Analysis of Different Programming Primitives used in a Beowulf Cluster

Mohamed Faidz Mohamed Said, Saadiah Yahya
Faculty of Computer & Mathematical Sciences
Universiti Teknologi MARA
40450 Shah Alam, MALAYSIA
faidzms@ieee.org, saadiah@tmsk.uitm.edu.my

Mohd Nasir Taib
Faculty of Electrical Engineering
Universiti Teknologi MARA,
40450 Shah Alam, MALAYSIA
dr.nasir@ieee.org

Abstract—**Beowulf cluster computing has been widely utilized by exploiting the commodity aspect of its hardware and also the open codes of its software. The implementation of message-passing within this kind of computing is performed via the explicit primitives. Basically, these explicit primitives are divided into two programming types; blocking and non-blocking communications. The effects of using these different primitives on this cluster computing have not been explored in details. This research project proposes a measurement method to empirically look into this effect and how it characterizes the operation of a task. The scope of this research is a collection of four computers that are connected to a switch via a network. Each computer is installed with Linux operating system and MPICH library software. Effects of the programming primitives are measured by a program in C language. The project outcome will offer comparison of the effect of different library routines. Application programmers can exploit this information to produce a better application on this Beowulf computing architecture.**

Keywords- cluster computing; primitives; MPICH

I. INTRODUCTION

Beowulf computing is currently one of the parallel computing architectures that has been used extensively either in the teaching, industrial and commercial sectors. This class of computing is formed by a collection of more than one computer that are linked via a network. The success of this computing architecture is in general due to the exploitation of its physical commodity components that are easily available in the market. On top of that, the software employed by this type of computing are open codes that can be freely downloaded from the public domain. There are some programming concepts that are being utilized by programmers in coding application. These concepts are multiprogramming, shared memory, data parallel and message-passing. In message-passing architectures, each computer is regarded as a building block. This building block viewpoint makes it easier to develop and scale compared to the shared memory perspective. Communication is through explicit input/output (I/O) operation and not inserted into the memory system. This message-passing scheme has similarity with the network of workstations, but its usage in Beowulf computing has a stronger integration between the processor and network.

Historically the goal of achieving performance through the exploitation of parallelism is as old as electronic digital computing itself which emerged from the World War II era. Many different approaches have been devised with many commercial or experimental versions being implemented over the years [1]. Parallel computing architectures may be codified in terms of the coupling and the typical latencies involved in performing parallel operations [2, 3]. The eight major architecture classes are systolic computers [4], vector computers [5], single instruction multiple data (SIMD) architecture [6], dataflow models [7], processor-in-memory (PIM) architecture, massively parallel processors (MPPs) [8], distributed computing [9] and lastly commodity clusters [10, 11]. Commodity clusters may be subdivided into four classes and they are Superclusters, Cluster farms, Workstation clusters and Beowulf clusters. Beowulf clusters incorporate mass-market PC technology and employ commercially available networks such as Ethernet for local area networks. Thus, these characteristics are entirely unlike in a traditional parallel computer where it is built of highly specialized hardware and the architecture is custom built. The term Beowulf cluster refers to a set of regular personal computers (PC) commonly interconnected through an Ethernet. It operates as a parallel computer but differs from other parallel computers in the sense that it consists of mass-produced commodity off-the-shelf (COTS) hardware.

Recently, a rapid increase in the use of this Beowulf clusters can be observed and this is due to mainly two reasons. Firstly, the magnitude of the PC market has allowed PC prices to decrease while sustaining dramatic performance increase. Secondly, the Linux community [12-24] has produced a vast asset of free software for these kinds of applications. Beowulf clusters emphasize no custom components, no dedicated processors, a private system area network and a freely available software base. Cluster computing involves the use of a network of computing resources to provide a comparatively economical package with capabilities once reserved for supercomputers. One of the initial work in developing a Beowulf cluster is carried out by Andersson [12] at the Department of Scientific Computing, Uppsala University, Sweden. On the architectural perspective, the Beowulf cluster can be divided into two types of variants.

The first is the rack-mounted system and the second is the bladed system. Firstly, the rack-mounted system is a collection of individual system units placed together and this study uses this type of implementation. An example of this rack-mounted system is shown by Fig. 1 where it demonstrates a typical home-built Beowulf cluster [25].

Fig. 1. A 52-node Beowulf cluster [25]

Secondly, the bladed system is a collection of individual motherboards put together within the close vicinity, like in computer laboratory. An example of this bladed system is demonstrated by Fig. 2 where it exhibits a 52-node Beowulf cluster [25] used by the McGill University pulsar group to search for pulsations from binary pulsars.

Fig. 2. A home-built Beowulf cluster [25]

An example of this type of implementation is done by Feng [26]. He presents a novel Beowulf cluster named Bladed Beowulf which is originally proposed as a cost-effective alternative to the traditional Beowulf clusters. In his later work, Feng [27] also introduces this Bladed Beowulf and its performance metrics.

Generally, there are many reviews on the preliminary works and discussions in many aspects of the cluster variants. These reviews and discussions include [28], [29], Underwood [30], Kuo-Chan [31], Yi-Hsing [32] and Farrell [33]. Uthayopas discusses the issues in building powerful scalable cluster [34] and also proposes system management for the Beowulf cluster [35]. Stafford [36] discusses the legacy and the future of Beowulf cluster with its founder, Donald Becker. Recent years have shown an immense increase in the use of Beowulf clusters [12, 37]. Their role in providing multiplicity of data paths, increased access to storage elements both in memory and disk and scalable performance is reflected in the wide variety of applications of parallel computing, such as Slezak [38], Yu-Kwong [39] and Chi-Ho [40]. The research works cover both the two memory architecture, namely the shared memory and the message passing. Most of the researches on the later memory architecture are based on the MPICH, a software written by Gropp and Lusk from the Argonne National Laboratory, University of Chicago [41]. The comparison between the message-passing and shared address space parallelism is presented by Shan [42]. For the benchmark segment, two microbenchmarks that analyze network latency that more realistically represents the way that MPI is typically used is presented by Underwood [43]. For comparing the communication types, the work is done by Coti [44] who presents scalability comparisons between MPI blocking and non-blocking check-pointing approaches and Grove [45] who presents tools to measure and model the performance of message-passing communication and application programs. He also presents a new benchmark that uses timing mechanism to measure the performance of a single MPI communication routine. From the numerous reviews made, most of them deal with the issues of the computer communication techniques, computational complexity of scheduling and operating system. Most of these works also focus on the communication latency and load among the networked machines. The network latency, the delay caused by communication between processors and memory modules over the network has been identified as a major source of degraded parallel computing performance. However, these researches have not ventured into the role and effect of the programming primitives used in the application software itself. The analysis on the effect of utilizing different communication primitives based on different data sizes should provide useful information concerning the performance characteristics of pertinent parallel programming codes within the clusters of PC.

This research gap requires detailed analysis by using a new method. Therefore, this research project empirically attempts to look into this effect and how these primitives characterize the operation of task, other than the completion time. The scope of this research is a collection of four computers that are connected to a switch via a network. Each computer is installed with Linux operating system and MPICH library software. Effects of the blocking and non-blocking communication primitives are measured by a program in C language which provides information of the time and rate.

This paper is organized as follows. Section II gives the theoretical background regarding the process in Beowulf parallel computing. Section III provides an in-depth look the

processes involved in the development of this computing. Vital aspects such as the experiment, validation, verification used as well as the benchmarks are also given in this section. Section IV provides the methodology used in the experiment. The measurement algorithm used in this research and its parameters are also shown in this section. Section V provides the analysis and discussions of this research. Lastly, Section VI concludes the findings of this research. For future work, it will also include the recommendations.

II. THEORETICAL BACKGROUND

In message-passing programming on Beowulf computing, a programmer employs message-passing library in order to produce a desired application. This user-level library operates on two principal mechanisms.

The first is the method to create separate process for execution on different computer. Based on the Multiple Program Multiple Data (MPMD) model, there are separate programs for each processor. One processor executes master process while the other processes started from within master process, as depicted in Fig. 3.

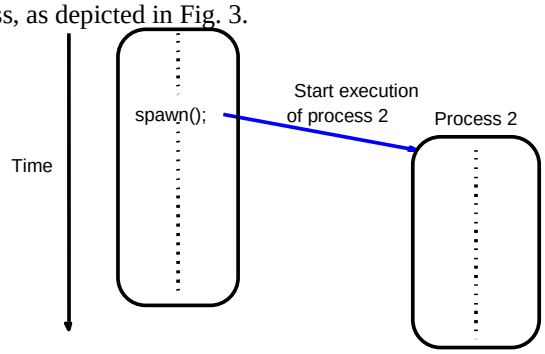

Fig. 3. Multiple Program Multiple Data (MPMD) model

The second is the method to send and receive messages. Basically, for the point-to-point send and receive primitives, passing a message between processes is performed using send() and recv() library calls as shown in Fig. 4.

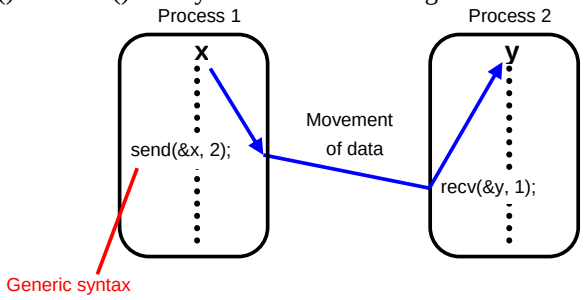

Fig. 4. Basic send and receive primitives

For the synchronous message passing, the routines actually return when message transfer completed. For the send routine, it waits until the complete message can be accepted by the receiving process before sending the message. While for the receive routine, it waits until the message it is expecting arrives. Synchronous routines intrinsically perform two actions: they transfer data and they synchronize processes. This is called blocking communication. The examples of the MPI blocking primitives are *MPI_Send()* and *MPI_Recv()*. The blocking primitives formats are *MPI_Send (buf, count, datatype, dest, tag, comm, request)* and *MPI_Recv (buf, count, datatype, src, tag, comm, request).*

However, for the asynchronous message passing, the routines do not wait for actions to complete before returning and it usually requires local storage for messages. In general, they do not synchronize processes but allow processes to move forward sooner. Thus, in this type of communication, the message-passing routines return before message transfer completed. Message buffer is needed between the source and the destination to hold message. This is called non-blocking communication and demonstrated in Fig. 5.

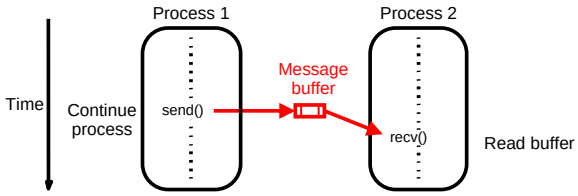

Fig. 5. Message-passing routines return before message transfer completed

The examples of the MPI non-blocking primitives are *MPI_Isend()* and *MPI_Irecv()*. For *MPI_Isend()*, the send will return immediately even before source location is safe to be altered. Meanwhile, for *MPI_Irecv()*, the receive will return even if no message to accept. The '*I*' in '*Isend*' and '*Irecv*' means Immediate. The primitives formats are *MPI_Isend (buf, count, datatype, dest, tag, comm, request)* and *MPI_Irecv(buf, count, datatype, src, tag, comm, request)*. The effects of using these primitives can be explored by empirically measure the completion time and rate based on different message sizes.

III. DEVELOPMENT

In order to accomplish this research, a sequence of development phases are performed (Fig. 6). It is crucial to organize the phases systematically as it is vital in ensuring a well-planned process completion.

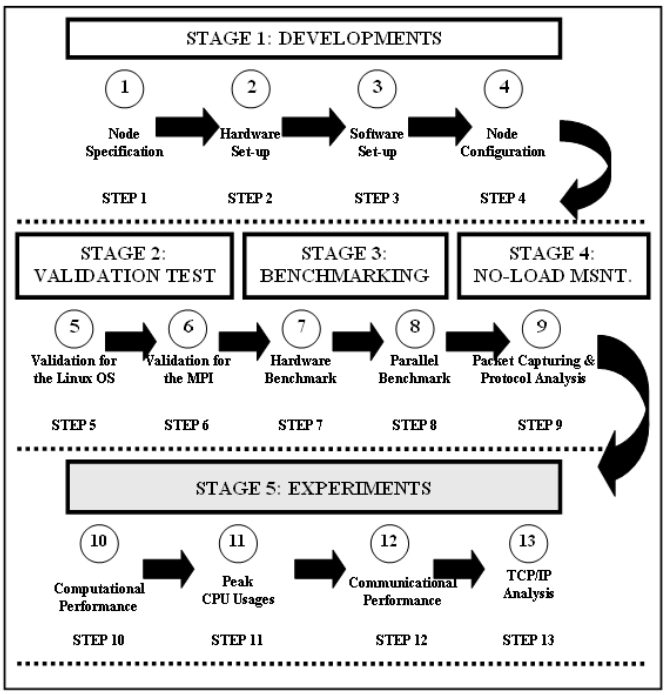

Fig. 6. Methodology for Beowulf cluster developments and experiments

In the early phase of doing this research, it practically starts with the developments stage (Stage 1), where there are four work phases of the developments work. These work phases are the node specification, the hardware set-up, the software set-up and the node configuration of the Beowulf cluster system. The work is arranged in this sequence to ensure that the proper hardware construction is created before setting up the software on top of it. The initial step in this node specification phase is specifying the master and the slave. In order to create a proper naming and numbering convention, the cluster system is conceptually divided into two main components, the master component and the slave component. The convention will have a name node together with a two-digit number. The master node is given a codename of *node00*. The two-digit number 00 is chosen to demonstrate the function of the master node as the front-end PC. Meanwhile the first slave node is given a name and number starting with *node01*. Thus the second node of this cluster system is *node02* and the subsequent third node is *node03*. The last consideration is the network interconnection. Due to the use of a network switch, the link topology being applied will be the star organization. Step 2 demonstrates the second work phase in the developments stage, namely the hardware set-up. This hardware installation phase covers the assembly work and the connections of the nodes through a network interconnect. All the nodes being used are complete standalone systems with monitors, hard drives, keyboards and their related peripherals. Basically the node is comprised of a CPU with a cache, a main memory, a personal computer interconnection (PCI) and a network interface card (NIC). This cluster system is conceptually a combination of four nodes namely individual PCs with a network interconnect device located at the centre of the arrangement. The general structure of this cluster system is presented in Fig. 7.

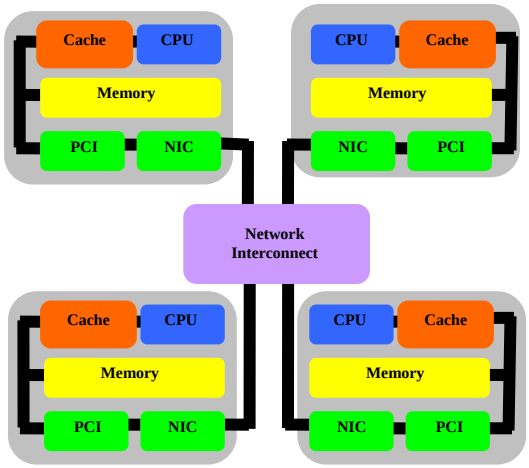

Fig. 7. Hardware set-up

Specifically, the physical units of the cluster system are of heterogeneous characteristics. The entire four nodes are connected through their respective RJ45 ports to a switch using unshielded twisted pair (UTP) cables. A full view of this cluster set-up is illustrated in Fig. 8.

Step 3 illustrates the third work phase in the developments stage; the software set-up. The software installation phase is generally divided into three components. The first software component is the *RedHat 9.0* OS. After the successful installation of the OS, the next component is the *MPICH 1.2.0* library. This software installation phase begins after the nodes are completely assembled physically. Step 4 demonstrates the fourth work phase in the developments section; namely the node configuration (Fig. 9). The node configuration phase for the OS part consists of several tasks.

Fig. 8. A full view of the cluster set-up

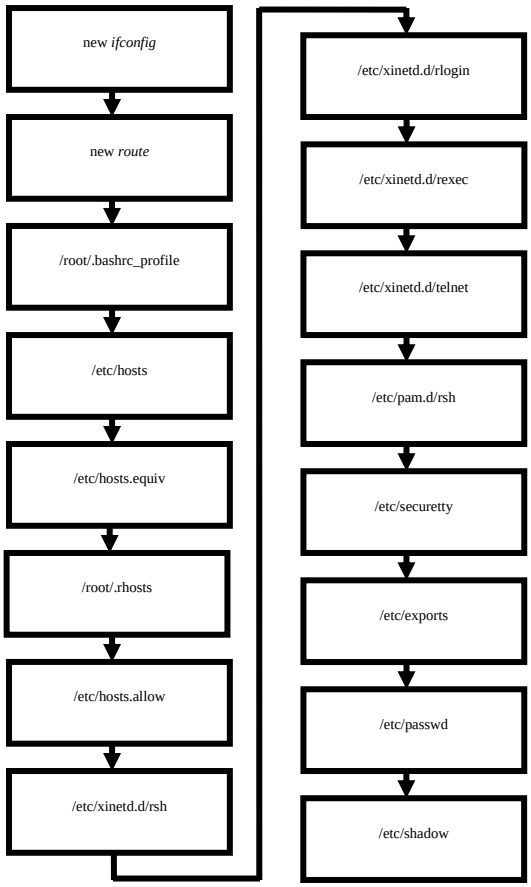

Fig. 9. Flowchart of node configuration

These tasks involve the creation and modification of important system files to ensure that the system is fully functional. The MPI library also has specific essential files that have to be correctly set to run parallel program. Lastly, the proper environment for the experimental data collection should be appropriately established to ensure that the data collection is consistent. This includes the correct command execution, file and directory location. The proper IP address and aliasing of all the nodes are primarily established in the */etc/hosts* file. Each node in the cluster has a similar hosts file with appropriate changes to the first line reflecting the hostname of that node itself. Thus, slave *node01* would have a first line of the text *192.168.0.9 node01* with the third line containing the IP and hostname of *node00*. All other nodes are configured in the same manner with the *127.0.0.1* localhost line is not removed. This file is edited on every cluster node by adding the names and IP addresses of every node in the cluster. This allows these machines to be accessed by name instead of by IP number. In general, the system files that need to be created and modified are illustrated by the flowchart as shown in Fig. 9.

For the validation test (Stage 2), there are two types of tests applied to ensure that the whole system is properly working and functioning. The first is the validation for the Linux OS installation and the second is the validation for the MPICH installation. In order to validate the successful setup of the Linux OS into each of the nodes, several attribute elements of the node system can be verified. These attribute elements will prove that the OS is correctly set-up and functioning. The elements consist of the configuration verification, the routing table verification, the data transfer verification, the re-verification of the overall performance and slave validation. These system elements are verified and confirmed on the master node as well as on all of the remote slave nodes. For the validation of the MPI installation (Step 6), *Hello World* program is applied. In this program, the MPI specifies the library calls to be used in a C program. The MPI program contains one call to *MPI_Init* and one call to *MPI_Finalize*. Therefore all other MPI routines must be called after *MPI_Init* and before *MPI_Finalize*. Furthermore the C program must also include the file *mpi.h* statement at the beginning of the program.

A. Benchmarking

In this benchmarking phase (Stage 3), it describes the chosen benchmarking being used in the Beowulf cluster system. For the reliability testing, the performance of the developed cluster is tested using the authoritative benchmarks. There are two kinds of benchmark programs; the hardware benchmarks and the parallel benchmarks. For comparison purposes, the Grendel cluster system (G-cluster) is chosen [12]. The hardware benchmarks used is the LMbench 2.0 benchmark while the parallel benchmark applied is the NAS Parallel Benchmark 2.3 (NPB 2.3). LMbench is a set of small benchmarks used to measure performance of computer components which are vital for efficient system performance. The aim of these benchmark tests is to provide the real application figures that can be achieved by normal applications. The main performance bottlenecks of current systems are latency, bandwidth or a combination of these two. LMbench tests focus on the system's ability to transfer data between processor, cache, memory, disk and network. However, these tests do not measure the graphics throughput, computational speed or any multiprocessor features of a computer node. Since LMbench is highly portable, it should run as is with *gcc* as default compiler. This LMbench benchmark tests six different aspects of the system. These are the processor and processes, the context switching, the communication latency, the file and virtual memory system latencies, the communication bandwidths and the memory latencies.

Firstly, the results of LMbench 2.0 benchmark for the processor and processes are displayed below (Table 1). The times shown are in microseconds (µs). In the nineth test (Table 1), for creating a process through *fork+exec*, the exec proc measures the time it takes to create a new process and have that process perform a new task. The time taken to exec proc for this cluster is 344.0 µs compared to 706.2 µs. Lastly, in the tenth test, for creating a process through *fork+/bin/sh –c*, the shell proc measures the time it takes to create a new process and have the new process running a

program by asking the shell to find that program and run it. The time taken to shell proc for this cluster is 2247 µs compared to 3605.3 µs. Generally, the comparison results from the LMbench tests for the processor and processes show that this Beowulf cluster produces significantly better performance of a small-scale cluster.

Table 1. LMbench 2.0 benchmark for the processor and processes – smaller is better

		This cluster	G-cluster
1	null call	0.45	0.27
2	null I/O	0.51	0.38
3	stat	1.78	3.72
4	open/close	2.38	4.63
5	select	5.937	26.3
6	signal install	0.79	0.77
7	signal handle/catch	2.59	0.95
8	fork proc	99.0	110.1
9	exec proc	344.0	706.2
10	shell proc	2247	3605.3

Secondly, the results of LMbench 2.0 benchmark for the communication latencies are exhibited below (Table 2). The times shown are in microseconds (µs). In the fifth test (Table 2), for the interprocess communication latency via TCP/IP, TCP measures the time it takes to send a token back and forth between a client/server. No work is done in the processes. The time taken for the TCP of this cluster is 14.1 µs compared to 16.4 µs.

Table 2. LMbench 2.0 benchmark for the local communication latencies (µs) – smaller is better

		This cluster	G-cluster
1	pipe	4.808	4.021
2	AF UNIX	9.46	8.34
3	UDP	11.9	11.5
4	RPC/UDP	21.4	26.4
5	TCP	14.1	16.4
6	RPC/TCP	25.9	39.1

Thirdly, the results of LMbench 2.0 benchmark for the local communication bandwidths are displayed below (Table 3). The measurements shown are in Mbytes per second (MB/s). In the third test, for reading and summing of a file, file reread measures how fast data is read when reading a file in 64KB blocks. Each block is summed up as a series of 4 byte integers in an unrolled loop. The benchmark is intended to be used on a file that is in memory. The bandwidth for the file reread of this cluster is 1149.3 MB/s compared to 332.9 MB/s. Generally, the comparison results from the LMbench tests for the local communication bandwidths show that this Beowulf cluster produces a better performance in the whole local communication bandwidths category tests conducted.

Table 3. LMbench 2.0 benchmark those are for the local communication bandwidths (MB/s) – bigger is better

		This cluster	G-cluster
1	pipe	1181	790.7
2	AF UNIX	2033	516.3
3	file reread	1149.3	332.9
4	Mmap reread	1164.3	462.0
5	Bcopy (libc)	369.8	300.6
6	Bcopy (hand)	387.9	264.1
7	mem read	1522	481.7
8	mem write	522.4	361.6

Finally, the results of LMbench 2.0 benchmark for the memory latencies are presented below (Table 4). The measurements shown are in nanosecond (ns).

Table 4. LMbench 2.0 benchmark for the memory latencies

		This cluster	G-cluster
1	L1 cache	0.836	2.279
2	L2 cache	7.7070	19.0
3	Main memory	118.5	151.0

For the memory read latencies, L1 cache, L2 cache and Main memory measure the time it takes to read memory with varying memory sizes and strides respectively. The entire memory hierarchy is measured onboard and external caches, main memory and TLB miss latency. It does not measure the instruction cache. Generally, the comparison results from the LMbench tests for the memory latencies show that this Beowulf cluster produces a better performance for a cluster since smaller is better.

IV. METHODOLOGY

In order to make the required measurement program, an algorithm is firstly designed. The program is coded using C language because of its suitable attribute and more flexible than the others. The program is tested to ensure it is correct and modifications will be done from time to time if needed. It starts with the program initialization and specifying the program parameters. To start a program, *MPI_Init()* is used before calling any MPI function. All processes are enrolled in a universe called *MPI_Comm_World*. Each process is given a unique rank number from *0* up to *p-1* for *p* processes. To terminate a program, *MPI_Finalize()* is used. To measure the execution time between two points in the code, *MPI_Wtime()* routines are used together with the appropriate variables. Thus, initially, the program may call the *MPI_Init* and later call *MPI_Comm_rank()* and *MPI_Comm_size()*. The body of the measurement program runs the test and the times are recorded. The rates and times are inversely proportional. To record the total amount of time that the test takes, the *MPI_Wtime()* function is used since the MPI timer is an elapsed timer: *start_time = MPI_Wtime(); run_time = MPI_Wtime() - start_time*. The time function is a function of several other routines of the first data length (*first*), the last data length (*last*), process 1 (*proc1*), process 2 (*proc2*), the communication test (*commtest*) and the context of the message-passing operation (*msgctx*): *time_function (first,last,incr,proc1,proc2, commtest, msgctx)*; In the main program, the communication test (*commtest*) is identified as double data type, and it is a function of the protocol used. This *commtest* is a function of *&argc, argv* and *protocol_name* where the protocol name is of char data type

and the options are blocking, nonblocking and overlap. The context of the message-passing operation (*msgctx*) is a function of *proc1* and *proc2*. Finally, the program ends with *MPI_Finalize();* and *return 0.* The *mpptest* of the Linux performance test is chosen because it has the merit of having the same purpose of the required measurement. All the experiments conducted for the blocking and non-blocking operations are performed based on various message sizes either at lower level or near saturation level to compare the effect of different packet sizes.

The measurement program will provide the completion time and the rate of each different message size. Thus, the performance effects of using blocking and non-blocking communication can be compared empirically after conducting a series of experiments on each library routines.

V. ANALYSIS AND DISCUSSIONS

In this section, it shows the results for the experiments on the communicational performance (Step 12). The comparisons are made from the perspective of the rate (bandwidth) as the message sizes are changed. Fig. 10 provides the results of the non-blocking operations on the Beowulf cluster based on the different message sizes and number of processors.

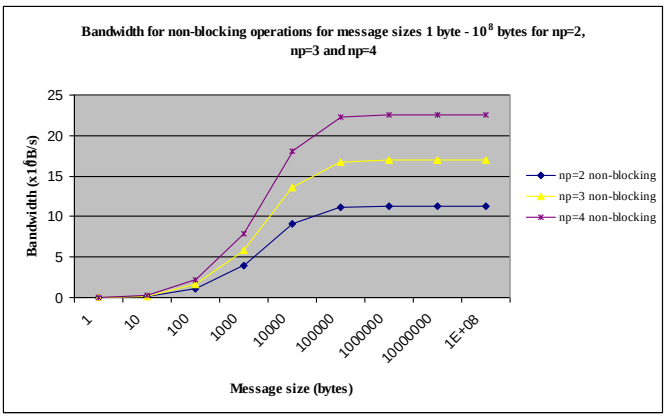

Fig. 10. Rate for the non-blocking operations for np=2, np=3 and np=4

The lowest line is the measurement for np=2, the middle line is the measurement for np=3 and the highest line is the measurement for np=4. By adding more processors, the rate of non-blocking communication for each np generally increases up to a certain saturation level. The saturation levels are different for each np. Therefore, all non-blocking operations with different np show almost the same characteristics of gradually rising and becoming stable during saturation level.

Similarly, Fig. 11 provides the results of the blocking operations on the Beowulf cluster based on the different message sizes and number of processors.

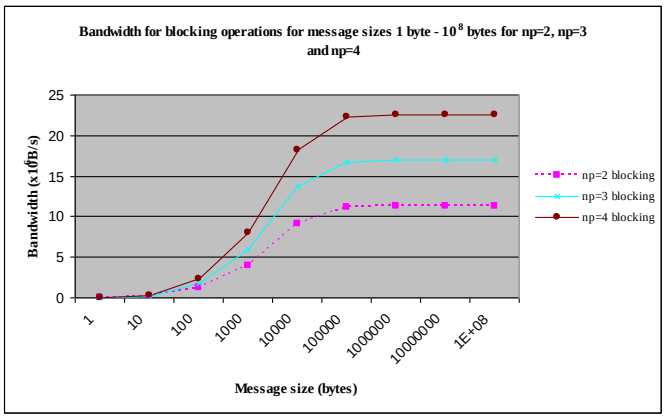

Fig. 11. Bandwidth for the blocking operations for np=2, np=3 and np=4

The lowest line is the measurement for np=2, the middle line is the measurement for np=3 and the highest line is the measurement for np=4. Similarly, by adding more processors, the rate of blocking communication for each np generally increases up to a certain saturation level. The saturation levels are different for each np. Therefore, all blocking operations with different np show nearly the same characteristics of gradually rising and becoming stable during saturation level.

Subsequently, Fig. 12 summarizes both results on the non-blocking and blocking operations for np=2, np=3 and np=4 in one graph.

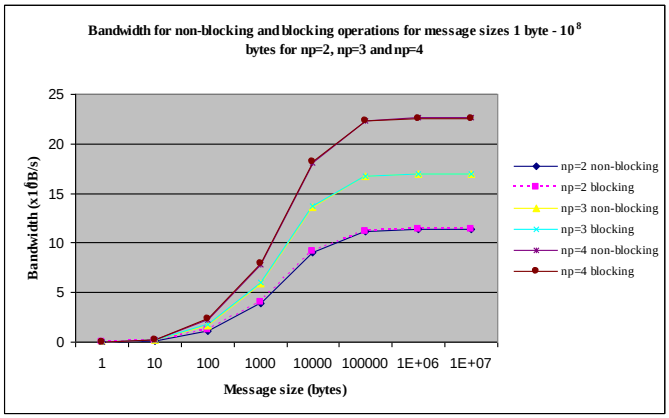

Fig. 12. Rate comparison between non-blocking and blocking operations for np=2, np=3 and np=4

Generally, both operations show almost the same rate of message passing between different sizes and number of processors. The rate differences between these operations are very minimal as per each np. This should indicate that the use of the non-blocking or blocking routines in this cluster computing has very little effects on the overall performance in terms of the rate of message transmission. Either routine could be applied without having to reconsider the overall impact on the running application.

VI. CONCLUSSION

This research provides significant findings on the developed Beowulf cluster system with its message-passing

implementation. This Beowulf cluster has been compared to other cluster in many benchmarks as to exhibit that this setup has a comparable high-performance computing capability. The cluster system shows the use the distributed memory system utilizing the message-passing interface programming model where the communication is via explicit messages primitives. These message primitives consist of the blocking and non-blocking communications. The blocking communication involves the send/receive request and waits until the reply is returned. However, when the programming model of non-blocking communication is used, the messages can return soon without waiting for the finish of communication operation because the communication operation can be managed by communication system in bottom layer of system. From this study, the research shows that the message rate will increase as the number of nodes increases. The average round-trip time also shows very small difference between the two MPI routines.

This research introduces an alternative method to observe this phenomenon by looking into the information on the time and rate based on the two different MPI routines. The benefit of understanding the performance of the message-passing communication primitives will provide programmers to write efficient parallel software and therefore will eventually contribute to the improved performance of parallel applications. The development studies obtained from this research could also be applied as key guidelines in developing similar Beowulf cluster computing system.

A. Recommendation

The scope of this research is based on a small-scale four-node cluster. Therefore, the work of the future research would be based on higher number of nodes as to investigate how well the programming primitives scale when the number of nodes is increased. The effects will be especially interesting when blocking primitives are used. Apart from the point-to-point communication, future work could also examine the collective communication as these are the group message-passing routines where these routines send and receive messages to a group of processes. Collective routines includes cases when all the other nodes send messages exactly to one node, as well as when every node send messages to all of the other nodes. Hence, the comparison between the point-to-point and collective communications could provide the efficiency comparison on both categories of the MPI routines. Lastly, the communications performed by the MPI library routines require buffer space to complete the operation. Future research can look into the performance effect when the size of this buffer space is changed.

REFERENCES

[1] G. K. Thiruvathukal, "Guest Editors' Introduction: Cluster Computing," *Computing in Science & Engineering*, vol. 7 no. 2, pp. 11-13, 2005.

[2] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, vol. 54 no. 12, pp. 1901-1909, 1969.

[3] M. Flynn, "Multiprocessors," in *Chapter 8 Lectures*, 1998.

[4] K. T. Johnson, A. R. Hurson, and B. Shirazi, "General-Purpose Systolic Arrays," *Computer*, vol. Nov. 1993, pp. 20-31, 1993.

[5] G. R. Luecke, B. Raffin, and J. J. Coyle, "Comparing the Communication Performance and Scalability of a Linux and a NT Cluster of PCs, a Cray Origin 2000, an IBM SP and a Cray T3E-600," 1999, pp. 26-35.

[6] M. Sung, "SIMD Parallel Processing," *Architectures Anonymous*, vol. 6, pp. 11, 2000.

[7] R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing," *J. Applied Mathematics*, vol. 14, pp. 1390-1411, 1966.

[8] H. Kai, W. Choming, W. Cho-Li, and X. Zhiwei, "Resource Scaling Effects on MPP Performance: The STAP Benchmark Implications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10 no. 5, pp. 509-527, 1999.

[9] K. Watanabe, T. Otsuka, J. I. Tsuchiya, H. Amano, H. Harada, J. Yamamoto, H. Nishi, and T. Kudoh, "Performance Evaluation of RHiNET 2/NI: A Network Interface for Distributed Parallel Computing Systems," 2003, pp. 318-325.

[10] M. Faidz, M. N. Taib, and S. Yahya, "Overlapping Effect of Message-Passing and Computation in a Beowulf Cluster Computing," *unpublished*, 2012.

[11] M. Faidz, M. N. Taib, and S. Yahya, "Analysis of the MPI Communication Performance in a Distributed Memory System Architecture," *unpublished*, 2012.

[12] K.-J. Andersson, D. Aronsson, and P. Karlsson, "An Evaluation of the System Performance of a Beowulf Cluster. Internal Report No. 2001:4," *http://www.nsc.liu.se/grendel*, 2001.

[13] "Linux links, http://www.linuxlinks.com/," accessed on 1 Feb 2007.

[14] M. Perry, "Building Linux Beowulf Clusters," *http://fscked.org/writings/clusters/cluster.html*, 2000.

[15] "Linux Start, http://www.linuxlinks.com/," accessed on 1 Feb 2007.

[16] "The Beowulf Underground," *http://beowulf-underground.org/*, accessed on 1 Feb 2007.

[17] "The Linux HOWTO Index, http://sunsite.unc.edu/mdw/HOWTO," accessed on 1 Feb 2007.

[18] "RedHat Linux, http://www.redhat.com/," accessed on 1 Feb 2007.

[19] "Mandrake Linux, http://www.redhat.com/," accessed on 1 Feb 2007.

[20] "Linux Software for Scientists, http://www.llp.fu-berlin.de/baum/linuxlist-a.html," accessed on 30 Jan 2007.

[21] "Linux Gazette," *http://www.linuxgazette.com/*, accessed on 1 Feb 2007.

[22] "Linux Journal's Linux Resources, http://www.ssc.com/linux," accessed on 1 Feb 2007.

[23] S. Blank, "Using MPICH to Build a Small Private Beowulf Cluster," *http://www.linuxjournal.com/article/5690*, 2002.

[24] "Scientific Applications on Linux, http://sal.kachinatech.com/," accessed on 1 Feb 2007.

[25] "Beowulf (computing), http://en.wikipedia.org/wiki/Beowulf_(computing)#Original_Beowulf_HOWTO_Definition," accessed on 4 Dec 2007.

[26] W. Feng, M. Warren, and E. Weigle, "The Bladed Beowulf: A Cost-Effective Alternative to Traditional Beowulfs," in *Proc. The IEEE International Conference on Cluster Computing (CLUSTER'02)*, 2002.

[27] W. Feng, M. Warren, and E. Weigle, "Honey, I Shrunk the Beowulf!," in *Proc. The International Conference on Parallel Processing (ICPP'02)*, 2002.

[28] L. Wen-lang, X. An-dong, and R. Wen, "The Construction and Test for a Small Beowulf Parallel Computing System," in *Proc. Third International Symposium on Intelligent Information Technology and Security Informatics (IITSI)*, Jinggangshan, China, 2010, pp. 767-770.

[29] M. Warren, E. H. Weigle, and W.-C. Feng, "High-Density Computing: A 240-Processor Beowulf in One Cubic Meter," in *Proc. The IEEE/ACM SC2002 Conference (SC'02)*, 2002.

[30] K. D. Underwood, R. R. Sass, and I. Walter B. Ligon, "Cost Effectiveness of an Adaptable Computing Cluster," in *Proc. The ACM/IEEE SC2001 Conference (SC'01)*, 2001.

[31] H. Kuo-Chan, C. Hsi-Ya, S. Cherng-Yeu, C. Chaur-Yi, and T. Shou-Cheng, "Benchmarking and Performance Evaluation of NCHC PC Cluster," National Center for High-Performance Computing, Hsinchu, Taiwan, 2000, pp. 923-928.

[32] C. Yi-Hsing and J. W. Chen, "Designing an Enhanced PC Cluster System for Scalable Network Services," in *Proc. 19th International Conference on Advanced Information Networking and Applications (AINA 2005)*, 2005, pp. 163-166.

[33] P. Farrell, "Factors Involved in the Performance of Computations on Beowulf Clusters," *Electronic Transactions on Numerical Analysis*, vol. 15, pp. 211-224, 2003.

[34] P. Uthayopas, T. Angskun, and J. Maneesilp, "On the Building of the Next Generation Integrated Environment for Beowulf Clusters," in *Proc. The International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN'02)*, 2002, pp. 1-6.

[35] P. Uthayopas, S. Paisitbenchapol, T. Angskun, and J. Maneesilp, "System Management Framework and Tools for Beowulf Cluster," Computer and Network System Research Laboratory, Kasetsart University, Bangkok, 2000.

[36] J. Stafford, "Beowulf Founder: Linux is Ready for High-Performance Computing," SearchOpenSource.com, 2004.

[37] R. Kunz and J. Watson, "Clusters - Modern High Performance Computing Platforms," Penn State Applied Research Laboratory, 2004.

[38] D. Fernandez Slezak, P. G. Turjanski, D. Montaldo, and E. E. Mocskos, "Hands-On Experience in HPC with Secondary School Students," *IEEE Transactions on Education*, vol. 53, pp. 128-135, 2010.

[39] K. Yu-Kwong, "Parallel program execution on a heterogeneous PC cluster using task duplication," 2000, pp. 364-374.

[40] L. Chi-Ho, P. Kui-Hong, and K. Jong-Hwan, "Hybrid parallel, evolutionary algorithms for constrained optimization utilizing PC clustering," 2001, pp. 1436-1441.

[41] W. Gropp and E. Lusk, *User's Guide for MPICH, a Portable Implementation of MPI Version 1.2.0*: Argonne National Laboratory, University of Chicago, 1996.

[42] H. Shan, J. P. Singh, L. Oliker, and R. Biswas, "Message Passing and Shared Address Space Parallelism on an SMP Cluster," *Parallel Computing*, vol. 29 no. 2, pp. 167-186, 2002.

[43] K. D. Underwood and R. Brightwell, "The Impact of MPI Queue Usage on Message Latency," in *Proc. International Conference on Parallel Processing (ICPP 2004)*, 2004, pp. 152-160.

[44] C. Coti, T. Herault, P. Lemarinier, L. Pilard, A. Rezmerita, E. Rodriguez, and F. Cappello, "MPI Tools and Performance Studies - Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI," in *Proc. 2006 ACM/IEEE Conference on Supercomputing* Tampa, Florida 2006, pp. 127.

[45] D. Grove and P. Coddington, "Precise MPI Performance Measurement using MPIBench," *http://parallel.hpc.unsw.edu.au/HPCAsia/papers/72.pdf*, 2001.