuses);

         for(i=0; i<nummess; i++){
            MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
         }
         MPI_Waitall((int)nummess,r,statuses);
      }

      if (rank == 1){
         for(i=0; i<nummess; i++){
            MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
         }
         MPI_Waitall((int)nummess,r,statuses);
```

```
            for(i=0; i<nummess; i++){
                MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
        }
        Int2[rep*numiter+iter]= MPI_Wtime();
    }
    order = 1;
}
```

3.1.2 Mathematic Model for Tag Algorithm

In the common situation, the computing time for transferring an iteration of a data package can be divided in to two parts, the actual time for sending the messages and its latency. The latency includes the time for successful matching as well as the time for unsuccessful matching which happens in the ReverseOrder pattern. However the aim of this project is to measure the (unsuccessful) matching time, so the unsuccessful matching time should be extracted from the latency part. But the time for successful matching are still included in the latency part because for both orders, this time is the same.

Hence, in the Tag Algorithm for a certain number of messages (n) the time (IOT_n or ROT_n) for transferring a round of multi-pingpong program, in either algorithm, can be divided in three parts. The first part can be seen as latency (L) which is comprise of the time for sending header files, time for calling subroutines ,pipeline latency and the time for *n* successful numbers of matching. This part should be a constant for all the messages if the time of *n* numbers successful matching is subtracted. The second part is the actual time (S) for sending a couple of messages which should be proportional to the number of messages (n) in a data block. The third part time is the time (M) for unsuccessful matching for all the messages which equals to the number of matching (zero or $\frac{n(n-1)}{2}$) multiplies the matching time per message (M). Please note that the unsuccessful matching time per message is different in InOrderTags and ReverseOrderTags. Thus, some formulas can be described as the following:

For InOrder Communication: $$IOT(n) = L + n \cdot S \quad ①$$

For ReverseOrder Communication: $$ROT(n) = L + n \cdot S + \frac{n(n-1)}{2} \cdot M \quad ②$$

The difference between the two equations is the total time for $\frac{n(n-1)}{2}$ numbers of unsuccessful matching and it can be calculated by ②-① equations:

$$Diff(n) = \frac{n(n-1)}{2} \cdot M \quad ③$$

Hence, the matching time per message can be written in the following form:

$$M = \text{Diff}(n) \Big/ \frac{n(n-1)}{2} \qquad ④$$

In theory, the values of *IOT* are proportion to n, the trend of the values of *IOT* should be linear, whereas the values of *ROT* are proportion to n^2 and the graph should be a parabola. Moreover, the total difference (*IOT-ROT*) for a data package with the fixed size of messages is a parabola as well, and the matching time per message should be a constant.

3.1.3 Implementation

After testing the benchmark programs on Morar, the program was ported to the Blue Gene/Q, HECToR, INDY and ECDF facilities. A great number of repetitions of the test have been done to get the more robust results. Theses have then been subsequently displayed in the results tables set out Chapter 4. This section explains the compilation and execution tools for these machines, followed by some implementation details.

3.1.3.1 Compilation and Execution

This project employs various makefiles to build the MPI programs on the Linux system for these four machines respectively according to the compilers installed in the machines. For instance, Blue Gene/Q utilize IBM XL compiler by default and MPI codes written in C programming language are compiled with the mpicc command in the makefile. After successful compilation, the program can be executed by a submission script which depends on the job submission environment of each machine. These benchmark programs need to run with 2 nodes, so this option should be set in the script file..

INDY-Windows is a different operating system. The compilation of MPI jobs is assisted by two pieces of software: Microsoft Visual Studio and Microsoft HPC Pack. The current version supplied on INDY-Windows is Microsoft Visual Studio Express 2012 for Windows Desktop and Windows HPC Server 2008 R2. By setting some parameters and options in Visual Studio and the HPC pack will be connected with the compiler, and then an executable program will be built in Visual Studio files. Moreover, the execution process of the job on INDY-Windows is managed by the HPC Server job scheduler which provides a primary interface for submitting and monitoring jobs to the backend nodes. The job manager is included in the Microsoft HPC Pack which is equipped with an integrated application platform for running, managing, and developing parallel computing software [5].

3.1.3.2 Implementation

After coding the program and testing on Morar, both Tag Algorithm and Comm Algorithm programs were compiled and run on the HPC facilities (Blue Gene/Q, HECToR, ECDF and INDY). This section describes the implementation process and the analysis strategy of the data results each of the machines. More specifically, each machine has two groups of output for the two algorithms, and each group contains the time for in order communication and the time for out of order communication.

To guarantee accuracy and stability, the programs adopts three methods. Firstly, the number of repetitions of the whole program can be defined at the begging of the program as a constant. The multi-pingpong process is repeated many times by changing the number of iterations and repetitions. Moreover the whole test was repeated multiple times at various times of a day in case of some exceptional conditions of the machines such as a big program occupying the computing resources for a long time causing the network to be always in a busy state which could make the test outcomes of the programs unreliable. Secondly, an analysis method was adopted that combines the maximum-minimum method and average method together to eliminate outliers and obtains a set of reliable results. Lastly, there is a check point for each size (i.e. size is 80 or 800) with the number of messages is 45. This method is useful for checking the outcomes are stable and also helpful to reject the outlier.

Under common situation, the number of messages (No. of Mess) is set as 1, 10,20,30,40,50,60,70,80,90 and 45 (works as a check point); the length of each message (Length) is assigned value of 10,100,1000,10000,100000 and 1000000 (note: this is the number of 8-byte double-precision floating point values in each message) to cover both the small and big data sets; the number of iterations (iters) is equal to 10 and the repetition (reps) is defined as 5. Additionally, the whole test is repeated at least 5 times in various time periods. If the outcomes of tests with above value are not constant enough, some extra tests were performed.

3.1.4 Data Analysis Strategy for Tag Algorithm

After implanting numerous tests, a huge amount of data will be obtained. The majority of the of robust data should be extracted and selected. Further to this, the average value should be calculated to produce the final results. This value is named as data unit which refers to the time for processing an iteration (send and receive) in InOrder pattern (*IOT*) or time for ReverseOrder pattern (*ROT*). This section explains the analysis strategy while the results obtained from all the machines. These are discussed in nest Chapter 4. This strategy primarily includes two kinds of

analysis, direct results analysis which is based on the direct results of the program and the in-depth analysis that requires more information by further calculating of the direct results.

3.1.4.1 Direct Results Analysis

The theoretical value of *IOT* and *ROT* should increase as the number of messages grows. This is attributed to the fact that a double-precision floating-point number occupies eight bytes in the machine, the total size of data package will therefore increase when more messages are transferred. Additionally, the *ROT* should be always slightly bigger or approximately equal to *IOT*. The reason for this is the fact that some extra unsuccessful matching will occur in ReverseOrderTags communication. According to section 3.1.2 the InOrderTags has *n* times successful matching for *n* messages, but the ReverseOrderTags need to launch *n* times successful matching for *n* messages and another $\frac{n(n-1)}{2}$ times unsuccessful matching.

3.1.4.2 In-depth Analysis

On the grounds of the direct results, firstly, the in-depth analysis should subtract the *IOT* from the *ROT* to obtain the value of the difference (Time Diff) between *IOT* and *ROT*. The results refer to the time of all $\frac{n(n-1)}{2}$ times unsuccessful matching. Hence, each message's matching time (Matching Time per Mess) can be obtained though dividing the values of Time *Diff* (5th column in Table 3) by the values of Matching *Diff* (6th column in Table 3). The equation is stated in section 3.1.2:

$$M = \text{Time Diff} \Big/ \frac{n(n-1)}{2}$$

If the multiple values of M taking into account the number of messages (*No of Mess*), are similar with each other, the assumption and mathematic model of the Tag Algorithm is correct and appropriate. Further to this the average values of M are calculated as the principal outcome. It is important to note that is this analysis demanded a high accuracy of the results, because the matching time pre message is at the level of a very small time variant such as microseconds. So if the machine's performance is not stable and reliable enough, the in-depth analysis will not be proceeded.

3.2 Comm Algorithm

Generally speaking, the Comm Algorithm is analogous to the Tag Algorithm apart from the method used to force the matching order of messages. It provides a separated communicator of

every message. The communication patterns in this algorithm are labeled InOrderComms and ReverseOrderComms.

3.2.1 Benchmark_comm Code Design

Most of the variable definitions are the same as the Benchmark_Tag program, such as the send buffers, receiver buffers, output arrays etc. However, the Benchmark_comm program has some special variables. Firstly, there is an array of communicators with size equal to the number of messages per data package. Secondly, in order to create many communicators, a function of MPI_Comm_dup should be employed here. This routine duplicates an existing communicator with all its cached information. Further to this, a new communicator with the same group of processes but with a new context is applied [8]. This routine provides an effective way to build the many private communicators needed by the Comm Algorithm. Furthermore, the number of communicators can be decided by the user and specified by input parameters. This improves the efficiency of the program as only the number of communicators needed by the algorithm is created. Thirdly, as before, all the print statements are put at the end to ensure that timing would not suffer interference.

The InOrderComms and ReverseOrderComms communication patterns are similar to the InOrderTags and ReverseOrderTags patterns, except that the InOrdreComms and ReverseOrderComms patterns employ communicators to force the order of matching for messages between two nodes.

The sending algorithm for both InOrderComms and ReverseOrderComms patterns can be represented as the following pseudo-code:

```
MPI_Barrier (MPI_COMM_WORLD)
FOR c = 1 TO n
   MPI_Isend (sbuf[c], …, comm[c], …request[c]);
END FOR
MPI_Waitall (requests)
FOR c = 1 TO n
   MPI_Irecv (rbuf[c], …, comm[c], …request[c]);
END FOR
MPI_Waitall (requests)
```

The receiving algorithm for InOrderComms pattern can be represented as the following pseudo-code:

```
IF (order == inorder)
   FOR c = 1 TO n
      MPI_Irecv (rbuf[c], ..., comm[c], ...request[c]);
   END FOR
   MPI_Waitall (requests)
   FOR c = 1 TO n
      MPI_Isend (sbuf[c], ..., comm[c], ...request[c]);
   END FOR
   MPI_Waitall (requests)
END IF
```

The receiving algorithm for ReverseOrderComms pattern can be represented as the following pseudo-code:

```
IF (order == reverseorder)
   FOR c = n-1 TO 0
      MPI_Irecv (rbuf[c], ..., comm[c], ...request[c]);
   END FOR
   MPI_Waitall (requests)
   FOR c = n-1 TO 0
      MPI_Isend (sbuf[c], ..., comm[c], ...request[c]);
   END FOR
   MPI_Waitall (requests)
END IF
```

The variables and subroutines here are almost the same as those in Tag Algorithm, so they are omitted here. The core part of the InOrderTags code that are the "for" loops which are illustrated as an example below to show the implementation of the pseudo-code.

```
if (order == 0){
   for(iter=0;iter<numiter;iter++) {
      Int1[rep*numiter+iter]= MPI_Wtime();
      if(rank == 0){
```

```
            for(n=0; n<nummess; n++) {
               MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
            }
            MPI_Waitall((int)nummess,r,status);
            for(n=0; n<nummess; n++) {
               MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
            }
            MPI_Waitall((int)nummess,r,status);
         }

         if (rank == 1){
            for(n=0; n<nummess; n++){
            MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
            }
            MPI_Waitall((int)nummess,r,status);
            for(n=0; n<nummess; n++){
               MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
            }
            MPI_Waitall((int)nummess,r,status);
         }
         Int2[rep*numiter+iter]= MPI_Wtime();
      }     //end of for loop
      order = 1;
   }
```

3.2.2 Mathematic Model for Comm Algorithm

It is possible to implement an MPI library so that each communicator has its own set of message queues but it is also possible that all communicators share a single set of message queues. This work can distinguish between these possible implementations of an MPI library by measuring the matching time. In both InOrder and ReverseOrder patterns, for a certain number of messages (n) the time (ITO_n or RTO_n) for an iteration consists of two parts. The first part is the latency (L) for all the time of sending envelope, calling subroutine, pipeline latency and successful message matching. The times for InOrderComms and ReverseOrderComms are the same. The second part is the actual time (S) for transfer a data package which increases with the number of messages growth.

On one hand, if each communicator has its own set of message queues, there is no unsuccessful matching, so the formulas for Comm Algorithm should are as following:

For InOrder Communication: $$IOT(n) = L + n \cdot S \quad ⑤$$

For ReverseOrder Communication: $$ROT(n) = L + n \cdot S \quad ⑥$$

The graph of IOT and ROT values should be linear and proportionate to n, and the difference between them can fluctuate around zero or a constant. In theory, the performance and efficiency of the Comm Algorithm should be better than Tag Algorithm. As every communicator offers a single queue of the unique messagse in the communicator on both nodes, only 1 successful match will be launched and no unsuccessful matching needed to be proceeded. For n messages, InOrderTags should proceed n times successful matching while the ReverseOrderTags should proceed another $\frac{n(n-1)}{2}$ times unsuccessful matching. For the performance point of view, this behavior is not efficient. By comparison, both InOrderComms and ReverseOrderComms should only require n matches for n messages because each message exists in its own communicator. Hence, irrespective of InOrder or ReverseOrder receiving, each message just requires 1 time to find its communicator. The disadvantage of the Comm Algorithm is that it is not a memory friendly program because it consumes lots of memory to store the communicators and the process of invoking communicators may also cost time.

On the other hand, if there is only one set of message queues, the ReverseOrder needs to perform extra $\frac{n(n-1)}{2}$ times unsuccessful matching (M). The formulas can be written as:

For InOrder Communication: $IOT(n) = L + n \cdot S$ ①

For ReverseOrder Communication: $ROT(n) = L + n \cdot S + \frac{n(n-1)}{2} \cdot M$ ②

The overall performance of this situation is similar to Tag Algorithm. The values of IOT are proportion to n, the graph of IOT should also be linear, whereas the values of ROT are proportion to n^2 and the graph should be a parabola. As well as, the total difference (IOT-ROT) for a data package with the fixed size of messages is a parabola.

3.2.3 Compilation and Implementation

Given that the compilation and implementation processes for the Benchmark_comm program are exactly the same as the Benchmark_tag program. So they are omitted here.

3.2.4 Data Analysis Strategy for Comm Algorithm

There are also two kinds of analysis as referred to in the Introduction section above, direct results analysis and in-depth analysis for the Comm Algorithm. These are explained in more depth below.

3.2.4.1 Direct Results Analysis

If each communicator has its own set of message queues, the time of InOrder (*IOT*) and

ReverseOrder (*ROT*) within a certain size of the Comm Algorithm should be linear depending on to the number of messages (*No of Mess*) and *ROT* should not be greater than *IOT* because each message has its own queue in its communicator. Otherwise, if the curve of *IOT* is a straight line but *ROT* values is tending to an parabolic line and is always higher than *IOT*, the assumption is that there is only one set of message queues.

3.2.4.2 In-depth Analysis

This section needs to calculate the difference between IOT and ROT. The findings can fluctuate around zero which stands for every communicator do create a unique queue for the single message in it and no extra unsuccessful matching needs to be forced in ReverseOrder pattern. The InOrderComms and ReverseOrderComms patterns overall performance is reviewed. The difference when it is identified, could also be a constant which means there is some additional latency such as network latency and longer waiting time in the ReverseOrder pattern. This is still allowable because it can prove the assumption and the mathematic model is correct and the communicator provides every message a separated queue, but the performance of InOrder patter is better than ReverseOrder pattern. However, if the results cannot fit with the above situation, it means there is only one set of message queues.

4. Machine Configuration and Program Outcomes

The supercomputing technology in the United Kingdom has reached the world advanced level. In 2013 International Supercomputing conference in Leipzig announced the newest Top 500 Supercomputer list, and five British supercomputers place in the top 50. To be more specific, DiRAC - Blue Gene/Q, ranks 23rd [1] , which is developed by the University of Edinburgh; HECToR - Cray XE6 maintained by University of Edinburgh places 41st in the new list [1]; and two Power 775 supercomputers are separately in the 44th and 45th place which managed by ECMWF (European Centre for Medium-Range Weather Forecasts) [1]. It means that the development prospect of the High Performance Computing technology in the United Kingdom is quite broad and optimistic. Hence, this project aims to investigate the performance of the UK advanced machines. Four of the most powerful HPC facilities are chosen to perform the micro-benchmark suite. These machines include the Blue Gene/Q, HECToR, INDY (Industry machine of EPCC) and ECDF (compute component of Edinburgh Compute and Data Facility), and another machine used for experiment called Morar (computing clusters used for MSc teaching in EPCC).

This chapter describes the hardware architecture and software of the four HPC facilities at first, followed by illustrating the outcomes of the benchmark programs. All the results will be showed

in the direct results table, then choose some kinds of figures to present the result more clearly. A couple of kinds of figures can be used to describe the trend of the IOT and ROT changes within a same size. The Direct Results figures such as Figure11 show the direct results' trend in graph. The Diff figures illustrate how the gap between IOT and ROT changes with the number of messages increases. The Matching Time figures are drawn with the values of the matching time per message within the same message size. The most important figure is Final Result figure which states the matching time per message across all orders of magnitudes of the messages size, which is the expected value of this project.

4.1 Blue Gene/Q

In 2004, IBM launched a project named Blue Gene Project which is designed to develop the most powerful, most energy efficient and low power consumption supercomputers in the world. Up to now, there are three generations of supercomputers have been created, Blue Gene/L, Blue Gene/P and Blue Gene/Q.

4.1.1 Machine Configuration

The third generation in the Blue Gene series, Blue Gene/Q, is available online in 2012 which is the most power and space efficient supercomputer in the world. Blue Gene/Q is the latest supercomputer in UK which was well placed in the rankings at 24 of the worldwide Top500 list. It is a distributed collection of computers around the United Kingdom that supporting calculations world widely in particle physics, in astrophysics and other fields [2].

The Blue Gene/Q system employed in this project is DiRAC Blue Gene/Q installation. This equipment is a part of UK's DiRAC facility which is the integrated supercomputing facility for theoretical modelling and HPC-based research in particle physics, astronomy and cosmology, areas in which the UK is world-leading [10]. The DiRAC Blue Gene/Q is a joint development with DiRAC, University of Edinburgh and IBM.

The Blue Gene/Q facility consists of 6144 compute nodes and 98,304 cores in total. These 6144 nodes can be divided into two partitions bgqfe2 and bgqfe4. The bgqfe2 with 4096 nodes provides access to PreGA (pre General Availability) and bgqfe4 with 2048 nodes connected to GA partition (General Availability). Only the PreGA partition is available for the most users. The peak performance can achieve 1.26 PFlop/s.

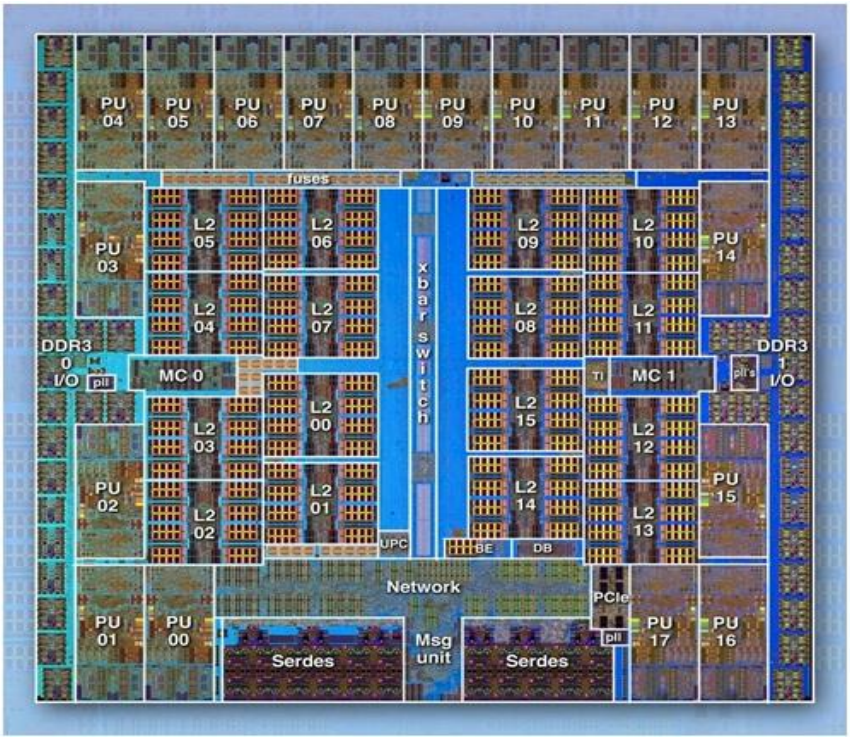

Figure 5 Hardware architecture of Blue Gene/Q

Figure 5 shows that each node has 16 cores Powerpc64 (Power ISA v2.0.6) A2 processor for application processes and an extra core take charge of operating system and interrupt handling functions. The chip die in the node only consumes 55W at the clock speed of 1.6 GHz. This processor is able to support many programming model such as POSIX, MPI, and OpenMP by using different compilers. The Powerpc64 A2 processor is capable of enhancing throughput by processing multiple independent threads (up to 64 threads on each node) simultaneously. The floating point unit in Blue Gene/Q is 4 wide double precision SIMD (single instruction multiple data) vector extensions (QPX). It conduced to 204.8 GFlop/s peak floating point performance of the chip.

Furthermore, the memory system is also a noteworthy feature of Blue Gene/Q. Firstly, each core has a 16 KB Level 1 data cache and 16 KB L1 instruction cache and many features of L1 cache can be specific by the user. In addition, this architecture has a sophisticated L1 prefetching, 16 stream and list-based prefetching. It augments the traditional stream prefetching and improves the single thread performance. Secondly, every node contains a 32 MB globally shared Level 2 eDRAM cache and the minimum bi-section bandwidth is 563 GB/s. Also, a 16GB DDR3 memory controller with a bandwidth of 42.6 GB/s exist in the chip.

The interconnect in Blue Gene/Q is a 40GB/s five-dimensional torus architecture. All the chip-to-chip communications in the 5D torus interconnect by 10 network links and each link has a peak bandwidth of 2GB/s send and 2GB/s receive. The bandwidth of communication between the 2 nearest neighbors in the 5D torus is around 1.75 GB/s per link. The shortest hardware latency

when a node communicates with the nearest neighbor is about 80 ns and the longest latency for the farthest neighbor is about 3μs. Another link is dedicated to I/O with a bandwidth of t 2.0 GB/s. Blue Gene/Q Dirac 1equiped with a 200 TB high performance parallel file system named GPFS (General Parallel File System) while the Dirac 2 mounts a 1000 TB GPFS system.

4.1.2 Program Outcomes

4.1.2.1 Outcomes of Tag Algorithm

Table 1 is the results for transferring an 80 bytes data by using Tag Algorithm on the Blue Gene/Q. Figure 6 shows how the IOT and ROT changes with number of message growth. It is easy to find that ROT is always slightly bigger or approximately equal to IOT. The trend of IOT is an almost straight line while the trend of ROT is an approximate parabola line. Figure 7 describes how Diff changes with the number of messages increases. The Diff values turn out to be a parabolic line because more messages have force to match. It is also fit with the equation ③ in section 3.1.2. Figure 8 consists of the values of Matching time, they wave slightly around 0.0223857, which means the matching time is a constant.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (μs)
80	1	0.000010	0.000010	0.000000	
	10	0.000039	0.000040	0.000001	0.0222222
	20	0.000074	0.000078	0.000004	0.0210526
	30	0.000110	0.000120	0.000010	0.0229885
	40	0.000149	0.000167	0.000018	0.0230769
	50	0.000188	0.000216	0.000028	0.0228571
	60	0.000228	0.000267	0.000039	0.0220339
	70	0.000280	0.000331	0.000051	0.0211180
	80	0.000319	0.000389	0.000070	0.0221519
	90	0.000359	0.000455	0.000096	0.0239700
Average					0.0223857

Table 1: Results of 80 bytes messages transferring by Tags Algorithm on BGQ

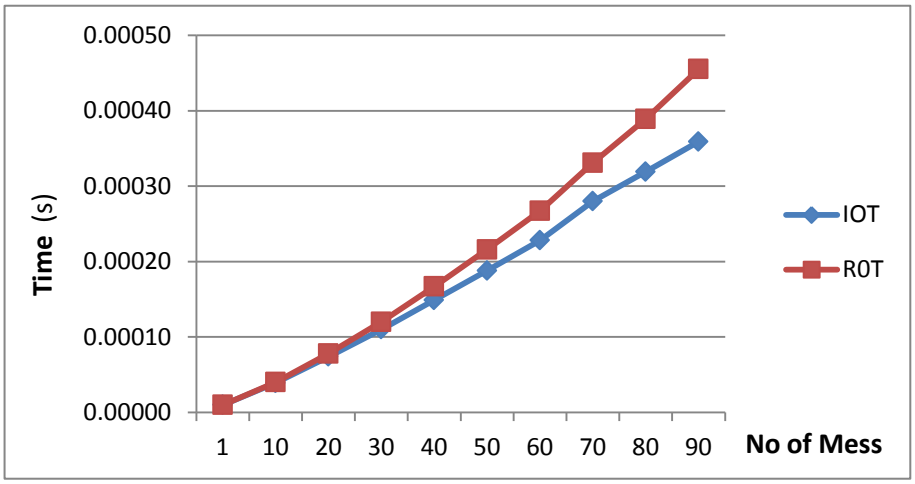

Figure 6: Direct results for 80 bytes messages transferring by Tags Algorithm on BGQ

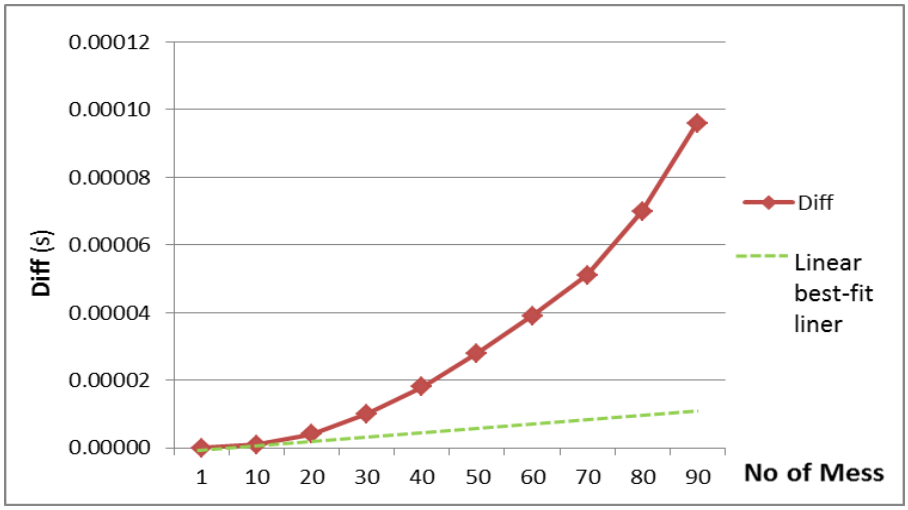

Figure 7: Diff figure of 80 bytes messages transferring by Tags Algorithm on BGQ

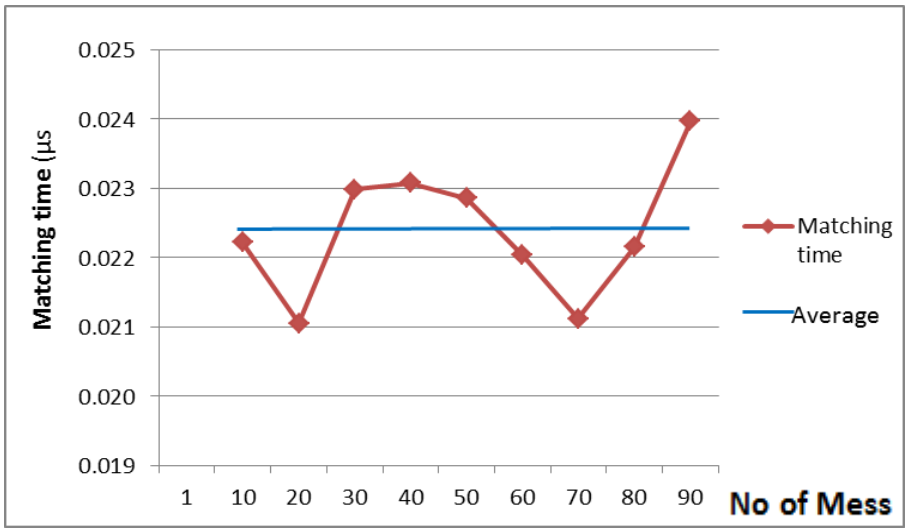

Figure 8: Matching time for 80 bytes messages transferring by Tags Algorithm on BGQ

Table 2 is the results for transferring 800 bytes data by using Tag Algorithm on the Blue Gene/Q Figure 9 is the graph for *IOT* and *ROT* and Figure 10 is the Matching Time figure shows the matching time calculated by different number of messages. The matching time is a constant with some fluctuations.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (μs)
800	1	0.000017	0.000017	0.000000	
	10	0.000068	0.000070	0.000002	0.0444444
	20	0.000128	0.000134	0.000006	0.0315789
	30	0.000188	0.000205	0.000017	0.0390805
	40	0.000248	0.000277	0.000029	0.0371795
	50	0.000308	0.000353	0.000045	0.0367347
	60	0.000369	0.000442	0.000073	0.0412429
	70	0.000440	0.000579	0.000139	0.0575569
	80	0.000501	0.000617	0.000116	0.0367089
	90	0.000561	0.000734	0.000173	0.0431960
Average					0.0408581

Table 2: Results of 800 bytes messages transferring by Tags Algorithm on BGQ

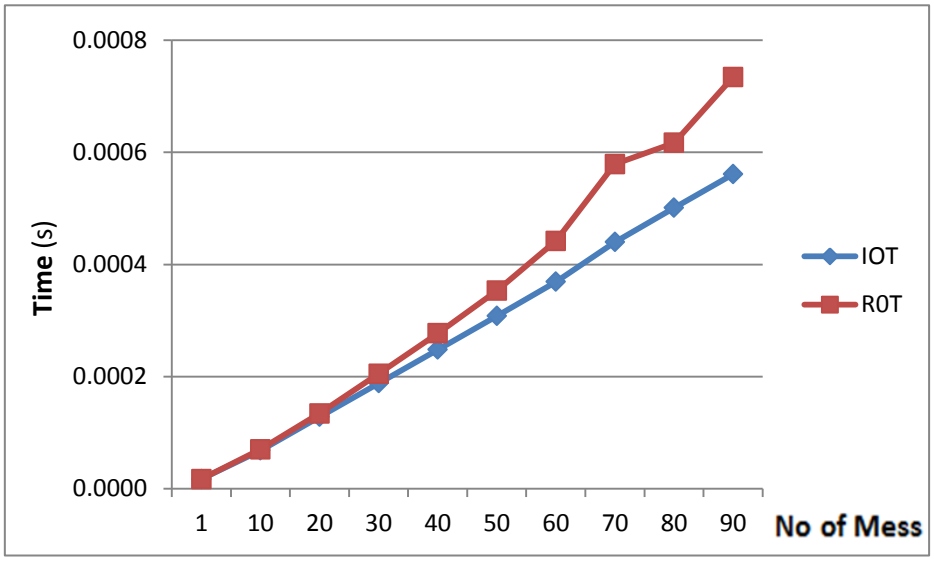

Figure 9: Direct results for 80 bytes messages transferring by Tags Algorithm on BGQ

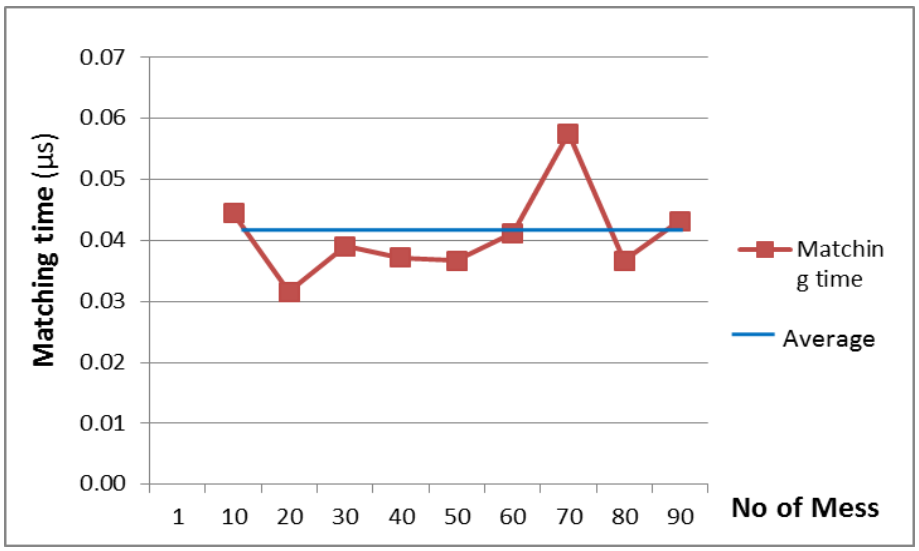

Figure 10: Matching time for 800 bytes messages transferring by Tags Algorithm on BGQ

Table 3 is the results for transferring 8000 bytes data by using Tag Algorithm on the Blue Gene/Q. Figure 11 is the Direct Results graph. All the results are reasonable.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
8000	1	0.000025	0.000026	0.000001	
	10	0.000093	0.000095	0.000002	0.0444444
	20	0.000176	0.000182	0.000006	0.0315789
	30	0.000257	0.000272	0.000015	0.0344828
	40	0.000340	0.000371	0.000031	0.0397436
	50	0.000417	0.000476	0.000059	0.0481633
	60	0.000502	0.000594	0.000092	0.0517891
	70	0.000598	0.000743	0.000145	0.0601794
	80	0.000683	0.000868	0.000185	0.0585443
	90	0.000757	0.000974	0.000317	0.0541823
Average					0.0470120

Table 3: Results of 8000 bytes messages transferring by Tags Algorithm on BGQ

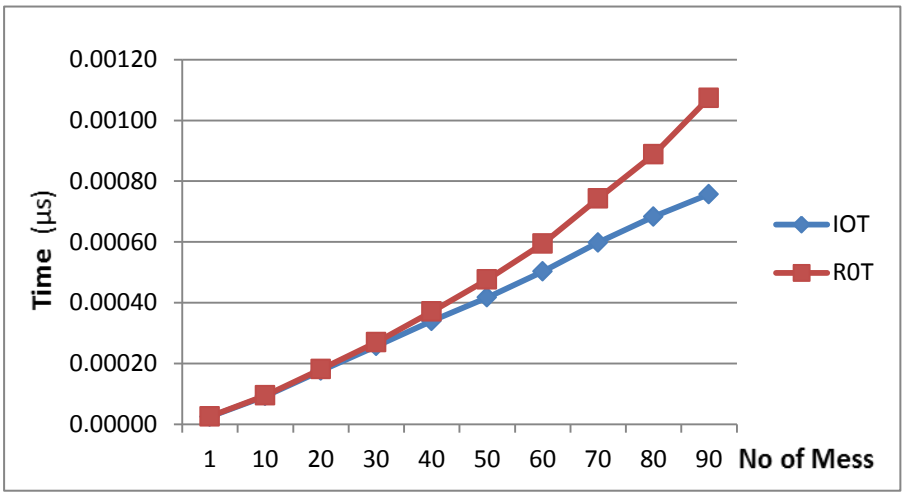

Figure 11: Direct results for 8000 bytes messages transferring by Tags Algorithm on BGQ

Table 4 is the results for transferring 80000 bytes data by using Tag Algorithm on the Blue Gene/Q and all the results are reasonable. Figure 12 is the direct results figure and Figure 13 is the Diff graph describes the gap between *IOT* and *ROT* growth. The *Diff* line seems to be parabolic whereas the dotted line which works as a linear best-fit line.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
80000	1	0.000066	0.000067	0.000001	
	10	0.000479	0.000484	0.000005	0.1111111
	20	0.000941	0.000952	0.000011	0.0578947
	30	0.001403	0.001419	0.000016	0.0367816
	40	0.001866	0.001888	0.000022	0.0282051
	50	0.002328	0.002356	0.000028	0.0228571
	60	0.002791	0.002825	0.000034	0.0192090
	70	0.003263	0.003304	0.000041	0.0169772
	80	0.003728	0.003774	0.000046	0.0146624
	90	0.004189	0.004243	0.000054	0.0134831
Average					0.0356868

Table 4: Results of 80000 bytes messages transferring by Tags Algorithm on BGQ

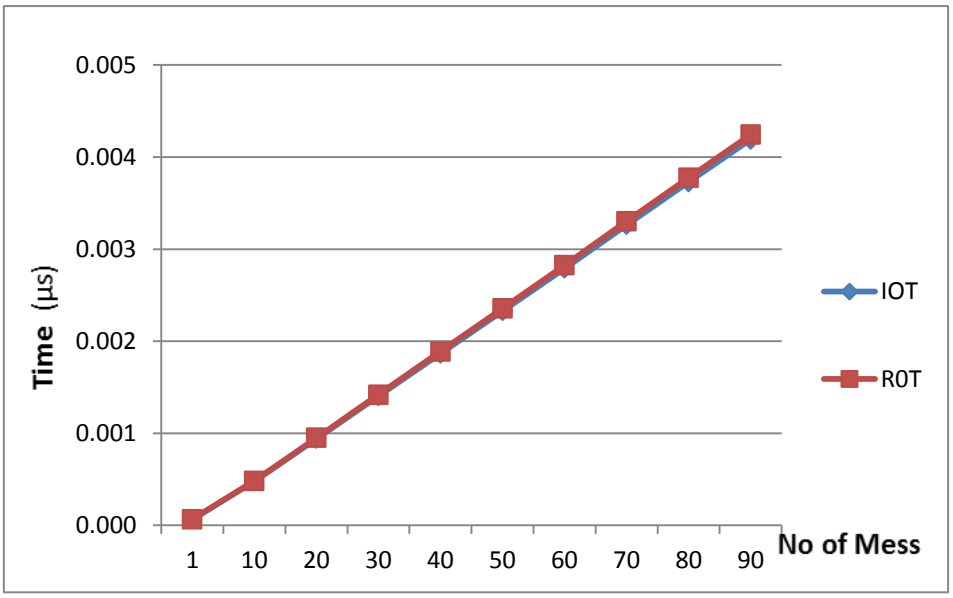

Figure 12: Direct results for 80000 bytes messages transferring by Tags Algorithm on BGQ

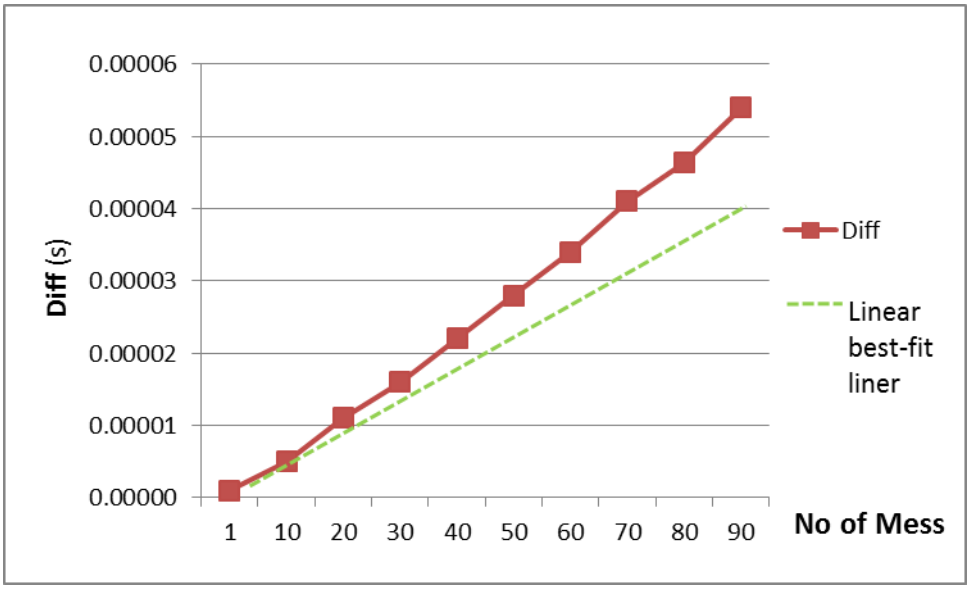

Figure 13: Diff figure of 80000 bytes messages transferring by Tags Algorithm on BGQ

Table 5 is the results for transferring 800000 bytes data by using Tag Algorithm on the Blue Gene/Q and Figure 14 is the Direct Results graph. All the results are reasonable. Figure 15 is the *Diff* graph draws the gap between *IOT* and *ROT* growth. The *Diff* line is similar to a parabola after deleting the outliners (*No of Mess* = 40,50 and 60). For the *No of Mess* = 40 data point, it is too large which means at that time the machine is quite busy, so it should be deleted. And for the *No of Mess* = 50 and 60 data point, the *Diff* is negative, that is impossible because the *ROT* should always equal to or greater than *IOT*.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
800000	1	0.000471	0.000472	0.000001	
	10	0.004529	0.004534	0.000005	0.1074074
	20	0.009054	0.009072	0.000018	0.0947368
	30	0.014902	0.014896	*-0.000006*	*-0.0145594*
	40	0.017967	0.018049	0.000154	0.1978632
	50	0.025449	0.025372	*-0.000076*	*-0.0624490*
	60	0.030076	0.030013	*-0.000063*	*-0.0354049*
	70	0.034713	0.034786	0.000073	0.0302968
	80	0.039363	0.039453	0.000090	0.0284810
	90	0.043987	0.044037	0.000050	0.0124844
Average					0.0398729

Table 5: Results of 800000 bytes messages transferring by Tags Algorithm on BGQ

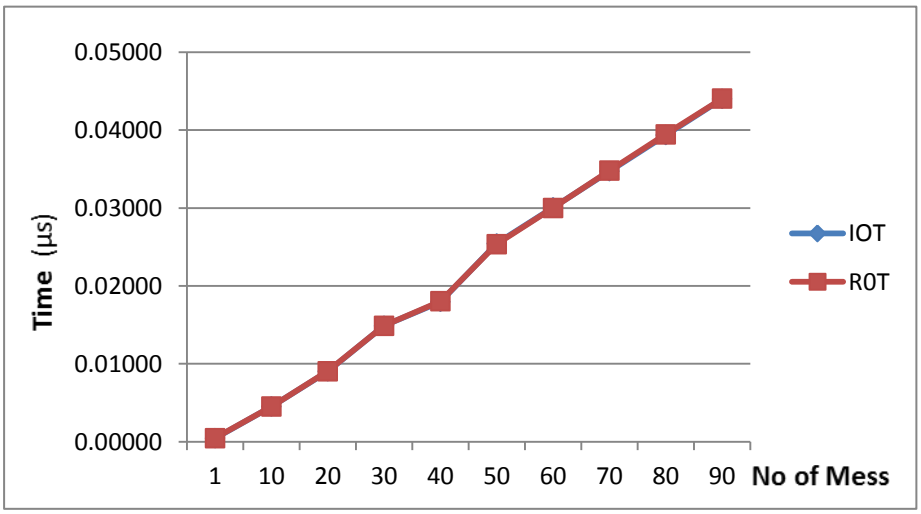

Figure 14: Direct results for 800000 bytes messages transferring by Tags Algorithm on BGQ

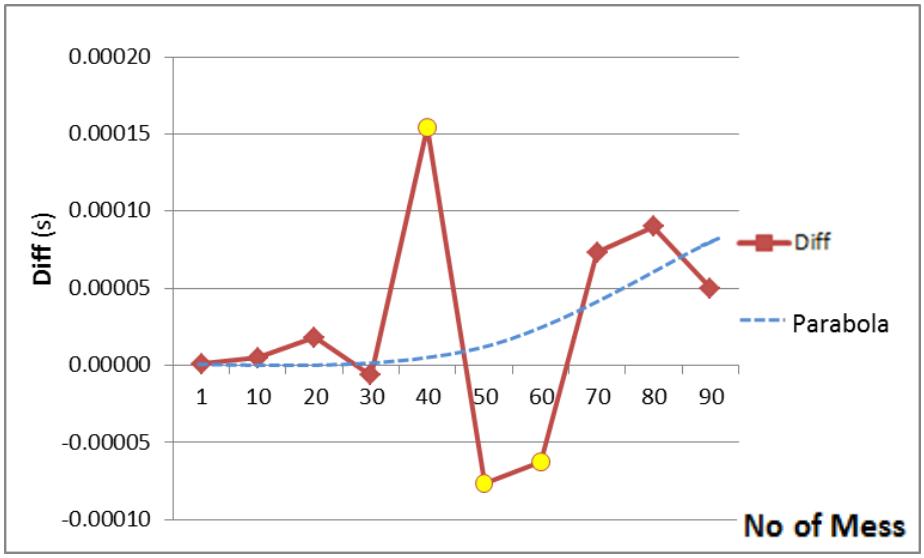

Figure 15: Diff figure of 800000 bytes messages transferring by Tags Algorithm on BGQ

Table 6 is the results for transferring 8000000 bytes data by using Tag Algorithm on the Blue Gene/Q and all the results are reasonable. Figure 16 is the graph of *IOT and ROT* (overlapped) and Figure 17 is the *Diff* graph draws the gap between *IOT* and *ROT* growth. The *Diff* line is similar to a parabola after deleting the outliners (*No of Mess* = 20 and 40).

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
8000000	1	0.004521	0.004522	0.000001	
	10	0.048558	0.048553	-0.000005	-0.1074074
	20	0.094889	0.094866	-0.000023	-0.1192982
	30	0.141237	0.141245	0.000007	0.0164751
	40	0.187415	0.187593	0.000178	0.2277778
	50	0.233983	0.234001	0.000019	0.0152381
	60	0.280288	0.280309	0.000021	0.0118644
	70	0.326609	0.326692	0.000083	0.0342305
	80	0.373000	0.373045	0.000045	0.0143460
	90	0.419215	0.419338	0.000122	0.0305035
Average					0.0137478

Table 6: Results of 8000000 bytes messages transferring by Tags Algorithm on BGQ

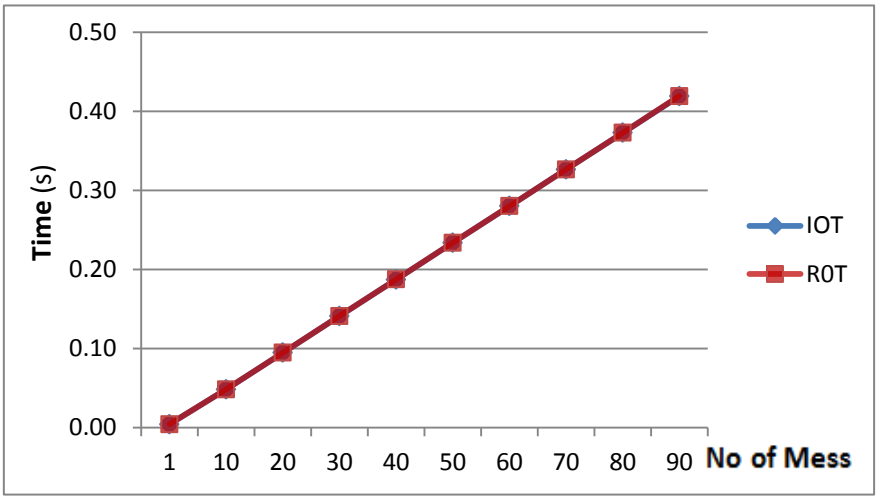

Figure 16: Direct results for 8000000 bytes messages transferring by Tags Algorithm on BGQ

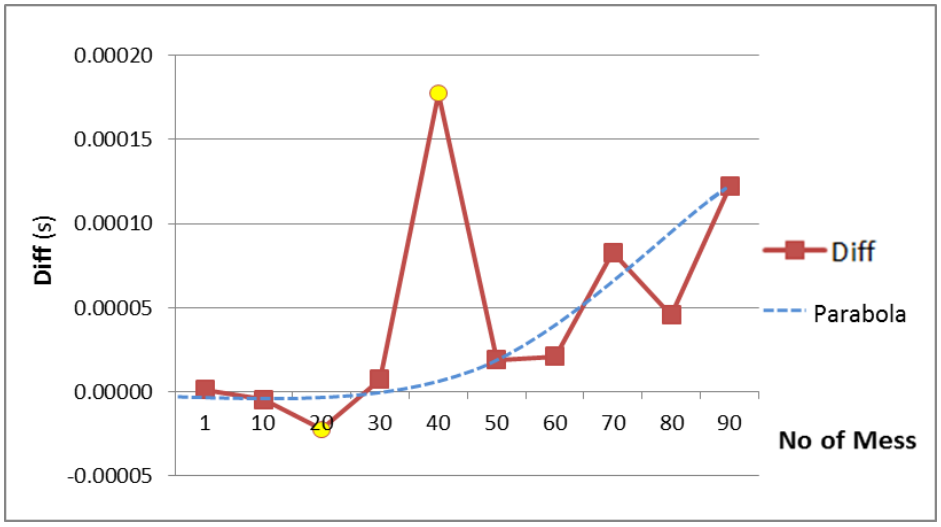

Figure 17: Diff figure of 8000000 bytes messages transferring by Tags Algorithm on BGQ

Additionally, based on the mathematic model in section 3.1.2, the equations ① and ② are helpful to resolve the values of *S* +*M* and *L* for InOrder pattern and *S, M and L* separately for ReverseOrder pattern. *L* stands for the overall latency which contains the time for successful matching, *S* is the actual time for sending messages and *M* stands for the matching time for *n* messages. Table 7 demonstrates the results of these values. The values of 8000000 bytes message are not reasonable for some machine factors, it can be ignored.

Pattern	Time (μs)	Size (bytes)					
		80	800	8000	80000	800000	8000000
InOrder (IOT)	S+M	3.921348	6.112360	8.224719	46.325843	446.134254	4459.483146
	L	6.078652	10.887640	16.775281	19.674160	24.865823	
Reverse Order (ROT)	S	3.895954	5.928902	8.013247	46.095954	447.078613	4461.424174
	M	0.028555	0.531720	0.055013	0.048555	0.055167	
	L	6.155491	11.317919	17.302445	20.864913	25.664740	

Table 7: Components of IOT and ROT

Finally, collect the averages from the six tables and form them in Table 8. Although the numbers fluctuate when the message size is 8000000, the other values are still reasonable. The exception data might due to the total size of the data package is enormous, the process of transferring messages are probably affected by machine statuses and other jobs on the machine. In order to guarantee the accuracy of the final results, this abnormal value should be eliminated. Then averaged reasonable values (apart from the value of size = 8000000), the final results appears that the matching time per massage is a constant about 0.036516 microsecond for various sizes of messages. Hence, draw a picture with the other five values in Figure 18, and the average number is drawn as a solid line and the linear best-fit line of the five values is drawn as a dotted line. We can find that they are very alike. It means that the matching time should be a constant no matter the size of the message of data package. In summary, these values of the matching time per message of Tag Algorithm are reliable, it is about 0.036516μs.

Size (bytes)	Matching Time (μs)
80	0.022386
800	0.037622
8000	0.040858
80000	0.035687
800000	0.039873
8000000	*0.013748*
Average	0.036516

Table 8: Message matching time on of Tag Algorithm BGQ

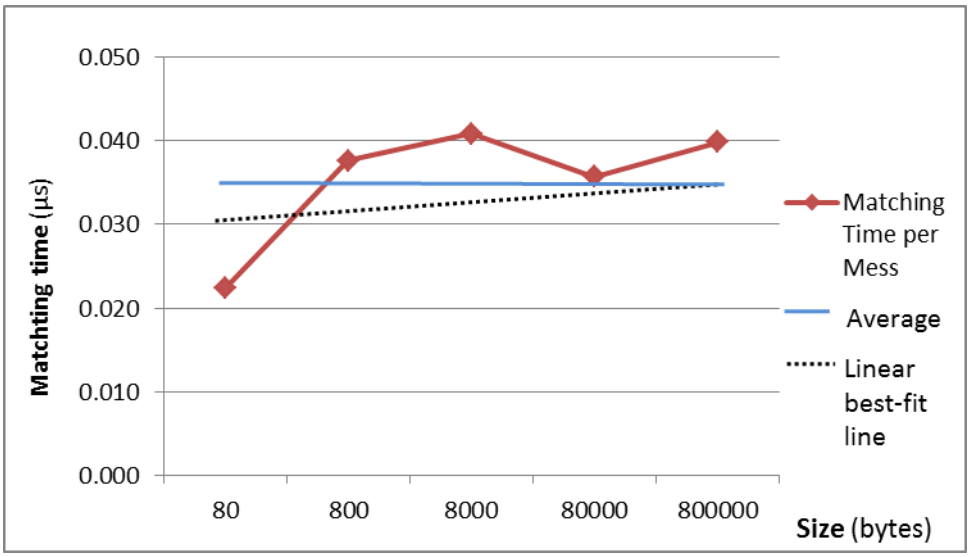

Figure 18: Massage mating time on of Tag Algorithm BGQ

It deserved to note that with the size per message increases, the line of *IOT* and *ROT* is highly overlapped with each other (i.e. Figure 14 and 16). It may triggered by two reasons: on one hand, with the size of date package increases to very huge (i.e size = 800,000 bytes), the time for matching no longer accounted for a huge share of total time because it costs a long time to transferring the big data package between node; on the other hand, as a long time is needed to sending data, it is massively more likely to be interfaced by the machines status and other network consuming jobs on the machine. That is to say, the *IOT* and *ROT* values might include a high latency and is no longer the pure time for messages sending and matching. Thus, the results of very huge data packages may not as accurate as appropriate data size, but the difference between *IOT* and *ROT* remains the actual total matching time, so the high latency cannot affect the matching time per message which is the expected result of this project. Additionally, by checking at the original results , when total size of the data package is very big, the *IOT* and *ROT* fluctuates heavily and even the *ROT* is less than the *IOT* sometimes

4.1.2.2 Outcomes of Comm Algorithm

In terms of Comm Algorithm, the Tables and Figures below shows the all results of Blue Gene/Q. Table 9 contains all the results of 80 bytes messages. Both *IOT* and *ROT* rises when number of messages and size of each message grows and the *ROT* is always lager than *IOT*. Figure 19 illustrates the *IOT* and *ROT* graph. The trend of *IOT* values is linear, however the *ROT* line seems to be a parabola. Figure 20, Diff Figure, demonstrates that fact very clear because the difference between *IOT* and *ROT* increases parabolically. Figure 21 shows the matching time changes with the number of message increase when every message size is 80 bytes.

Size (bytes)	No. of Mess	IOT	ROT	Time Diff.	No. of Match
80	10	0.000047	0.000048	0.000001	0.022222
	20	0.000093	0.000097	0.000004	0.021052
	30	0.000143	0.000151	0.000008	0.018391
	40	0.000193	0.000209	0.000016	0.020512
	50	0.000244	0.000268	0.000024	0.019591
	60	0.000295	0.000329	0.000034	0.019209
	70	0.000356	0.000402	0.000046	0.019048
	80	0.000408	0.000469	0.000061	0.019304
	90	0.000455	0.0005353	0.000080	0.020058
Average					0.019932

Table 9: Results of 80 bytes messages transferring by Comm Algorithm on BGQ

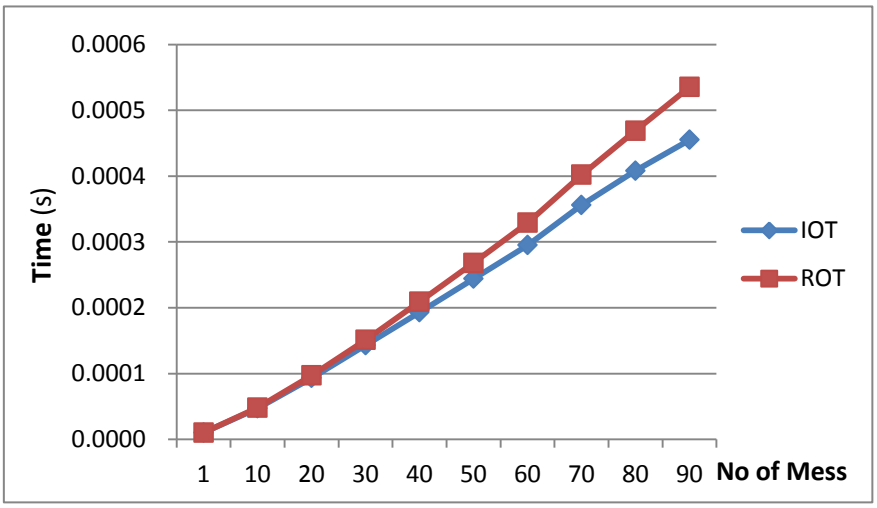

Figure 19: Direct results for 80 bytes messages transferring by Comm Algorithm on BGQ

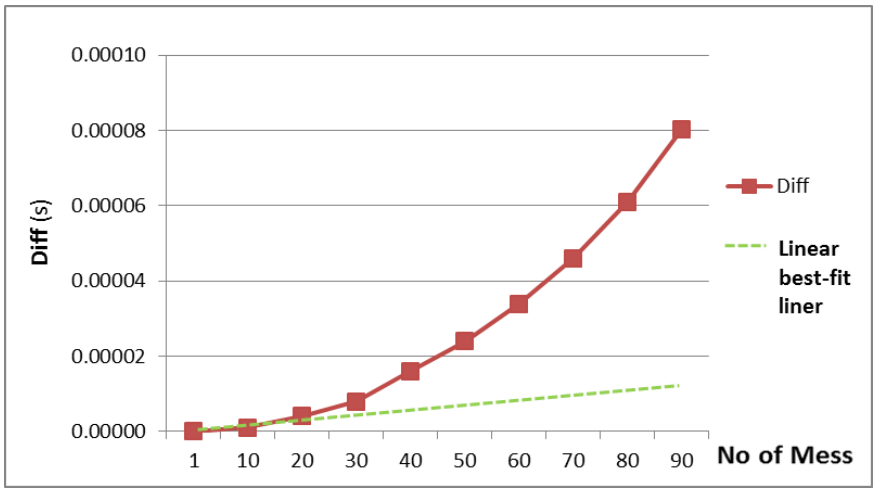

Figure 20: Diff figure of 80 bytes messages transferring by Comm Algorithm on BGQ

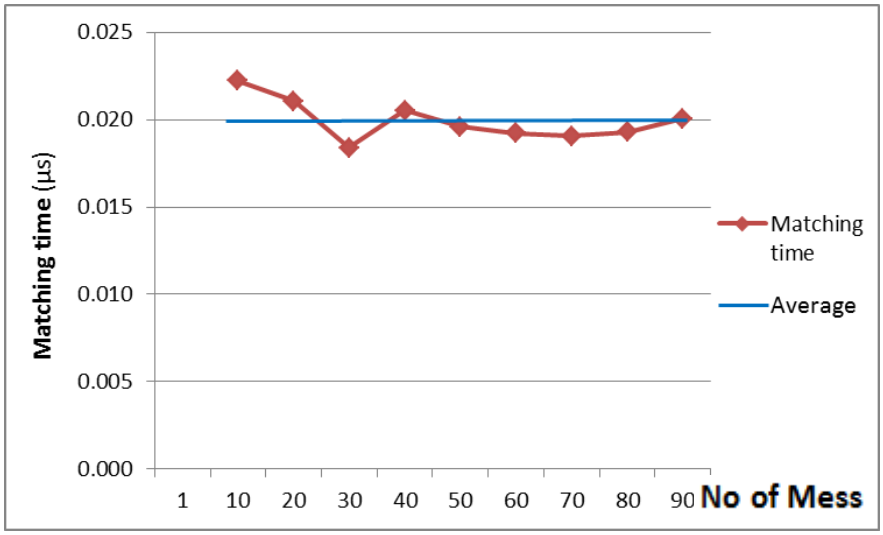

Figure 21: Matching time for 80 bytes messages transferring by Comm Algorithm on BGQ

The remaining results of the Comm Algorithm's outcomes included in Table 10, it also proves the same fact as 80 bytes message, so it will not repeat here.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
800	1	0.000018	0.000018	0.000000	
	10	0.000078	0.000079	0.000001	0.022222
	20	0.000149	0.000154	0.000005	0.026316
	30	0.00022	0.000234	0.000014	0.032184
	40	0.000291	0.00031	0.000019	0.024359
	50	0.000363	0.000391	0.000028	0.022857
	60	0.000435	0.000473	0.000038	0.021469
	70	0.000518	0.000568	0.000050	0.020704
	80	0.000589	0.000656	0.000067	0.021203
	90	0.000661	0.000740	0.000079	0.019725
Average					0.023449
8000	1	0.000027	0.000027	0.000000	
	10	0.000103	0.000103	0.000000	0.000000
	20	0.000195	0.000202	0.000007	0.036842
	30	0.000287	0.000304	0.000017	0.039080
	40	0.00038	0.000404	0.000024	0.030769
	50	0.000469	0.000503	0.000034	0.027755
	60	0.000562	0.000605	0.000043	0.024294
	70	0.000663	0.000716	0.000053	0.021946
	80	0.000758	0.000824	0.000066	0.020886
	90	0.000840	0.000920	0.000080	0.019975
Average					0.024616

80000	1	0.000068	0.000068	0.000000	
	10	0.000485	0.000487	0.000002	0.044444
	20	0.000952	0.000956	0.000004	0.021053
	30	0.001419	0.00143	0.000011	0.025287
	40	0.001887	0.001906	0.000019	0.024359
	50	0.002356	0.002385	0.000029	0.023673
	60	0.002824	0.002862	0.000038	0.021469
	70	0.003301	0.00336	0.000059	0.024431
	80	0.003769	0.003836	0.000067	0.021203
	90	0.004238	0.004315	0.000077	0.019226
Average					0.025016
800000	1	0.000473	0.000473	0.000000	
	10	0.004535	0.004538	0.000003	0.066667
	20	0.00907	0.009078	0.000008	0.042105
	30	0.014927	0.014943	0.000016	0.036782
	40	0.020594	0.020584	*-0.000010*	*-0.012821*
	50	0.025508	0.02554	0.000032	0.026122
	60	0.030145	0.030124	*-0.000021*	*-0.011864*
	70	0.034798	0.034879	0.000081	0.033540
	80	0.039443	0.039516	0.000073	0.023101
	90	0.044091	0.044172	0.000081	0.020225
					0.024873
8000000	1	0.004523	0.004523	0.000000	
	10	0.048595	0.048681	0.000086	1.911111
	20	0.094996	0.094936	*-0.000060*	*-0.315789*
	30	0.141308	0.141375	0.000067	0.154023
	40	0.187616	0.187703	0.000087	0.111538
	50	0.234118	0.234098	*-0.000020*	*-0.016327*
	60	0.280418	0.280477	0.000059	0.033333
	70	0.326844	0.32692	0.000076	0.031470
	80	0.37322	0.373256	0.000036	0.011392
	90	0.419550	0.419656	0.000106	0.026467
Average					0.216358

Table 10: Results of 800 to 8000000 bytes messages transferring by Comm Algorithm on BGQ

The *Average* values can be extracted in Table 11. We can find that the values are quite alike, except the value of 8000000 bytes message (ignored), which means the matching time in of Comm Algorithm will not change with different size of message. The Figure 22 is easier to observe this fact. The averaged value can be seen as a constant.

Size (bytes)	Matching Time (µs)
80	0.019932
800	0.023449
8000	0.024616
80000	0.025016
800000	0.024873
8000000	*0.216358*
Average	0.023577

Table 11: Matching time of Comm Algorithm on BGQ

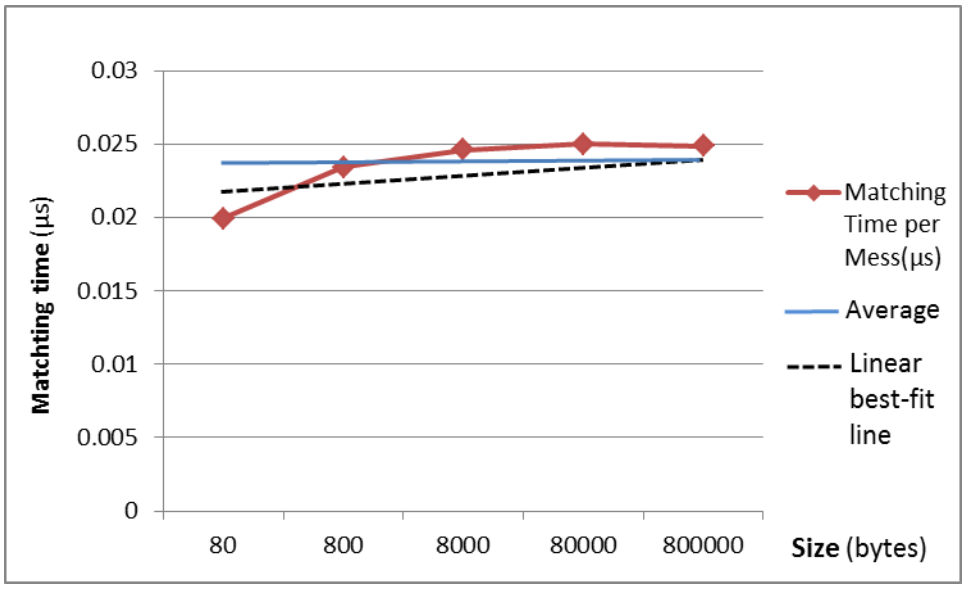

Figure 22: Massage matching time of Comm Algorithm on BGQ

It proves that the assumption that each communicator creates a unique queue for each message is wrong, all the messages communicated in different communicators still stored in the same queue. The time to force the matching order of messages remains involved in. Hence, the program cannot be optimized by using different communications, and the efficiency cannot be enhanced by multi-communicator pattern.

4.2 HECToR

HECToR is a parallel supercomputer which stands for the UK's high end computing resource, funded by the UK Research Councils [4]. It is capable of over 800,000,000,000,000 calculations per second that serves both academia and industry field in the UK and Europe. For example, recently the scientists come from The University of Manchester, University of Oregon and Yale has been successfully using HECToR to simulate how dinosaurs moved [4]. The development process of HECToR are divided into 4 phases, Phase 1 (Cray XT4), Phase 2a (Cray XT5), Phase2b (Cray XE6) and Phase 3 (Cray XE6). This chapter mainly introduces the Cray XT4 and Cray XE6 because Cray XT5 were modified slight based on Cray XT4.

4.2.1 HECToR Configuration

The current Phase 3 hardware configuration of HECToR is Cray XE6 system came online in Dec 2011. This system is composed of 30 cabinets and compute blades in total. There are fore compute nodes in every blade which result in a total of 2816 compute nodes. Each node contains two 16-core AMD Opteron 2.3GHz (Interlagos) processors, so it means that there are 90,112 cores in total. Each processor built with 32 GB main memory shared between 32 cores by using the SMP architecture and cc-NUMA architecture, which amounts to a system total of about 90 TB. The theoretical peak performance of Cray XE6 can reach 827 TFlop/s (Figure 23) [4].

HECToR Phase 3 utilizes Cray Gemini Interconnect whose benefits are high bandwidth, low latency and good overlap of computation and communication. It supports 2 nodes per network chip and each dual-socket node is interfaced to Gemini interconnect through HyperTransposrt 3.0(HT3) technology. The Gemini chip contains 10 network links to implement a 3D-torus of processors. There are also 16 blades serves as login nodes, I/O nodes and network controllers. The MPI point-to-point bandwidth of Gemini system is around 5 GB/s and the latency between two nodes is about 1-1.5μs [4].

The Cray XE6 has a shared, high-performance parallel filesystem whose high-performance RAID disks are over 1 PB. All the compute nodes can access to the disks and read and write to the distributed parallel file system. The backup system of Cray XE6 is a NAS space but with 70 TB of disk space to hold the user's home directory space as well as other files.

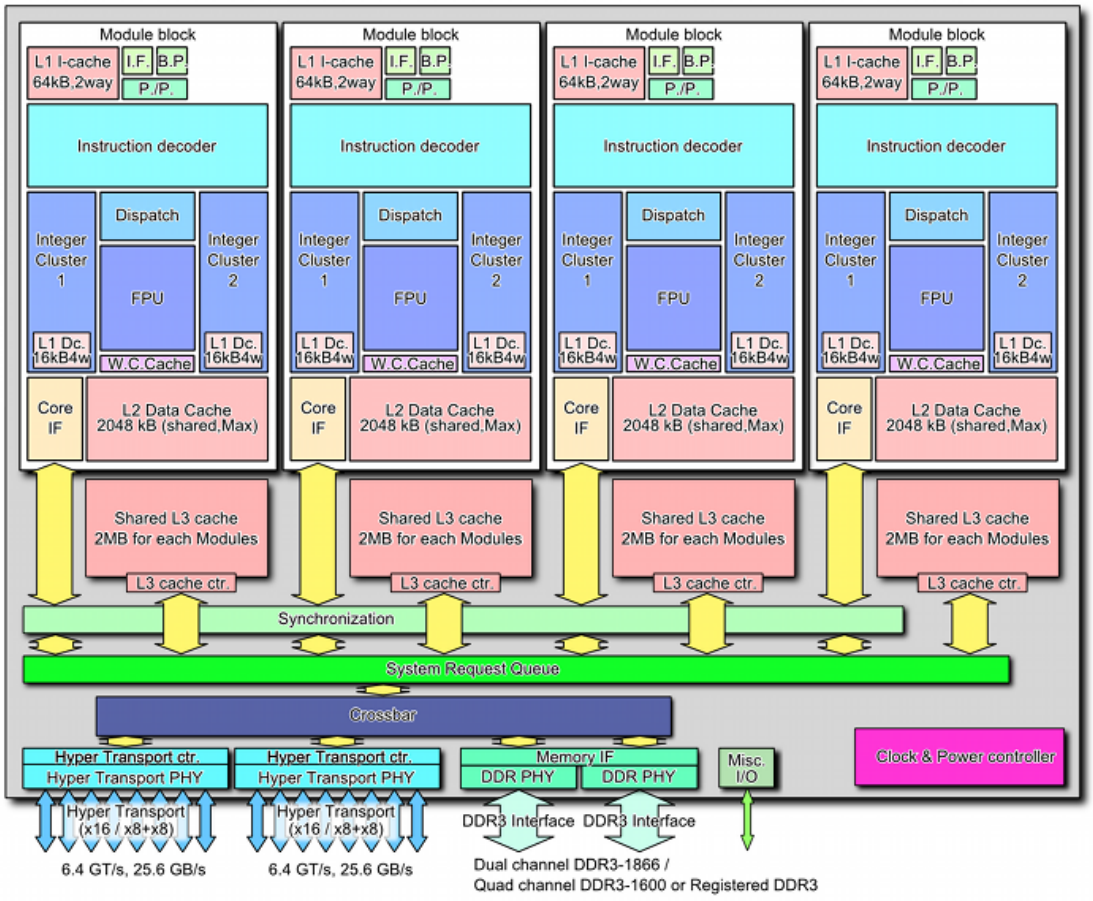

Figure 23: Hardware Architecture of HECToR

A wide range of software are currently installed in HECToR Phase 3, such as operating system, libraries, third-party applications, modules and compilers and some tools to satisfy the need of multipurpose applications. This paragraph will introduce some major or default configurations in this facility as well as the softwares require to be used in this project, detailed application lists and versions can be found at the HECToR homepage [http://www.hector.ac.uk/howcan/software/]. HECToR is running a CLE operating system, and it is divided into two partitions to obtain a better performance. A full-featured Linux distribution runs on the services nodes and login nodes. A reduced version of Compute Node Linux (CNL) runs on all the compute nodes. There are many different kinds of numerical, data and parallel libraries exist in HECToR. One of the most important libraries is the Cray Scientific Computing Library which is usually loaded by default with the appropriate PrgEnv module. This program uses cray-mpich2 library to implement the MPI program. Various kinds of third-part applications have run successfully on the supercomputer. For example, NWChem is used for Gas-phase Electronic Structure, Amber assist Classical molecular simulation and so on and so forth. In terms of compilers, three kinds of compilers are available in this facility for either Fortran or C programming, PGI, GNU and Cray compilers. It is easy to use the Cray compiler wrappers cc or CC to compile MPI code without specifying any headers or libraries. Eventually, some performance analysis tools also included here such as TAU and Scalasca [4].

4.2.2 Programs Outcomes

This section firstly shows the results of HECToR by the tables below, and then evaluates the performance issues by these figures. As the same facts with Blue Gene/Q, the total time for sending a data package of either pattern for an iteration is decided by the number of messages and size of each message.

4.2.2.1 Outcomes of Tag Algorithm

Table 12 is the results for transferring an 80 bytes data by using Tag Algorithm on the HECToR. Figure 24 shows how the *IOT* and *ROT* changes with number of message growth. It is easy to find that. *ROT* is always approximately equal to or slightly larger than *IOT*. The trend of *IOT* is an almost straight line but the trend of *ROT* is an approximate parabolic line. Figure 25 describes how *Diff* changes with the number of messages increases. The shape of *Diff* values turn out to be a parabola because more messages have force to match. It is also fit with the equation ③ in section 3.1.2. Figure 26 includes the values of Matching time, they wave slightly around 0.011439, which means the matching time is a constant. The matching time is high when there are 10 messages, it dues to the total size of the data package is too small to measure, but it does not impact the averaged value seriously.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (μs)
80	1	0.000004	0.000005	0.000001	
	10	0.000013	0.000014	0.000001	0.022222
	20	0.000024	0.000026	0.000002	0.010526
	30	0.000033	0.000037	0.000004	0.009195
	40	0.000044	0.000052	0.000008	0.010256
	50	0.000054	0.000065	0.000011	0.008980
	60	0.000066	0.000084	0.000018	0.010169
	70	0.000079	0.000104	0.000025	0.010352
	80	0.000087	0.000121	0.000034	0.010759
	90	0.000100	0.000142	0.000042	0.010487
Average					0.011439

Table 12: Direct results for 80 bytes messages transferring by Tag Algorithm on HECToR

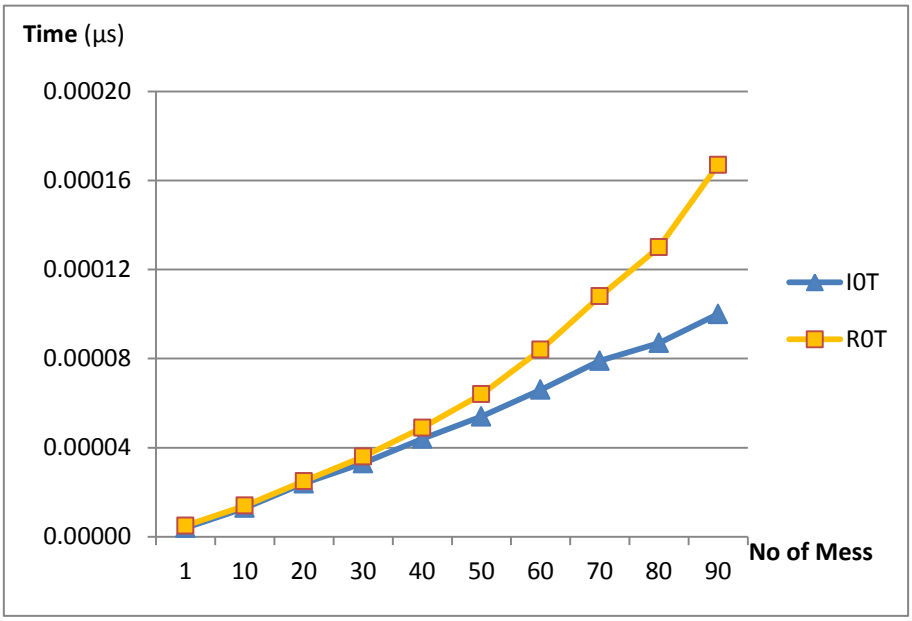

Figure 24: Direct results for 80 bytes messages transferring by Tag Algorithm on HECToR

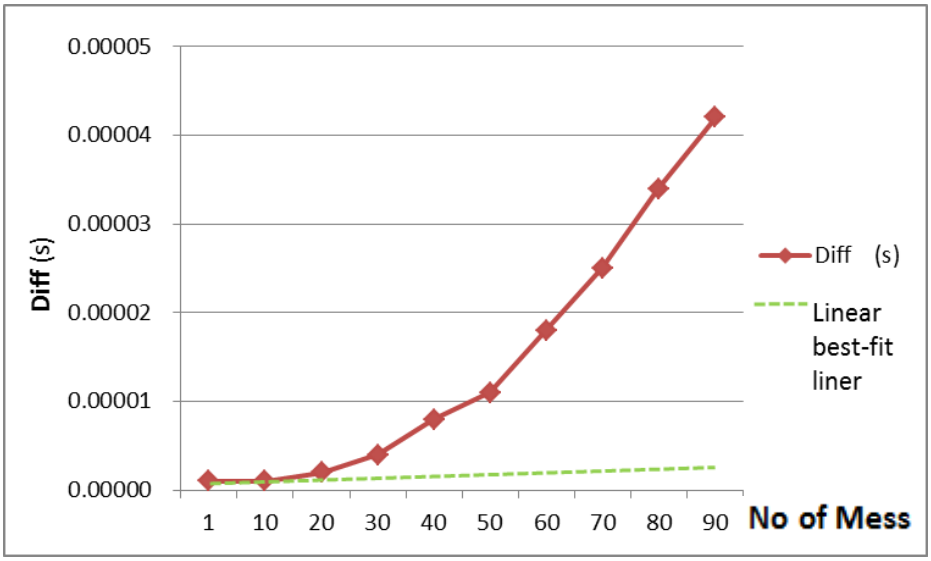

Figure 25: Diff figure for 80 bytes messages transferring by Tag Algorithm on HECToR

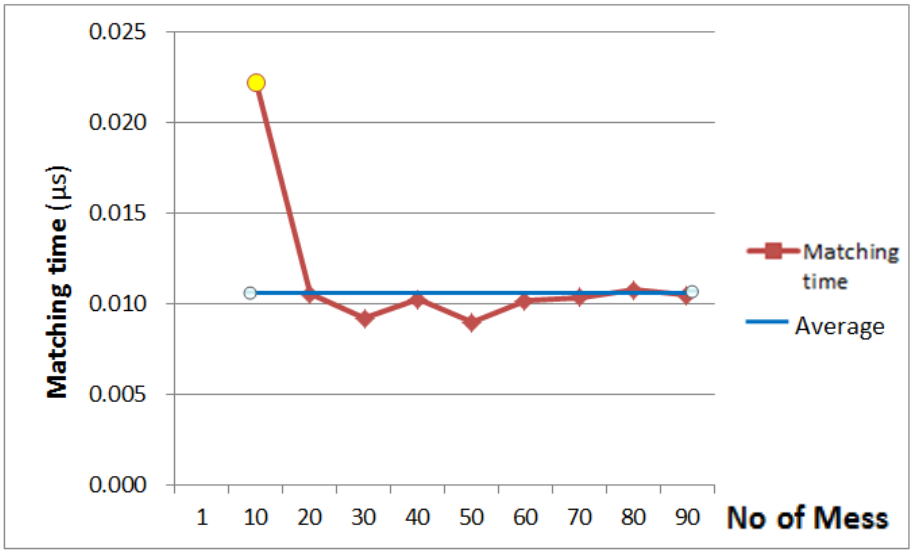

Figure 26: Matching time for 80 bytes messages transferring by Tag Algorithm on HECToR

As HECToR's results share the same pattern with Blue Gene/Q, no more figures used here to illustrate the *Diff* line and Matching time line for every size message. Table 13 is the results for transferring an 800 bytes data by using Tag Algorithm on the HECToR. Figure 27 is the corresponding direct results figure of *IOT* and *ROT*.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
800	1	0.000004	0.000005	0.000001	
	10	0.000014	0.000016	0.000002	0.044444
	20	0.000030	0.000033	0.000003	0.015789
	30	0.000043	0.000048	0.000005	0.011494
	40	0.000056	0.000065	0.000009	0.011538
	50	0.000069	0.000085	0.000016	0.013061
	60	0.000082	0.000109	0.000027	0.015254
	70	0.000097	0.000129	0.000032	0.013251
	80	0.000110	0.000155	0.000045	0.014241
	90	0.000123	0.000190	0.000067	0.016729
Average					0.017311

Table 13: Direct results for 800 bytes messages transferring by Tag Algorithm on HECToR

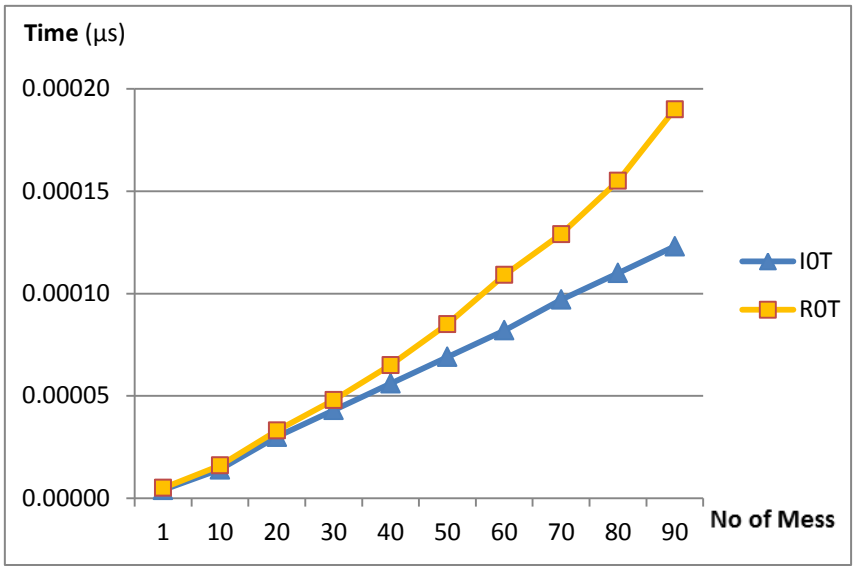

Figure 27: Direct results for 800 bytes messages transferring by Tag Algorithm on HECToR

Table 14 is the results for transferring an 8000 bytes data by using Tag Algorithm on the HECToR. Figure 28 is the corresponding direct results figure of *IOT and ROT.*

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
8000	1	0.000015	0.000016	0.000001	
	10	0.000050	0.000052	0.000002	0.044444
	20	0.000103	0.000107	0.000004	0.021053
	30	0.000141	0.000147	0.000006	0.013793
	40	0.000202	0.000213	0.000011	0.014103
	50	0.000237	0.000251	0.000014	0.011429
	60	0.000282	0.000301	0.000019	0.010734
	70	0.000324	0.000358	0.000034	0.014203
	80	0.000376	0.000432	0.000056	0.017722
	90	0.000443	0.000521	0.000078	0.019476
Average					0.018551

Table 14: Direct results for 8000 bytes messages transferring by Tag Algorithm on HECToR

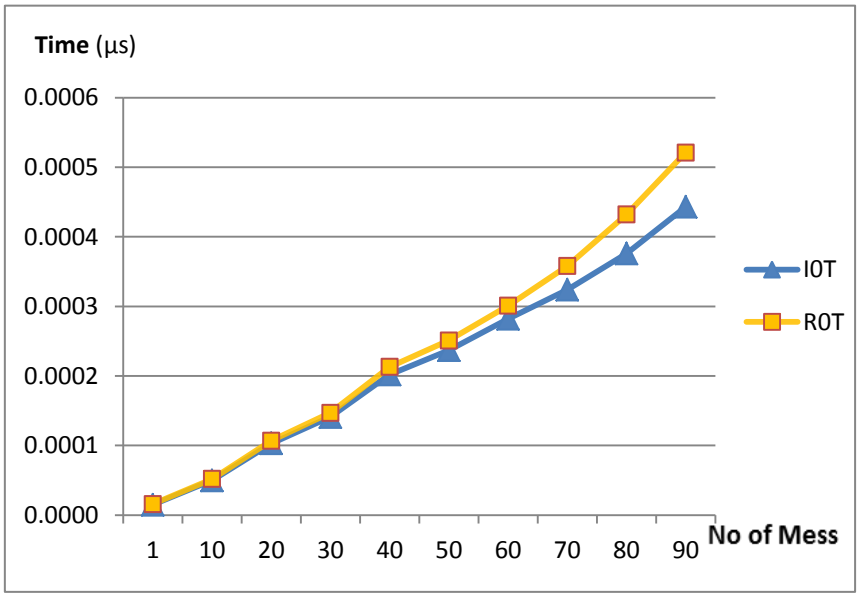

Figure 28: Direct results for 8000 bytes messages transferring by Tag Algorithm on HECToR

Table 15 is the results for transferring an 80000 bytes data by using Tag Algorithm on the HECToR. Figure 29 is the corresponding direct results figure of *IOT and ROT*.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
80000	1	0.000044	0.000045	0.000001	
	10	0.000270	0.000272	0.000002	0.050000
	20	0.000510	0.000517	0.000007	0.038596
	30	0.000756	0.000769	0.000013	0.030345
	40	0.001094	0.001112	0.000018	0.023504
	50	0.002204	0.002239	0.000035	0.028571
	60	0.002041	0.002075	0.000034	0.019128
	70	0.004339	0.004312	-0.000027	-0.011062
	80	0.002415	0.002421	0.000006	0.001989
	90	0.003652	0.003595	-0.000056	-0.014018
Average					0.018562

Table 15: Direct results for 80000 bytes messages transferring by Tag Algorithm on HECToR

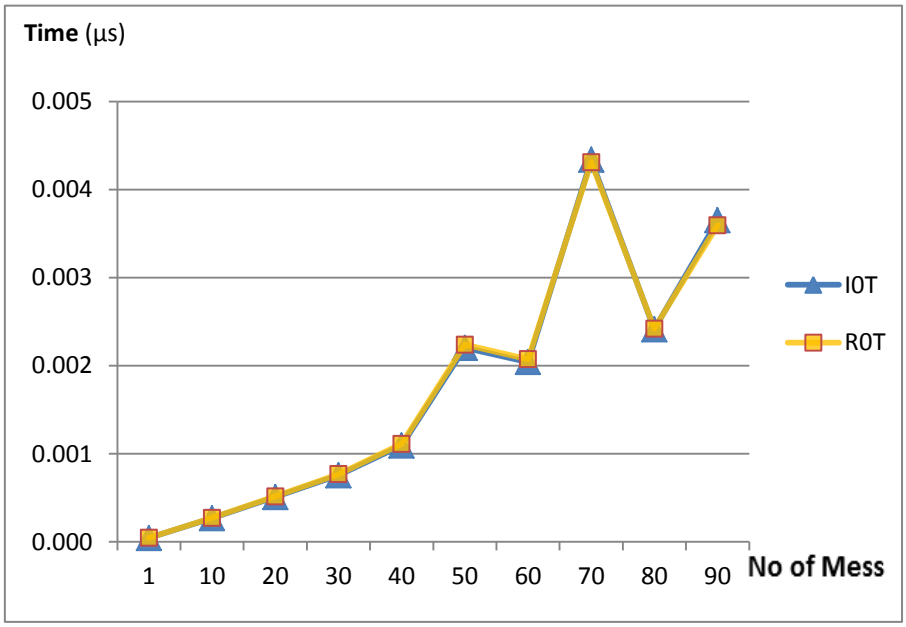

Figure 29: Direct results for 80000 bytes messages transferring by Tag Algorithm on HECToR

Table 16 is the results for transferring an 800000 bytes data by using Tag Algorithm on the HECToR. Figure 28 is the corresponding direct results figure of *IOT and ROT.*

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (μs)
800000	1	0.000326	0.000326	0.000000	
	10	0.002993	0.002996	0.000003	0.066667
	20	0.006390	0.006379	-0.000011	-0.057895
	30	0.011556	0.011581	0.000025	0.057471
	40	0.011030	0.011072	0.000042	0.053846
	50	0.014050	0.013996	-0.000054	-0.044082
	60	0.019779	0.019814	0.000035	0.019774
	70	0.028667	0.028570	-0.000097	-0.040166
	80	0.027187	0.027356	0.000169	0.053481
	90	0.033944	0.034146	0.000202	0.050437
Average					0.017726

Table 16: Direct results for 800000 bytes messages transferring by Tag Algorithm on HECToR

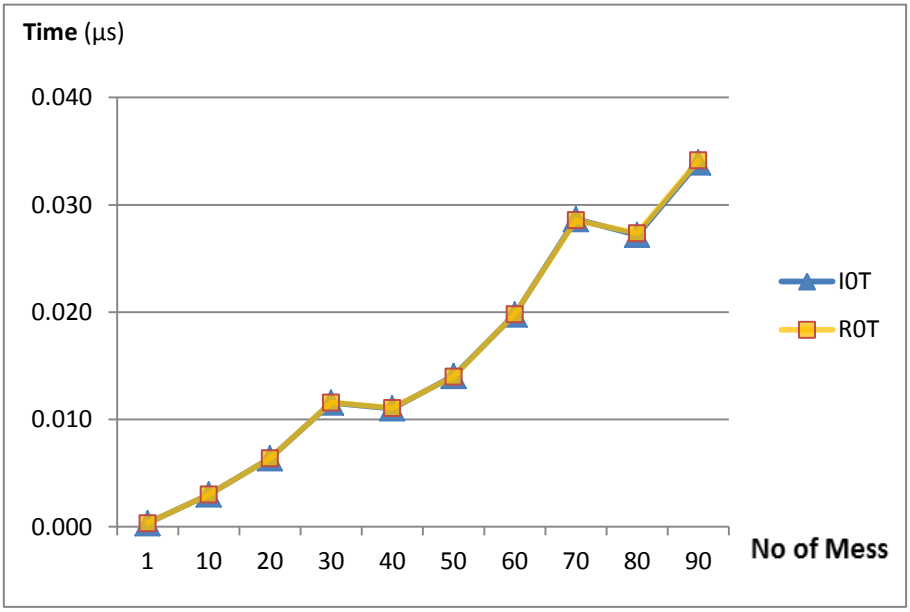

Figure 30: Direct results for 800000 bytes messages transferring by Tag Algorithm on HECToR

Table 17 is the results for transferring an 8000000 bytes data by using Tag Algorithm on the HECToR. Figure 31 is the corresponding direct results figure of *IOT* and *ROT*.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching time (µs)
8000000	1	0.002767	0.002771	0.000004	
	10	0.024675	0.024696	0.000021	0.466667
	20	0.053338	0.053387	0.000050	0.260526
	30	0.134800	0.134897	0.000096	0.221839
	40	0.102637	0.102696	0.000059	0.076282
	50	0.204592	0.204626	0.000034	0.027755
	60	0.247394	0.247512	0.000118	0.066554
	70	0.177311	0.177354	0.000043	0.017943
	80	0.279823	0.279972	0.000149	0.047310
	90	0.226005	0.226055	0.000050	0.012547
Average					0.133047

Table 17: Direct results for 8000000 bytes messages transferring by Tag Algorithm on HECToR

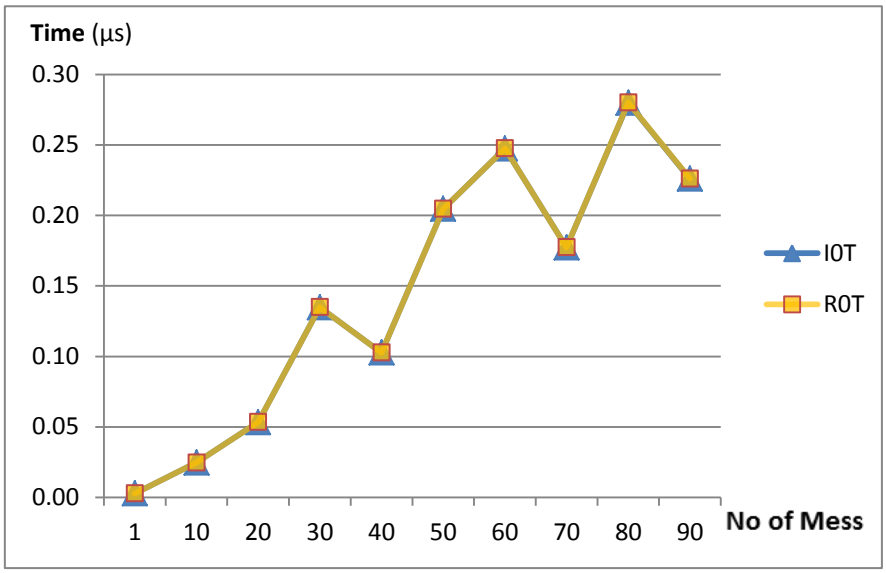

Figure 31: Direct results for 8000000 bytes messages transferring by Tag Algorithm on HECToR

According to the analysis strategy states in Chapter 3.1.2, the final results of HECToR are showed in the Table 18 which is the matching time per message in Tag Algorithm with the different size. However, the results associated with large data size might be inaccurate to some degree, such as the matching time per message with size equal to 8000000 is much smaller than the others amount to there are some negative value of the difference between *IOT* and *ROT* exist. In order to guarantee the accuracy of the final results, this abnormal value should be eliminated. Hence, draw a picture with the other five values in Figure 31, and the average number is drawn as a solid line and the trade of the five values is drawn as a dotted line. Figure 32 shows that the general trade is still stable and seems as a constant by ignoring the effect of the outliers. Hence, the mating time per message of HECToR is around 0.016718µs.

Size (s)	Matching Time per Mess (µs)
80	0.011439
800	0.017311
8000	0.018551
80000	0.018562
800000	0.017726
8000000	*0.133047*
Average	0.016718

Table 18: Message matching time for HECToR

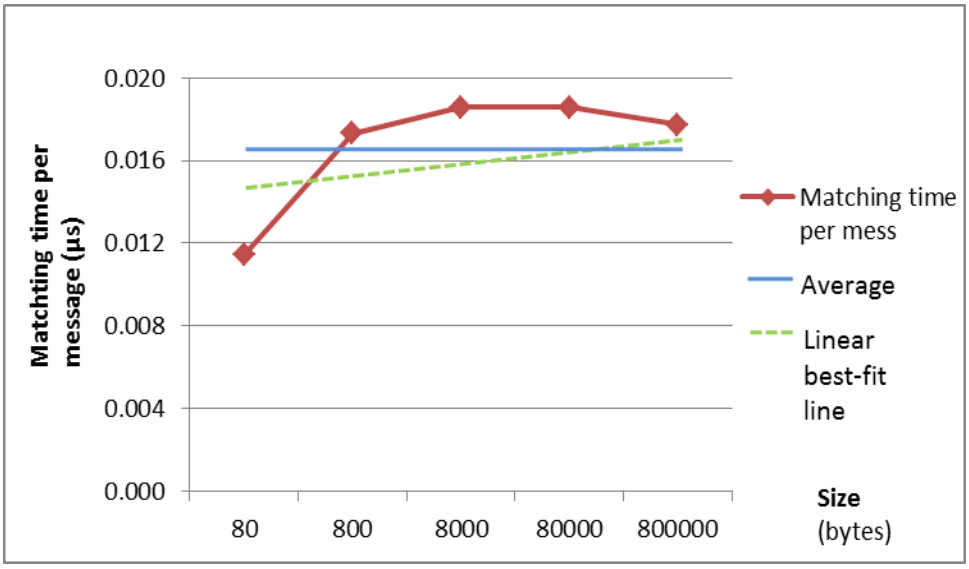

Figure 32: Message matching time for HECToR

Finally, according to the equations ①and ② in sections, Table 19 demonstrates various values includes the pure time for sending a message *(S)*, time for a message matching (*M*) and the network latency (*L*) with the size per message changes. Again, the values of 8000000 bytes should be ignored.

Pattern	Time (µs)	Size (bytes)					
		80	800	8000	80000	800000	8000000
InOrder	S+M	3.759494	5.987342	8.329114	46.215190	450.139241	*4559.483146*
	L	6.240506	11.012658	16.670886	19.784810	20.860759	
Reverse Order	S	3.695954	5.878035	8.191908	46.095954	450.078613	
	M	0.024425	0.031792	0.048555	0.048555	0.035665	*4597.500342*
	L	6.355491	11.158382	17.710983	20.855491	21.664740	

Table 19: Confirmatory results for Tag Algorithm on HECToR

In summary, the results of HECToR prove the fact that both *IOT* and *ROT* sustained increases and there is always a gap between them. Also, the fluctuations in *IOT* and *ROT* are becoming more and more heavily and a few value of the difference of *IOT* and *ROT* is negative, which stands for the data transfer process can be affected by other factors more easily.

4.2.2.2 Outcomes of Comm Algorithm

This section shows the outcomes of the Comm Algorithm programs of HECToR with multiple lengths of messages.

Table 20 lists the direct results for 80 bytes messages of Comm Algorithm. Both *IOT* and *ROT* increases with number of messages and size of each message grows and the *ROT* is always lager

than *IOT*. Figure 33 illustrates the *IOT* and *ROT* values in graph. The trend of *IOT* values is linear and *ROT* is a parabola. Figure 34is the Diff Figure, demonstrates difference between *IOT* and *ROT* increases parabolically. Figure 35 is the message matching time when the size of message is 80 bytes. The first two value (*No of Mess* = 10 and 20) is not so accurate because the total data size is so small, but they do not make a big impact on the averaged value.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
80	1	0.000004	0.000004		
	10	0.000014	0.000014	0.000000	0.000000
	20	0.000024	0.000025	0.000001	0.003008
	30	0.000034	0.000036	0.000002	0.004981
	40	0.000045	0.000049	0.000005	0.005769
	50	0.000056	0.000064	0.000008	0.006367
	60	0.000067	0.000078	0.000011	0.005989
	70	0.000081	0.000095	0.000014	0.005659
	80	0.000092	0.000111	0.000019	0.005967
	90	0.000111	0.000136	0.000025	0.006349
Average					0.005511

Table 20: Direct results for 80 bytes messages transferring by Comm Algorithm on HECToR

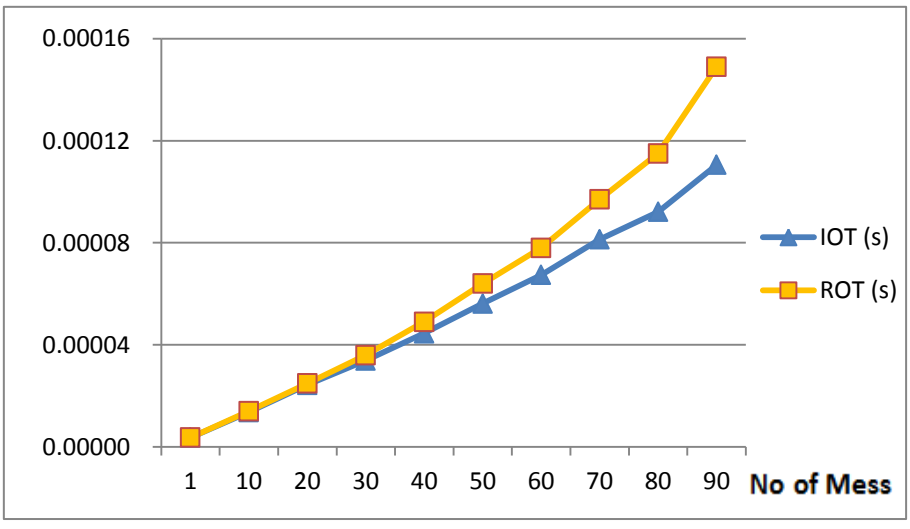

Figure 33: Direct results for 80 bytes messages transferring by Comm Algorithm on HECToR

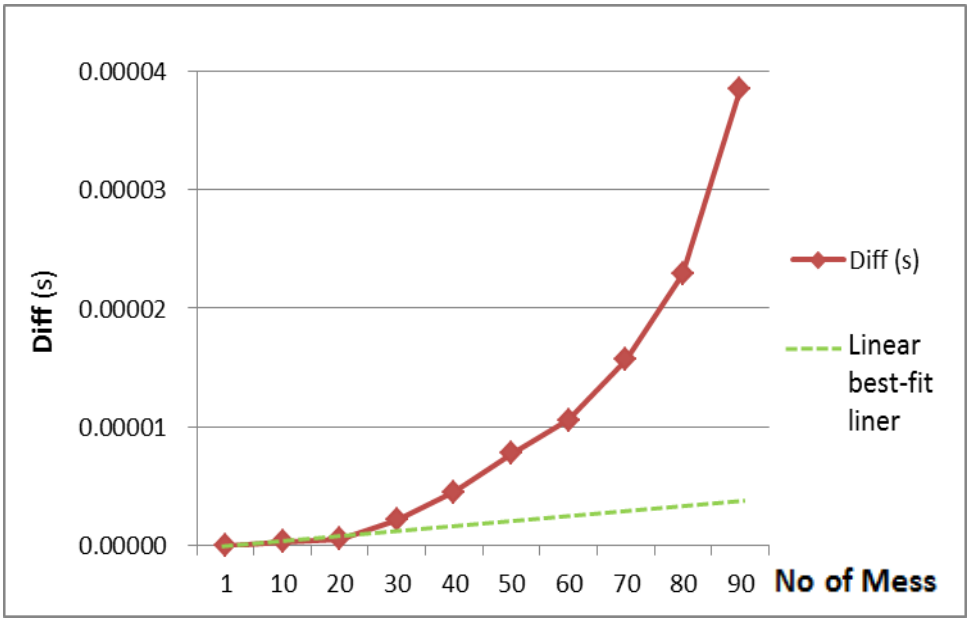

Figure 34: Diff figure for 80 bytes messages transferring by Tag Algorithm on HECToR

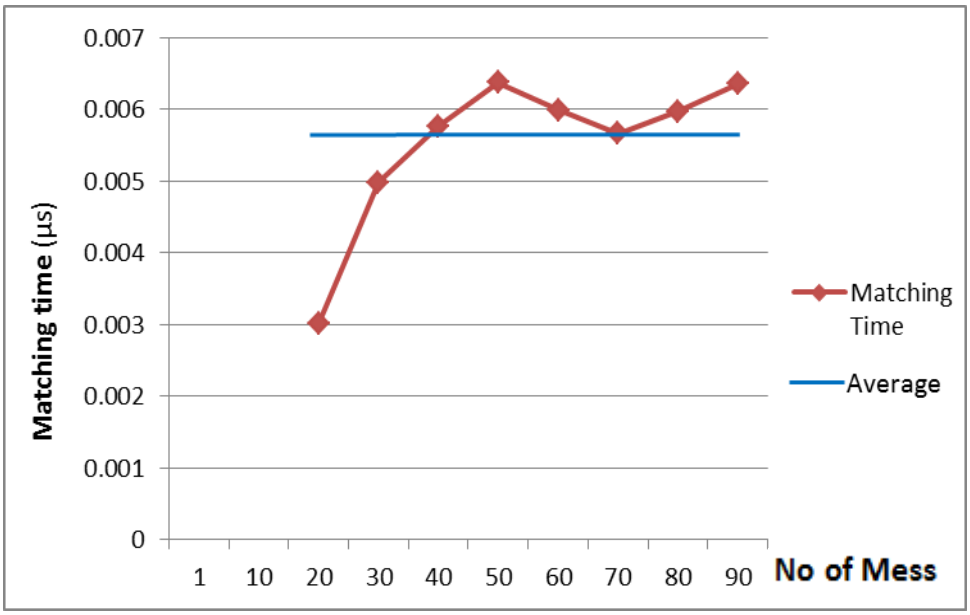

Figure 35: Matching time for 80 bytes messages transferring by Comm Algorithm on HECToR

The remaining results of the Comm Algorithm's outcomes (size = 800 to 8000000 bytes) included in Table 21, it also proves the same fact as 80 bytes message, so it will not repeat here. Only one fact needs to be noticed, the results fluctuate strongly since length equals to 10000. It means the machine is not stable and the performance cannot be guaranteed.

Size (bytes)	No. of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
800	1	0.000005	0.000005	0.000000	
	10	0.000017	0.000017	0.000000	0.000000
	20	0.000034	0.000035	0.000001	0.005263
	30	0.000043	0.000045	0.000002	0.004598
	40	0.000057	0.000061	0.000004	0.004487
	50	0.000075	0.000079	0.000004	0.003265
	60	0.000097	0.000110	0.000013	0.007264
	70	0.000101	0.000115	0.000014	0.005797
	80	0.000115	0.000141	0.000025	0.008017
	90	0.000130	0.000174	0.000044	0.011028
Average					0.005524
8000	1	0.000009	0.000010	0.000000	
	10	0.000052	0.000051	-0.000000	0.000000
	20	0.000113	0.000114	0.000001	0.003759
	30	0.000170	0.000173	0.000003	0.006130
	40	0.000226	0.000232	0.000006	0.007949
	50	0.000321	0.000329	0.000008	0.006297
	60	0.000308	0.000321	0.000013	0.007203
	70	0.000363	0.000384	0.000021	0.008765
	80	0.000411	0.000445	0.000034	0.010669
	90	0.000579	0.000636	0.000057	0.014332
Average					0.007234
80000	1	0.000047	0.000048	0.000001	
	10	0.000271	0.000272	0.000002	0.000000
	20	0.000500	0.000504	0.000005	0.023684
	30	0.000720	0.000725	0.000005	0.011954
	40	0.001376	0.001377	0.000001	0.001603
	50	0.001751	0.001748	-0.000003	-0.002721
	60	0.002069	0.002063	-0.000006	-0.003164
	70	0.001807	0.001811	0.000005	0.001988
	80	0.001952	0.001951	-0.000001	-0.000211
	90	0.002172	0.002174	0.000003	0.000642
Average					0.003753
800000	1	0.000378	0.000382	0.000004	
	10	0.002707	0.002709	0.000003	0.000000
	20	0.004790	0.004792	0.000001	0.006579
	30	0.007482	0.007488	0.000006	0.013793
	40	0.011686	0.011679	-0.000007	-0.008974
	50	0.018536	0.018555	0.000019	0.015306
	60	0.014098	0.014098	0.000000	0.000188

	70	0.021689	0.021719	0.000029	0.012127
	80	0.023061	0.023112	0.000051	0.016060
	90	0.029814	0.029808	-0.000007	-0.001706
Average					0.005930
8000000	1	0.002865	0.002890	0.000025	
	10	0.032016	0.032008	-0.000008	0.000000
	20	0.062322	0.062388	0.000066	0.348684
	30	0.094084	0.094031	-0.000053	-0.122989
	40	0.123546	0.123569	0.000022	0.028526
	50	0.127233	0.127224	-0.000009	-0.007143
	60	0.159169	0.159261	0.000092	0.051977
	70	0.198715	0.198730	0.000015	0.006211
	80	0.197474	0.197463	-0.000011	-0.003639
	90	0.220974	0.221028	0.000053	0.013358
Average					0.034998

Table 21: Results of 800 to 8000000 bytes messages transferring by Comm Algorithm on HECToR

Finally, the matching time for each message size is collected in to Table 22. It shows that the message matching time seems to be a constant with some fluctuations. The value with 8000000 bytes message is dramatic large, it should not be included in. In general, the matching time for this machine is about 0.005590 microseconds. Figure 36 illustrates the different matching according to the message size. All the values wave around 0.005590 that should be the matching time for Comm Algorithm on HECToR.

Size (bytes)	Matching Time (μs)
80	0.005511
800	0.005524
8000	0.007234
80000	0.003753
800000	0.005930
8000000	*0.034998*
Average	0.005590

Table 22: Matching time of Comm Algorithm on HECToR

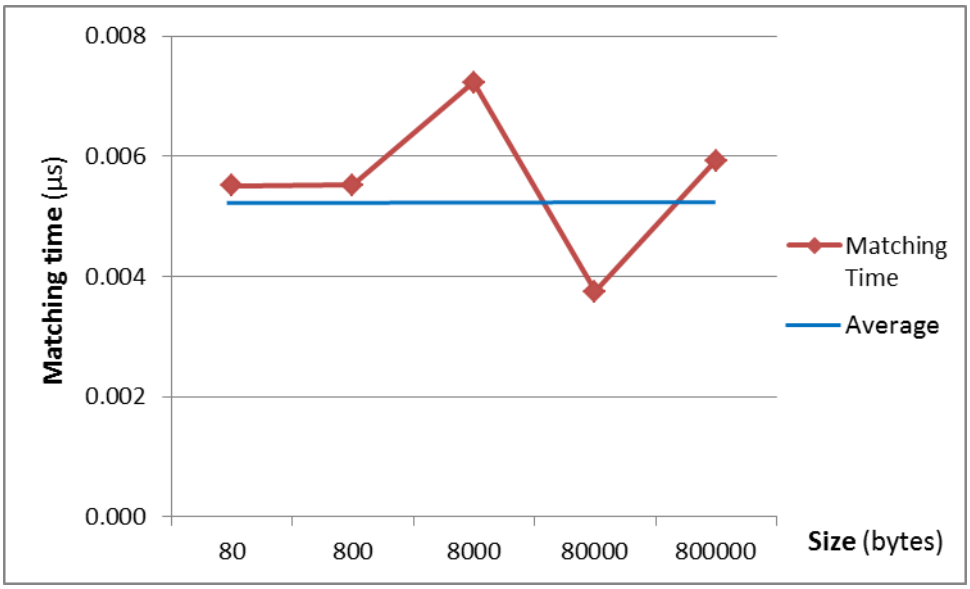

Figure 36: Massage matching time of Comm Algorithm on HECToR

4.3 INDY

4.3.1 INDY Configuration

INDY is the EPCC (Edinburgh Parallel Computing Center) Industry machine designed to meet the on-demand access and high performance computational capabilities needs of users in both the scientific and industrial area. INDY is a heterogeneous Linux and Windows high performance cluster. There are two front nodes, INDY0 which is ruing SLES 11 sp1 Linux operating system is and INDY1 which is using Windows 2008 R2 operating system. On the other hand, INDY contains two dozen 64-core backend nodes and each node has four Opteron 6276 2.3 GHz Interlagos processors which amount to 1536 cores in total [4]. Each core has 4 GB memory, giving a total of 256 shared RAM memory per backend node. There are two big memory nodes in INDY-Linux, comp000 and comp001, configured with a total of 512 GB RAM (8GB per core). All the nods are connected with each other by a very high speed and low latency Ethernet switch from Gnodal. Eventually, every node can be equipped with two Nvidia Tesla K20 GPU cards to satisfied the future need of CUDA programming. The backend nodes can be accesses through both operating systems to obtain a more efficient and stable performance [4].

In addition, INDY also works as an accelerator which provides access to some supercomputer in UK. For instance, the most advanced supercomputers HECToR and Blue Gene/Q can be augmented by the comprehensive on-demand and high performance accelerator as well.

4.3.2 Programs Outcomes of INDY-Linux

The following tables and graphs assist us to understand the overheads of messages and overall performance of INDY-Linux. Tag Algorithm is firstly implemented on the machine and followed by Comm Algorithm.

4.3.2.1 Outcomes of Tag Algorithm on INDY0

Table 23 is the results of sending and receiving 80 bytes data by using Tag Algorithm on the INDY. Figure 37 shows how the *IOT* and *ROT* changes with number of message increases. *ROT* is always approximately equal to or greater than *IOT*. The shape of *IOT* line is an almost straight while the trend of *ROT* is a parabolic line. Figure 38 describes how *Diff* changes with the number of messages grows. The shape of *Diff* lines turn out to be a Figure 39 covers the values of Matching time, they wave slightly around 0.009625, which means the matching time is a constant.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
80	1	0.000001	0.000001	0.000000	
	10	0.000007	0.000007	0.000000	
	20	0.000012	0.000013	0.000001	0.007018
	30	0.000019	0.000024	0.000005	0.011823
	40	0.000027	0.000034	0.000007	0.008425
	50	0.000034	0.000045	0.000011	0.008980
	60	0.000043	0.000059	0.000016	0.009040
	70	0.000049	0.000075	0.000026	0.010559
	80	0.000058	0.000093	0.000035	0.010918
	90	0.000066	0.000107	0.000041	0.010237
Average					0.009625

Table 23: Direct results for 80 bytes messages transferring by Tag Algorithm on INDY0

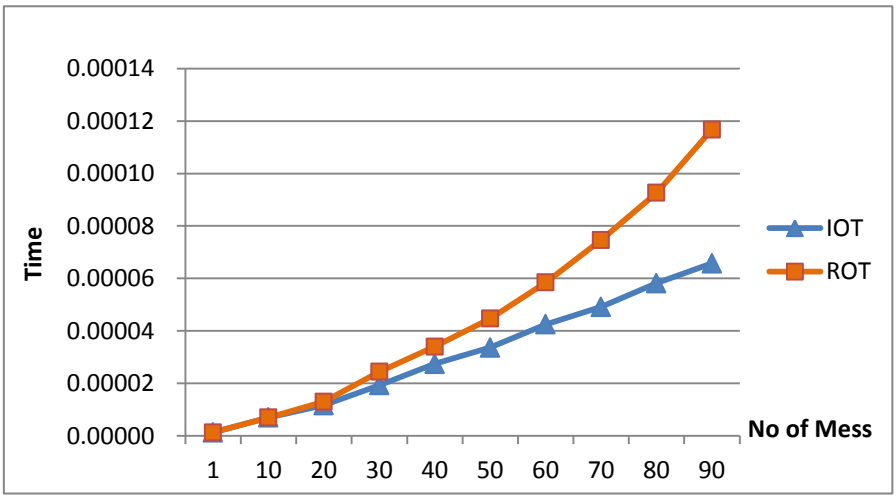

Figure 37: Direct results for 80 bytes messages transferring by Tag Algorithm on INDY0

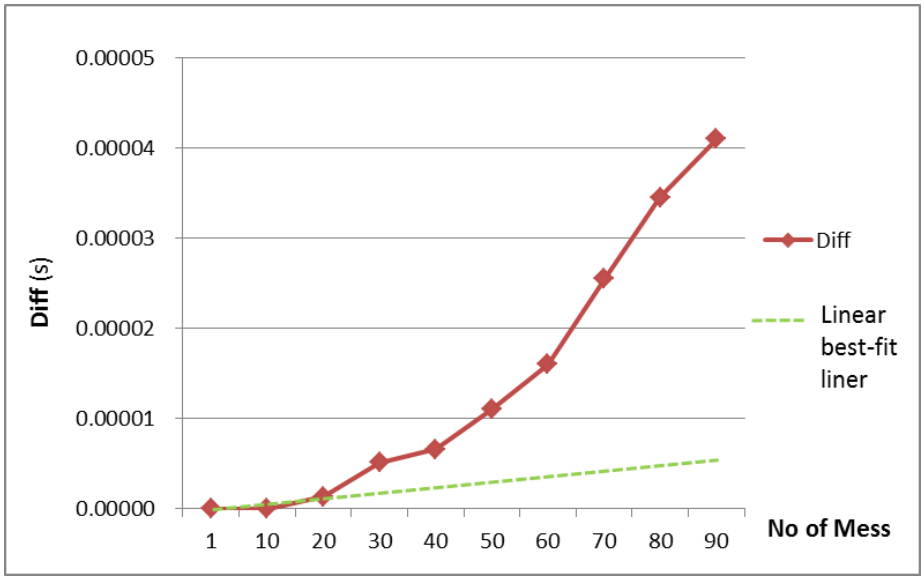

Figure 38: Diff figure for 80 bytes messages transferring by Tag Algorithm on INDY0

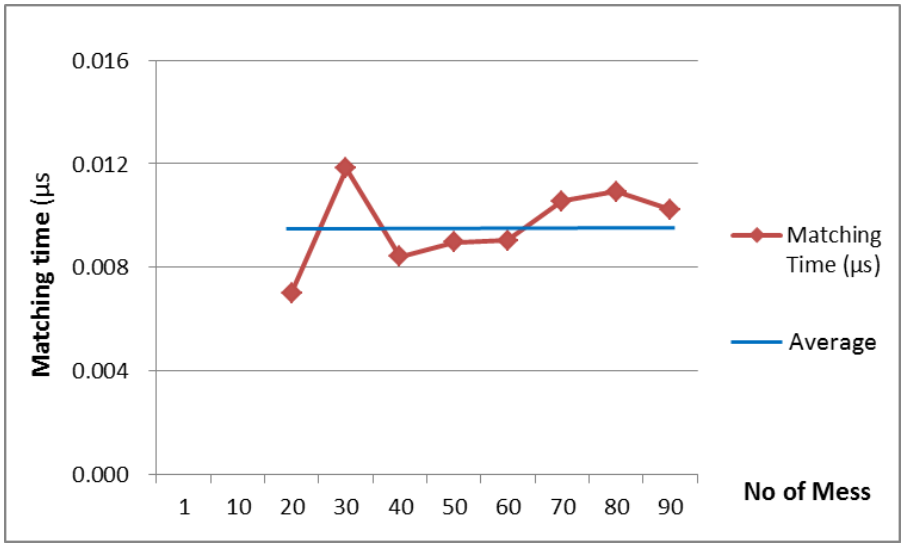

Figure 39: Matching time for 80 bytes messages transferring by Tag Algorithm on INDY0

Table 24 lists the results for 800 bytes messages sending and receiving by Tag Algorithm on INDY-Linux. Figure 40 describes how the *IOT* and *ROT* changes when more messages are transferred. All the results are reasonable.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
800	1	0.000002	0.000002	0.000000	
	10	0.000011	0.000011	0.000000	0.000000
	20	0.000022	0.000023	0.000001	0.003509
	30	0.000032	0.000035	0.000003	0.006240
	40	0.000043	0.000051	0.000009	0.010897
	50	0.000062	0.000077	0.000015	0.012245
	60	0.000072	0.000089	0.000017	0.009746
	70	0.000083	0.000110	0.000027	0.011180
	80	0.000092	0.000133	0.000042	0.013186
	90	0.000099	0.000165	0.000066	0.016438
Average					0.009271

Table 24: Direct results for 800 bytes messages transferring by Tag Algorithm on INDY0

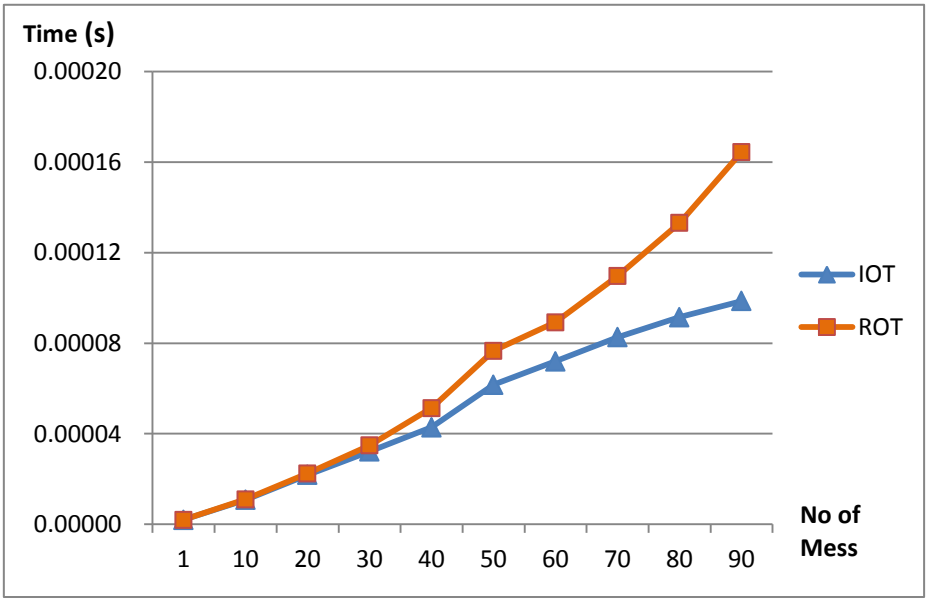

Figure 40: Direct results for 800 bytes messages transferring by Tag Algorithm on INDY0

Table 25 and Figure 41 demonstrate the *IOT* and *ROT* increase with the number of message grows when the size of each message is 8000 bytes.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
8000	1	0.000016	0.000016	0.000000	
	10	0.000119	0.000120	0.000001	0.022222
	20	0.000234	0.000235	0.000001	0.007018
	30	0.000361	0.000365	0.000004	0.009655
	40	0.000508	0.000515	0.000007	0.008547
	50	0.000589	0.000601	0.000012	0.009633
	60	0.000772	0.000789	0.000017	0.009887
	70	0.000885	0.000910	0.000026	0.010559
	80	0.001065	0.001098	0.000033	0.010506
	90	0.001061	0.001100	0.000039	0.009655
Average					0.010854

Table 25: Direct results for 8000 bytes messages transferring by Tag Algorithm on INDY0

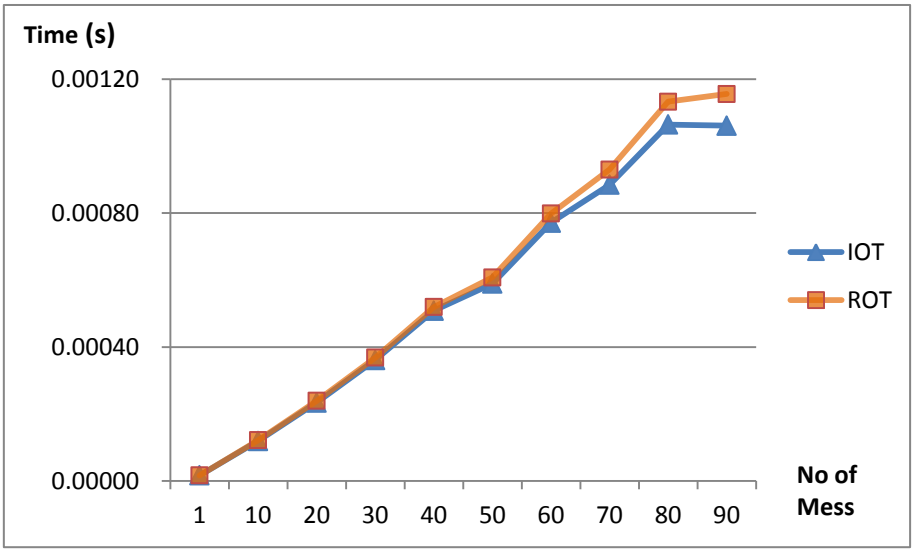

Figure 41: Direct results for 8000 bytes messages transferring by Tag Algorithm on INDY0

Table 26 lists the results for 80000 bytes messages, some outliners can be found from the *Diff* column. They should be eliminated because the *ROT* cannot less than *IOT*. Forty presents of the values of *Diff* are negative means the machine is unstable enough when a number of bytes messages transferred which is a relatively a longer time. Account for too many values are defective, this group of data cannot be adopted. Figure 42 shows the trend of *IOT* and *ROT*.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
80000	1	0.000110	0.000112	0.000001	
	10	0.001168	0.001171	0.000003	0.074074
	20	0.001856	0.001860	0.000004	0.019737
	30	0.003222	0.003220	-0.000002	-0.005255
	40	0.004574	0.004566	-0.000008	-0.009890
	50	0.005547	0.005536	-0.000010	-0.008513
	60	0.005877	0.005878	0.000001	0.000646
	70	0.007279	0.007283	0.000004	0.001656
	80	0.008257	0.008254	-0.000003	-0.000995
	90	0.008816	0.008824	0.000008	0.001962
Average					0.008158

Table 26: Direct results for 80000 bytes messages transferring by Tag Algorithm on INDY0

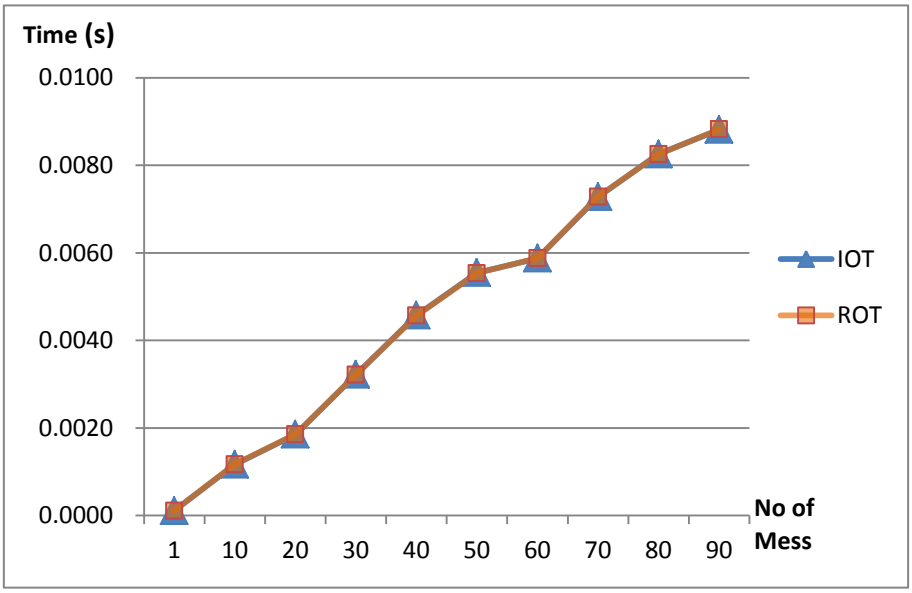

Figure 42: Direct results for 80000 bytes messages transferring by Tag Algorithm on INDY0

Table 27 and Figure 43 also state the fact that, the results are not accurate enough when a number of 800000 bytes messages transferred. Moreover the values in *Matching Time* column fluctuate dramatically, and even the average is a negative number. This also means the stability of INDY-Linux is very poor.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
800000	1	0.001207	0.001204	-0.000003	
	10	0.009752	0.009747	-0.000005	-0.114815
	20	0.020822	0.020793	-0.000029	-0.151316
	30	0.027925	0.027988	0.000063	0.145211
	40	0.034317	0.034276	-0.000041	-0.053077
	50	0.042729	0.042729	0.000000	0.000000
	60	0.051255	0.051292	0.000038	0.021328
	70	0.060453	0.060475	0.000023	0.009420
	80	0.075238	0.075381	0.000143	0.045208
	90	0.079159	0.079181	0.000023	0.005618
Average					-0.010269

Table 27: Direct results for 800000 bytes messages transferring by Tag Algorithm on INDY0

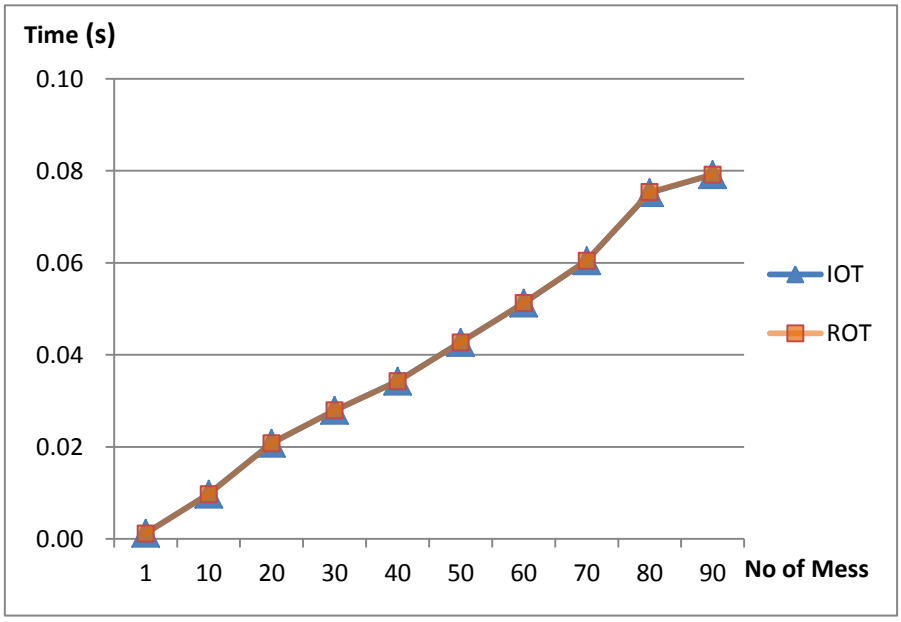

Figure 43: Direct results for 800000 bytes messages transferring by Tag Algorithm on INDY0

Again, lots of defective values appear in Table 28 and Figure 44. The average value of matching time per message results in a very big number, about 0.26 microsecond. It proves that the INDY-Linux maybe not powerful enough to perform computations with large data set.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
8000000	1	0.010273	0.010261	-0.000012	
	10	0.088977	0.089063	0.000086	1.911111
	20	0.179345	0.179336	-0.000009	-0.047368
	30	0.282066	0.282135	0.000068	0.156782
	40	0.355420	0.355519	0.000099	0.126282
	50	0.484526	0.484646	0.000120	0.098286
	60	0.539874	0.539954	0.000081	0.045537
	70	0.672511	0.672485	-0.000026	-0.010683
	80	0.743352	0.743554	0.000202	0.064030
	90	0.805049	0.805042	-0.000007	-0.001685
Average					0.260254

Table 28: Direct results for 8000000 bytes messages transferring by Tag Algorithm on INDY0

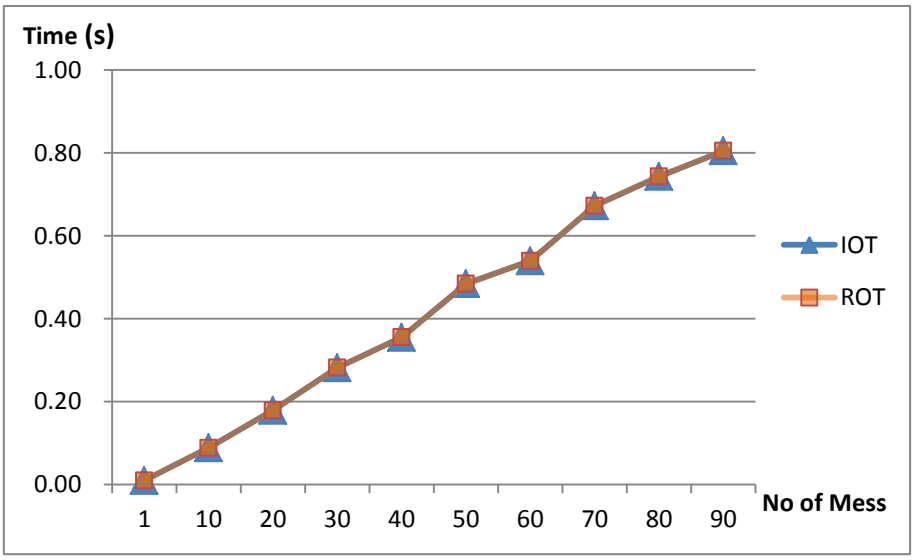

Figure 44: Direct results for 8000000 bytes messages transferring by Tag Algorithm on INDY0

As there are twenty presents of the values are not accurate, only three groups results can be used to calculate the averaged message matching time, the final results of message matching time is not quite reliable .The results matching time per message of Tag Algorithm on INDY-Linux shows in Table 29, the value of 80000,800000 and 8000000 size is unreliable so this value requires to be deleted. The approximate value of the matching time per message might be 0.009916 µs. Figure 45 draw a picture of the message matching time. In summary, the INDY-Linux is an unstable machine and large data set program cannot benefit from it.

Size (bytes)	Matching Time per Mess (µs)
80	0.009625
800	0.009271
8000	0.010854
80000	*0.008158*
800000	*-0.010269*
8000000	*0.260254*
Average	0.009916

Table 29: Message matching Time of Tag Algorithm on INDY0

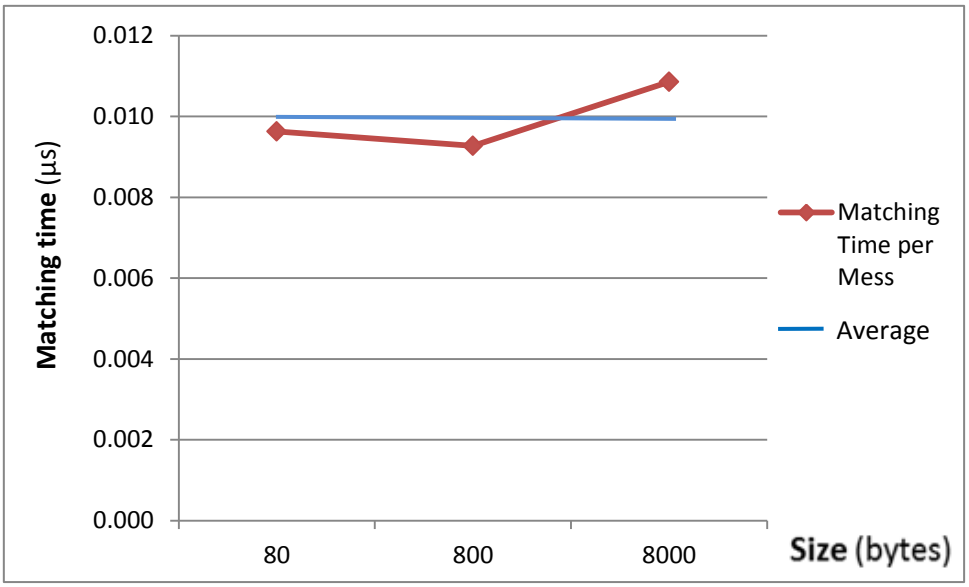

Figure 45: Approximate value of Matching time of Tag Algorithm on HECToR

4.3.2.2 Outcomes of Comm Algorithm on INDY0

Next section associated with results of Comm Algorithm. Table 30 is the direct result table of 80 bytes message associated with Comm Algorithm on INDY- Linux. Figure 46 shows *IOT* and *ROT* increases with number of messages grows. But the *IOT* is not an exactly straight line and the last point is lower than the previous. This may be caused by the machine's statue. Figure 47 describe the variation trend of *Diff* line which is a parabola with some fluctuations. Figure 48illustrate the averaged matching in this group which is a constant about 0.008809 microsecod.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
80	1	0.000001	0.000001	0.000000	
	10	0.000006	0.000006	0.000000	
	20	0.000012	0.000013	0.000001	0.006015
	30	0.000020	0.000024	0.000005	0.010509
	40	0.000026	0.000035	0.000008	0.010769
	50	0.000036	0.000047	0.000012	0.009388
	60	0.000042	0.000060	0.000018	0.010282
	70	0.000077	0.000090	0.000013	0.005569
	80	0.000093	0.000118	0.000025	0.007785
	90	0.000068	0.000109	0.000041	0.010154
Average					0.008809

Table 30: Direct results for 80 bytes messages transferring by Comm Algorithm on INDY0

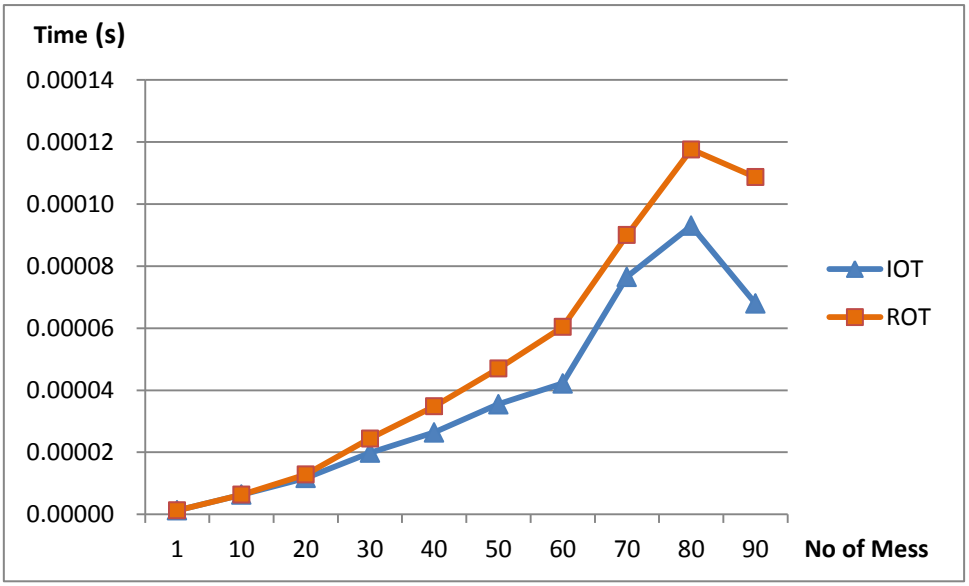

Figure 46: Direct results for 80 bytes messages transferring by Comm Algorithm on INDY0

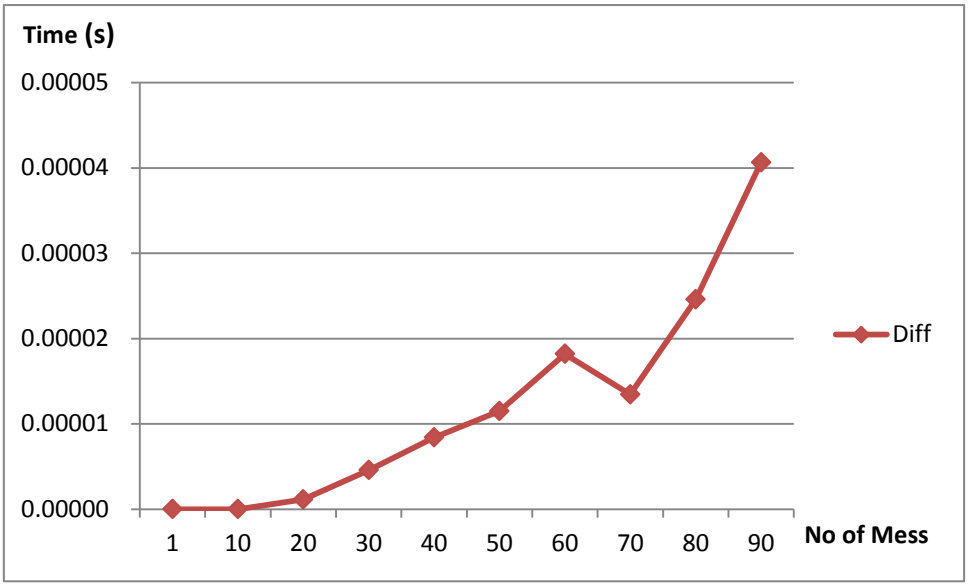

Figure 47: Diff figure for 80 bytes messages transferring by Comm Algorithm on INDY0

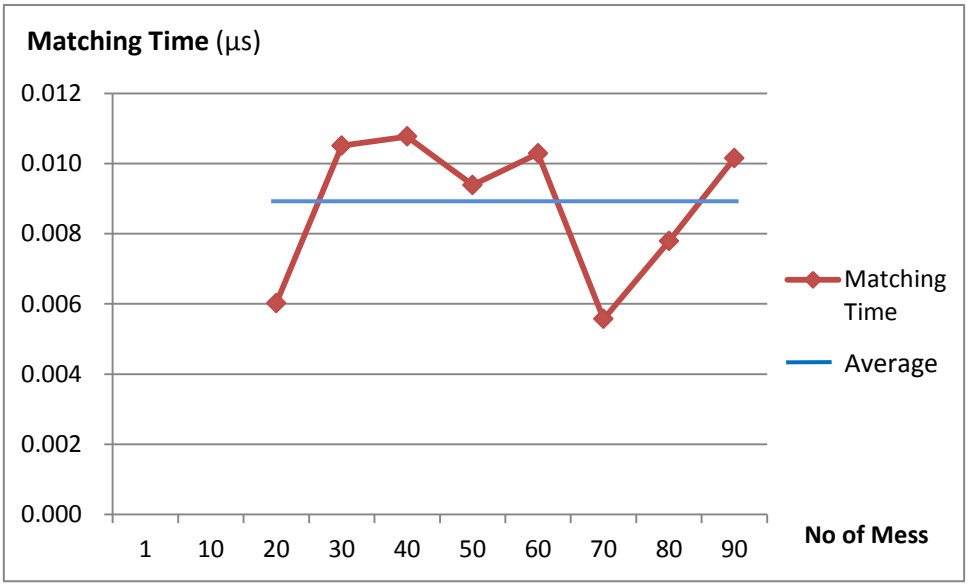

Figure 48: Matching time for 80 bytes messages transferring by Comm Algorithm on INDY0

The remaining parts of the Comm Algorithm implanted on INDY-Linux illustrates in Table 31. The values in the first two group (size = 800 and 8000 bytes) is much smaller than the others. Since the size of data become to 80000 and over, the time for matching increases significantly. It may due to the memory issues. Comm Algorithm is a memory unfriendly algorithm because it creates many communicator on the two nodes which consume the memory resources. Certainly, there must be many other reasons which could be detected in the future.

Size (bytes)	No of Mess	IOT (s)	ROT (s)	Diff (s)	Matching Time (µs)
8000	1	0.000015	0.000015	0.000000	
	10	0.000120	0.000122	0.000002	0.044444
	20	0.000249	0.000250	0.000001	0.006316
	30	0.000377	0.000379	0.000002	0.004598
	40	0.000509	0.000514	0.000005	0.006838
	50	0.000645	0.000652	0.000007	0.005714
	60	0.000803	0.000812	0.000009	0.005085
	70	0.000875	0.000880	0.000005	0.001863
	80	0.001000	0.001005	0.000005	0.001582
	90	0.001186	0.001196	0.000010	0.002372
Average					0.008757
80000	1	0.000112	0.000115	0.000002	
	10	0.001304	0.001309	0.000005	0.103704
	20	0.002518	0.002525	0.000007	0.036842
	30	0.003684	0.003692	0.000008	0.018391
	40	0.004667	0.004683	0.000015	0.019872
	50	0.005712	0.005730	0.000018	0.014694
	60	0.006622	0.006640	0.000019	0.010452
	70	0.006364	0.006385	0.000021	0.008696
	80	0.008026	0.008050	0.000024	0.007489
	90	0.009204	0.009240	0.000036	0.009072
Average					0.025468
800000	1	0.001212	0.001212	0.000000	
	10	0.009154	0.009158	0.000003	0.075556
	20	0.021064	0.021102	0.000038	0.200000
	30	0.030597	0.030677	0.000080	0.184674
	40	0.036756	0.036903	0.000146	0.187821
	50	0.049036	0.048673	*-0.000363*	*-0.296327*
	60	0.049988	0.050038	0.000050	0.028249
	70	0.070209	0.069707	*-0.000502*	*-0.207690*
	80	0.069709	0.069762	0.000053	0.016772
	90	0.076505	0.076648	0.000143	0.035643
Average					0.024966
8000000	1	0.008985	0.008987	0.000002	
	10	0.086059	0.086077	0.000018	0.407407
	20	0.178340	0.178437	0.000097	0.513158
	30	0.293806	0.294027	0.000221	0.508046
	40	0.381480	0.381990	0.000510	0.653846
	50	0.496478	0.497406	0.000928	0.757279
	60	0.569531	0.571017	0.001487	0.839831
	70	0.671736	0.674834	0.003098	1.282954

	80	0.778934	0.785788	0.006854	2.169066
	90	0.836946	0.850249	0.013303	3.321660
Average					1.161472

Table 31: Results of 800 to 8000000 bytes messages transferring by Comm Algorithm on INDY0

After collecting the averaged results in Table 30 and Table 31, Table 32 can be obtained. The final result of matching time per message is about 0.0150592 micorsceond. The value for 8000000 bytes is too large to be unreasonable which should be deleted. Though other values are accurate, they still come with heavy fluctuations. Hence the INDY-Linux is not suitable for Comm Algorithm either.

Size (bytes)	Matching Time per Mess (μs)
80	0.007830
800	0.024966
8000	0.025468
80000	0.008757
800000	0.008275
8000000	1.161472
Average	0.0150592

Table 32: Message matching Time of Comm Algorithm on INDY0

4.3.3 Programs Outcomes of INDY-Windows

The program can be run successfully on INDY-Linux but it failed on INDY-Windows. The code was successfully compiled by the Visual Studio, and the executable application can be submitted to the backend nodes, but it failed as soon as submission completed. The compiled program can be run at frontend, so the code proved to be correct. There might be some other reasons with the software configuration. Because of time constraints, the tests of INDY-Windows will not be performed in this project. Nevertheless, some of the program results running on the frontend nodes can be quoted to show a general idea of the MPI programs with Windows operating system, though these results do not have relative property, because it is a shared architecture on frontend and the performance of frontend nodes are unsteady. Table 33 shows the some direct results of Comm Algorithm on INDY-Windows.

No of Mess	Size (bytes)	IOT (s)	ROT (s)	Diff (s)
1	80	0.000009	0.000005	-0.000005
	800	0.000012	0.000008	-0.000004
	8000	0.000032	0.000031	-0.000001
	80000	0.000789	0.001100	0.000311
	800000	0.000850	0.000859	0.000008
	8000000	0.055325	0.013792	-0.041534
45	80	0.000081	0.000087	0.000007
	800	0.000130	0.000134	0.000003
	8000	0.001196	0.001237	0.000041
	80000	0.005336	0.005110	*-0.000226*
	800000	0.046649	0.046405	*-0.000244*
	8000000	0.587515	0.589183	0.001667
90	80	0.000158	0.000167	0.000009
	800	0.000255	0.000274	0.000019
	8000	0.002358	0.002458	0.000100
	80000	0.010916	0.010753	*-0.000163*
	800000	0.092371	0.092812	0.000440
	8000000	1.059923	1.044196	*-0.015728*

Table 33: Direct results of Comm Algorithm on INDY-Windows

4.4 ECDF

4.4.1 ECDF Configuration

The benchmark program was also compiled and executed on the high-performance compute cluster called Eddie which is operated by ECDF. ECDF, short for Edinburgh Compute and Data Facility, is developed and maintained by the University of Edinburgh and funded by the Science Research Investment Fund (SRIF3) since October 2007. This facility provides flexible and ample computing resources for individual and research community in different subject domains by configuring with multiple languages and program integration. A number of building blocks act as worker nodes run an industry standard Linux based operating system. Up to now, two generations of Eddie have been unveiled, Mark 1 Cluster and Mark 2 Cluster. Since the first generation is unavailable now, only Mark 2 Cluster will be introduced in this Chapter. More information can be found from the ECDF homepage [5].

In Mark 2 Phase 1, Eddie comprises 130 IBM iDataplex DX360 M3 serves and 128 of which contain two quad-core Intel “Westmere” E5620 processors, giving a total of 1024 CPU cores to run the Scientific Linux 4.5 64-bit operating system. Each node has a 24 GB DDR3 RAM, a 250 GB hard disk and two gigabit network cards. Eddie is a distributed memory system machine. Each node contains a 64 KB (32 KB for data and 32 KB for Instruction) Level 1 cache, 256 KB Level 2 cache and 12 MB shared Level 3 cache. All the worker nodes are able to communicate with each other by Ethernet network with a 10 Gigabit network core.

The Mark 2 Phase 2 are improved slightly based on Phases 1, the number of nodes is increased to 156 and the processors in each node are changed into two six-core Intel "Westmere" E5645. Most nodes still connected by the Gigabit Ethernet but 68 nodes A are additionally configured with Infiniband fast and low latency interconnect.

In Mark 2 Phase 3, there are only 6 worker IBM iDataplex DX360 M4 nodes. Each node contains two Intel "Sandy Bridge" E5-2620 six-core processors. However, each node are installed with a 64GB DDR3 RAM and a 500 GB hard disk.

Eddie uses a GPFS (General Parallel File System) storage system which is based on SAN disk platforms. The disk platforms consists of two parts: Tier 1 storage and Tier 2 storage. The first on is built with two IBM DS5300 systems. The disk systems use 300 GB FC disks. The Tier 2 storage is provided by two Sun StorageTek 6540 systems which use 1000GB SATA drives. The total storage achieves 275 TB by adding the two storage systems together. Worker nodes access to in these two storage systems via eight IBM X3650 M3 servers, with 48GB RAM and 10GE networking. As the users’ work directories are shared in the GPFS, all the worker nodes in Eddie can access to all the data in the same way.

4.4.2 Programs Outcomes of ECDF

Before demonstrating the results of ECDF with tables and graphs, a fact should be pre-declared that the original outcomes of the program show a great volatility, for example, Table 34 lists the original results of 10 message of 80 bytes, the minimum number is 2 microsecond while the maximum number is 228 microsecond, even the average number is 148 microsecond. It means that the machine is remarkably unstable. The remaining results have the same situation as this table, so ECDF proved to be not suitable for the micro-benchmark suite. It may due to the configuration or architecture of the machine which requires to be investigated in future. This project will not take the ECDF facility into account any more.

Job No	IOT (s)	ROT (s)
2602077	0.000002	0.000002
2602078	0.000200	0.000198
2602079	0.000203	0.000188
2602080	0.000003	0.000003
2602081	0.000200	0.000196
2602082	0.000228	0.000189
2602084	0.000197	0.000196
Minimum	0.000002	0.000002
Maximum	0.000228	0.000198
Average	0.000148	0.000139

Table 34: Original Results of ECDF with 10 80 bytes messages transferring by Tag Algorithm

4.5 Morar

Morar is chosen as the testing machine to run the benchmark program to assure the robustness, correctness and portability of these programs. It applies to parallel jobs with message-passing interface and its working environment of the HPC system is analogous to the other machines such as HECToR.

Morar is consists of 128 AMD Interlagos cores, they are orginised in two shared-memory boxes (Morar1, Morar2) and each box has 64 CPUs separately. Various kinds of compilers, compiler flags and libraries are available on this machine to help users to compile and executer the MPI programs.

6. Performance Comparison

Based on the analysis and computational results in the last chapter, this section compares the performance of the four machines, Blue Gene/Q, HECToR, INDY-LINUX-LINUX and ECDF computational results.

6.1 Direct Results Comparison

In terms of the Tag Algorithm, some direct results of these machines are listed in Table 35. It includes the 12 values for each machine which is divided by *No of mess*, multiplied by the size (bytes) of each message. For example, 1*800 means the number of messages equal to 1 and size per message is 80 bytes. These results shows that, HECToR is the fastest machine for this particular benchmark algorithm with a smaller *IOT* of 80 bytes message ay 10 microsecond and the smallest *IOT* for 720000000 bytes message is 0.226005.

However, though the implementation process, it has been found that the stability of HECToR is not as good as Blue Gene/Q. Blue Gene/Q has a such a robust stability that every original value within the group are almost same. The performance of Blue Gene/Q is quite optimal as well, so it might be the most powerful supercomputers among the four HPC facilities. INDY-Linux provides an acceptable overall performance. Because it is the best choice for a small message with the shortest *IOT* 1 microsecond when transfer 80 bytes message. But when the size of data package increases, *IOT* grows significantly (i.e 0.8 microsecond for 720000000 bytes message), so it not suitable for large messages. The ECDF obtained unacceptable results, so it is not applied to this project.

No of mess * size (bytes)	BGQ's IOT (s)	HECToR's IOT (s)	INDY-LINUX's IOT (s)
1*80	0.000010	0.000004	0.000001
90*80	0.000359	0.000100	0.000066
1*800	0.000017	0.000004	0.000002
90*800	0.000561	0.000123	0.000099
1*8000	0.000025	0.000015	0.000016
90*8000	0.000757	0.000443	0.001061
1*80000	0.000066	0.000044	0.000110
90*80000	0.004189	0.003652	0.008816
1*800000	0.000471	0.000326	0.001207
90*800000	0.043987	0.033944	0.079159
1*8000000	0.004521	0.002767	0.010273
90*8000000	0.419215	0.226005	0.805049

Table 35: Direct IOT results of Tag Algorithm on three machines

6.2 Final Result Comparison

The summary Table 36 presents the final result comparison of the entire project. For Tag Algorithm, it illustrates that the time to force the matching order of the message on HECToR is the shortest (only 0.016718 μs), which is one of the reasons why HECToR occupied a higher speed

and lower latency. The matching time of BGQ (0.036516 μs) is acceptable while the INDY-LINUX's (0.061953 μs) value is not as satisfactory as others. For Comm Algorithm, the Matching time of HECToR is much smaller than the others. And INDY-Linux's result cannot be guaranteed because the machine is unstable.

	BGQ	HECToR	INDY-LINUX
Matching time for Tag Algorithm (μs)	0.036516	0.016718	0.061953
Matching time for Comm Algorithm(μs)	0.023577	0.005590	0.0150592

Table 36: Matching time Comparison among the three machines
(using identical input parameters for each job)

7. Conclusion and Further Work

To conclude, this project provides a new method the evaluate some performance issues, especially the latency and message matching time, of the four HPC facilities such as Blue Gene/Q, HECToR, INDY and ECDF. The micro-benchmark suite has proved to be a program of high portability, strong robustness and high reliability. Both the Tag Algorithm and Comm Algorithm programs can be implemented successfully on all the supercomputers, adapting to both Linux and Windows operating systems by using various compilers and script files.

The whole benchmark test has been launched hundreds of thousands times to obtain the reasonable original data. These original data was extracted by a specific method which combines the maximum-minimum method and the average method together to eliminate outliers, and then further to this the direct result was extracted. Hence, the overall latency and message matching time of these machines can be resolved by a series of calculations and analysis based on the direct program results.

It terms of Tag Algorithm, time of InOrder pattern increases linearly when the number of messages grows, the time of ReverseOrder pattern rises parabolically with the increasing number of messages. The difference between InOrder and ReverseOrder communication time growth as a parabola line when the number of messages changes. Finally, the matching time per messages turns out as constant though some slight fluctuations exist. The results when a huge data package is applied are not as accurate. The fluctuations may be caused by interference due to other jobs on the machine. On the whole, the mathematic model and assumption of Tag Algorithm are correct, and the final results are reasonable and robust.

The 'one message queue per communicator' model and assumption has been proved to be incorrect by the Comm Algorithm results. However, the 'one message queue for all communicators' model and assumption is consistent with the Comm Algorithm results and is therefore very likely to be correct. The MPI implementer does not create a unique queue for the single messages in communicator, all the messages are still stored in the same queue in some sequence. So forcing the ordering of matching the messages still consume some time, which is similar to Tag Algorithm. The multiple-communicators model cannot optimize the performance of MPI libraries.

Furthermore, by comparing the performance of these four machines, the Blue Gene/Q has proved to be the most stable and powerful machine with optimal computational capabilities. HECToR is the fastest machine but is not stable enough. INDY-Linux can transfer some small messages with a high speed, but it is not able to scale to a large data set. ECDF is not applied to this benchmark program because there is a great volatility in the results.

This project has some scope for further investigation. The failure of execution on the INDY -Windows machine can be researched in more detail to test the MPI libraries on the Windows operating system. The reason for the unsteady performance issues of ECDF could also be investigated. Furthermore, some more HPC facilities might be involved with this micro-benchmark suite.

Appendix

Bechmark_tag Code

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define reps 5

int main(int argc, char* argv[]){
  int rank, size;
  int nummess,length,numiter;
  int rep,i,j,iter;
  int tag,extent;
  double *sbuf,*rbuf;
  double Totmess;

  MPI_Comm comm;
  MPI_Request r[100];
  MPI_Status statuses[100];

  comm    = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

     // Abort if run on less than 2 processors.
  if(size < 2){
     if(rank == 0){
        printf("The code must be run on at least 2 processors.\n");
     }
     MPI_Finalize();
     exit(1);
  }

  if(argc < 3) {
     if(rank == 0){
        printf("Code requires 3 input arguments: \n The number of messages.\n The array
length.\n The number of iterations. \n");
     }
     MPI_Finalize();
```

```
    exit(1);
  }

  if(rank > 1){
    printf( "Rank %d not participating \n",rank);
  }

  if (rank == 0) {
    nummess = atoi(argv[1]);
    length = atoi(argv[2]);
    numiter = atoi(argv[3]);
    printf("Number of messages = %d, Array length = %d , Number of iterations
= %d\n",nummess,length,numiter);
  }

  MPI_Bcast(&nummess,1,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Bcast(&length,1,MPI_INT,0,MPI_COMM_WORLD);
  MPI_Bcast(&numiter,1,MPI_INT,0,MPI_COMM_WORLD);

  int element = reps*numiter;

  double Int1[reps*numiter];
  double Int2[reps*numiter];
  double Ret1[reps*numiter];
  double Ret2[reps*numiter];

  double Intime[reps*numiter];
  double Retime[reps*numiter];
  double Inarray[reps*numiter];
  double Rearray[reps*numiter];

  double Insum=0;
  double Resum=0;
  double Inava, Reava;


    // Allocate array
  sbuf= malloc(nummess*length*sizeof(double));
  rbuf= malloc(nummess*length*sizeof(double));

  if (!sbuf || !rbuf) {
    printf("Could not allocate send/recv buffers of size %d\n", length );
    MPI_Abort( MPI_COMM_WORLD, 1 );
  }
```

```
for(i=0;i<nummess*length;i++){
   sbuf[i] = (double)rank + 10.0;
   rbuf[i] = (double)rank + 10.0;
}

/*Elements location
if (rank == 0)
   for(i=0;i<nummess*length;i+=length){
     printf("value of sbuf[%d]: %f with address %lu \n", i, (sbuf[i]), (unsigned long)(&sbuf[i]));
     printf("value of rbuf[%d]: %f with address %lu \n", i, (rbuf[i]), (unsigned long)(&rbuf[i]));
     }*/

for (rep=0; rep<reps; rep++){

   int order=0;

   // Time the parallel execution.
   MPI_Barrier(MPI_COMM_WORLD);

   // Swap back and forth for iter times


   /**************       InOrder Receive       **************/
   if (order == 0){
      /**************       Test       **************/
      for(iter=0;iter<numiter;iter++) {
         /**************       Batch       **************/
Int1[rep*numiter+iter]= MPI_Wtime();

         if(rank == 0){

            for(i=0; i<nummess; i++){
               MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);

            for(i=0; i<nummess; i++){
               MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
         }

         if (rank == 1){
```

```
            for(i=0; i<nummess; i++){
               MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);

            for(i=0; i<nummess; i++){
               MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
         }
         Int2[rep*numiter+iter]= MPI_Wtime();
         /*************       Batch End       *************/
      }
      /*************       Test End       *************/
      order = 1;
   }
/****************************/


/**************       ReverseOrder Reveive       **************/
   if (order == 1){

      for(iter=0;iter<numiter;iter++){

Ret1[rep*numiter+iter] = MPI_Wtime();

         if(rank == 0){
            for(i=0; i<nummess; i++){
               MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);

            for(i=0; i<nummess; i++){
               MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,1,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
         }

         if (rank == 1){
            for(i=nummess-1;i>=0;i--){
               MPI_Irecv(&rbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
```

```
            for(i=nummess-1;i>=0;i--){
               MPI_Isend(&sbuf[i*length],length,MPI_DOUBLE,0,i,comm,&r[i]);
            }
            MPI_Waitall((int)nummess,r,statuses);
         }
         Ret2[rep*numiter+iter] = MPI_Wtime();
      }

      order = 0;
      //          printf("end of RO");
   }
 /***************************/
 }

 if(rank == 0) {
    for (i=0;i<element;i++) {
       //         MPI_Type_size(MPI_DOUBLE,&extent);
       //          Totmess = 2.0*extent*length/1024*numiter/1024*nummess;
//     printf("222222\n");
       Intime[i] = Int2[i]-Int1[i];
       Retime[i] = Ret2[i]-Ret1[i];
       Inarray[i]= Intime[i];
       Rearray[i]= Retime[i];
       //          printf( "\n%d I: %f\n", i, Intime[i] );
       //          printf( "%d R: %f \n",i, Retime[i] );
    }
 }

 /*Elements location
 if (rank == 0)
    for(i=0;i<nummess*length;i+=length){
      printf("value of sbuf[%d]: %f with address %lu \n", i, (sbuf[i]), (unsigned long)(&sbuf[i]));
      printf("value of rbuf[%d]: %f with address %lu \n", i, (rbuf[i]), (unsigned long)(&rbuf[i]));
      }*/


  if(rank == 0) {
       for (i=5;i<element;i++){
          Insum += Intime[i];
       }

       for (j=5;j<element;j++){
          Resum += Retime[j];
```

```
        }
        Inava = Insum/(element-5);
        Reava = Resum/(element-5);

        printf("\nNumber of messages = %d, Array length = %d ,\
Number of iterations = %d\n",nummess,length,numiter);
        printf("Insum %f\n",Insum);
        printf("Resum %f\n",Resum);
        printf("\nAverage of InOrder:          Inava %f\n",Inava);
        printf("Average of ReverseOrder: Reava %f\n\n\n",Reava);
   }

   if(rank == 0) {
      for (i=0;i<element;i++) {

         printf( "\n%d I: %f\n", i, Inarray[i] );
         printf( "%d R: %f \n",i, Rearray[i] );
      }
      }



 free(sbuf);
 free(rbuf);

 MPI_Finalize();

}
```

Benchmark_comm Code

```
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

#define reps 5

int main(int argc, char* argv[]){
  int rank, size;
  int nummess,length,numiter;
  int rep;
  int i,j,iter,n;
  int tag,extent;
  double *sbuf,*rbuf;
  double Totmess;
  MPI_Comm comm[100];
  MPI_Request r[100];
  MPI_Status status[100];
  MPI_COMM_WORLD;
  tag=0;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  // Abort if run on less than 2 processors.
  if(size < 2){
     if(rank == 0){
        printf("The code must be run on at least 2 processors.\n");
     }
     MPI_Finalize();
     exit(1);
  }

  if(argc < 3) {
     if(rank == 0){
        printf("Code requires 3 input arguments: \n The number of messages.\n The array length.\n The number of iterations. \n");
     }
     MPI_Finalize();
     exit(1);
  }
```

```
 if(rank > 1){
    printf( "Rank %d not participating \n",rank);
 }

 if (rank == 0) {
       nummess = atoi(argv[1]);
       length = atoi(argv[2]);
       numiter = atoi(argv[3]);
       printf("Number of messages = %d, Array length = %d , Number of iterations
= %d\n",nummess,length,numiter);
 }

 MPI_Bcast(&nummess,1,MPI_INT,0,MPI_COMM_WORLD);
 MPI_Bcast(&length,1,MPI_INT,0,MPI_COMM_WORLD);
 MPI_Bcast(&numiter,1,MPI_INT,0,MPI_COMM_WORLD);

 int element = reps*numiter;

 double Int1[reps*numiter];
 double Int2[reps*numiter];
 double Ret1[reps*numiter];
 double Ret2[reps*numiter];

 double Intime[reps*numiter];
 double Retime[reps*numiter];
 double Inarray[reps*numiter];
 double Rearray[reps*numiter];

 double Insum=0;
 double Resum=0;
 double Inava, Reava;

 for (i=0;i<nummess+1;i++){
    MPI_Comm_dup(MPI_COMM_WORLD,&comm[i]);
 }

 // Allocate array
 sbuf= malloc(nummess*length*sizeof(double));
 rbuf= malloc(nummess*length*sizeof(double));

 if (!sbuf || !rbuf) {
    printf("Could not allocate send/recv buffers of size %d\n", length );
    MPI_Abort( MPI_COMM_WORLD, 1 );
```

```
  }

  for(i=0;i<nummess*length;i++) {
     sbuf[i] = (double)rank + 10.0;
     rbuf[i] = (double)rank + 10.0;
  }

////////////////////////////////
/*Elements locations
  if (rank == 0)
     for(n=0;n<nummess*length;n+=length){
       printf("value of sbuf[%d]: %f with address %lu \n", n, (sbuf[n]), (unsigned long)(&sbuf[n]));
       printf("value of rbuf[%d]: %f with address %lu \n", n, (rbuf[n]), (unsigned long)(&rbuf[n]));
       }*/
////////////////////////////////////

  for (rep=0; rep<reps; rep++){

     int order=0;
     // Time the parallel execution.
     MPI_Barrier(MPI_COMM_WORLD);

     /**************       InOrder Receive       **************/
     if (order == 0){
        /**************       Test       **************/
        for(iter=0;iter<numiter;iter++) {
           /**************       Batch       **************/
Int1[rep*numiter+iter]= MPI_Wtime();

           if(rank == 0){

              for(n=0; n<nummess; n++) {
                 MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
              }
              MPI_Waitall((int)nummess,r,status);
              for(n=0; n<nummess; n++) {
                 MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
              }
              MPI_Waitall((int)nummess,r,status);
           }

           if (rank == 1){
              for(n=0; n<nummess; n++){
              MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
```

```
            }
            MPI_Waitall((int)nummess,r,status);
            for(n=0; n<nummess; n++){
               MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
            }
            MPI_Waitall((int)nummess,r,status);
          }

        Int2[rep*numiter+iter]= MPI_Wtime();
          /*************       Batch End       *************/
        }     //end of for loop
        /*************       Test End       *************/
        //       printf("end of IO");
    order = 1;
    }
    /*****************************/

    /**************       ReverseOrder Reveive       **************/
    if (order == 1){

        for(iter=0;iter<numiter;iter++){

Ret1[rep*numiter+iter] = MPI_Wtime();

           if(rank == 0){
              for(n=0; n<nummess; n++){
    MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
  }
    MPI_Waitall((int)nummess,r,status);
  for(n=0; n<nummess; n++){
    MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,1,0,comm[n],&r[n]);
  }
  MPI_Waitall((int)nummess,r,status);
           }

           if (rank == 1){
  for(n=nummess-1;n>=0;n--){
    MPI_Irecv(&rbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
  }
  MPI_Waitall((int)nummess,r,status);
  for(n=nummess-1;n>=0;n--){
    MPI_Isend(&sbuf[n*length],length,MPI_DOUBLE,0,0,comm[n],&r[n]);
  }
    MPI_Waitall((int)nummess,r,status);
```

```
            }

            Ret2[rep*numiter+iter] = MPI_Wtime();
        }

        order = 0;
        //          printf("end of RO");
    }
 /***************************/
 }



 if(rank == 0) {
    for (i=0;i<element;i++) {
        //         MPI_Type_size(MPI_DOUBLE,&extent);
        //          Totmess = 2.0*extent*length/1024*numiter/1024*nummess;
//     printf("222222\n");
        Intime[i] = Int2[i]-Int1[i];
        Retime[i] = Ret2[i]-Ret1[i];
        Inarray[i] = Intime[i];
        Rearray[i] = Retime[i];
        //          printf( "\n%d I: %f\n", i, Intime[i] );
        //          printf( "%d R: %f \n",i, Retime[i] );
    }
 }

 /*Elements location
 if (rank == 0)
    for(n=0;n<nummess*length;n+=length){
      printf("value of sbuf[%d]: %f with address %lu \n", n, (sbuf[n]), (unsigned long)(&sbuf[n]));
      printf("value of rbuf[%d]: %f with address %lu \n", n, (rbuf[n]), (unsigned long)(&rbuf[n]));
      }*/


  if(rank == 0) {
        for (i=5;i<element;i++){
            Insum += Intime[i];
        }

        for (j=5;j<element;j++){
            Resum += Retime[j];
        }
        Inava = Insum/(element-5);
```

```
        Reava = Resum/(element-5);

        printf("\nNumber of messages = %d, Array length = %d ,\
Number of iterations = %d\n",nummess,length,numiter);
            printf("Insum %f\n",Insum);
printf("Resum %f\n",Resum);
printf("\nAverage of InOrder:          Inava %f\n",Inava);
printf("Average of ReverseOrder: Reava %f\n",Reava);

   }
   if(rank == 0) {
      for (i=0;i<element;i++) {
        printf( "\n%d I: %f\n", i, Inarray[i] );
        printf( "%d R: %f \n",i, Rearray[i] );
      }
   }


 free(sbuf);
 free(rbuf);

 for (n=0;n<nummess+1;n++) {
    MPI_Comm_free(&comm[n]);
 }

 MPI_Finalize();

}
```

Reference

[1] Top500, “Top500 List”. [Online]. Available: http://www.top500.org/lists/2012/11/. [Accessed August 2013].

[2] P. Kogge, et.al, *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*, DARPA, 2008, [Online]. Available: http://users.ece.gatech.edu/mrichard/ExascaleComputingStudyReports/exascale_final_report_100208.pdf. [Accessed August 2013].

[3] IBM, “ *Blue Gene*”, [Online], Available: https://www.dirac.ac.uk/. [Accessed August 2013]

[4] UK National Supercomputing Service, [Online], Available: http://www.hector.ac.uk/. [Accessed August 2013]

[5] Edinburgh Parallel Computing Center, “*Industry Machine Wiki*”, [Online], Available: https://www.wiki.ed.ac.uk/display/EPCCIM/Industry+Machine+Wiki [Accessed August 2013]

[6] The University of Edinburgh, “Edinburgh Compute and Data Facility”, [Online], Available: http://www.ed.ac.uk/schools-departments/information-services/services/research-support/research-computing/ecdf/ [Accessed April 2013]

[7] D.Holmes, "*McMPI - a Managed-code Message Passing Interface Library for High Performance Communication in C#*", Edinburgh: University of Edinburgh, 2012.

[8] W. Gropp, E.wing Lusk and A. Skjellum, “*Using MPI: Portable Parallel Programming with the Message-Passing Interface*”, MIT Press, 1999.

[9] Intel, "*Intel MPI Benchmarks*", [Online],Available: http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.[Accessed August 2013].

[10] The University of Edinburgh, *“DiRAC Blue Gene/Q*,”[Online], Available: http://www.epcc.ed.ac.uk/facilities/dirac. [Accessed August 2013]