Overlapping Communication and Computation in MPI by Multithreading [*]

Mao Jiayin Song Bo Wu Yongwei Yang Guangwen
Department of Computer Science and Technology;
Tsinghua National Laboratory for Information Science and Technology
Tsinghua University
Beijing 100084, China

Abstract

Since the emergence of MPI (Message Passing Interface), how to improve its performance has always been a goal for MPI library implementation. This paper proposes an efficient architecture to achieve this goal from a very low level: using multi-threaded model to implement MPI point-to-point operations in order to overlap communication and computation. Compared to single-threaded active polling communication used by most of the implementations, multi-threaded provides more parallelism. A dedicated thread, which is responsible for communication, can spare substantive CPU time for computation. Multi-threaded point-to-point operations have been implemented and tested, and the result is compared with single-threaded implementations in this paper. Both theoretical analysis and experiment result show that the multi-threaded model greatly enhances the efficiency of MPI point-to-point communication. This model is adopted in an ongoing MPI implementation project called FiTMPI.

Keywords: *MPI, multi-threaded, overlapping communication and computation*

1. Introduction

MPI (Message Passing Interface) [7] has becoming a dominant standard in the world of parallel computing. It is intended to establish a practical, portable, efficient, and flexible standard for message passing. MPI standard consists of more than a hundred functions, which can be categorized into several modules, including point-to-point communications, collective communications and so on. The efficiency of Point-to-point communications directly influence the overall performance: on the one hand, they are directly called point-to-point functions by user's code, on the other hand, many implementations use them as the basis of collective communications.

The ability to efficiently overlap communication and computation has long been considered as a significant performance benefit for MPI applications [4]. In message passing paradigm, communication is usually carried out by network interface adapter, and if they can perform communication asynchronously, the cost of message passing on the host processor can be greatly reduced, thus low CPU overhead can be achieved, and the saved CPU resources can be utilized in computation.

This paper proposes a multi-threaded model to overlap communication and computation. This way is different from other methods to achieve overlapping, such as OS-bypass mechanism and non-blocking operation provided by MPI itself.

This new model is used in FiTMPI, which is an ongoing project aims at implementing a high performance, easy-to-use, and grid enabled MPI library.

The rest of this paper is organized as follows. In the following section, related work is briefly introduced. The detailed design of multi-threaded model and implementation issues are described in Section 3 and the theoretical evaluation of this model is showed in Section 4. Section 5 presents the experiment results and Section 6 outlines possible avenues of future investigation.

In this paper, point-to-point communication mainly refers to the communication on TCP/IP protocol stack on Ethernet.

2. Related Works

Multithreading is a mature technique in the communication world. After being studied in lab for decades, multithreading programming (MT) has already been widely supported across platforms [10]. There are three different definitions for thread libraries competing for attention today: Win32, OS/2, and POSIX [2].

[*]This Work is supported by ChinaGrid project of Ministry of Education of China, Natural Science Foundation of China (60373004, 60373005, 90412006, 90412011, 60573110, 90612016), and National Key Basic Research Project of China (2004CB318000, 2003CB316907)

Although multithreading is used to achieve overlapping communication and computation in this paper, the overlapping problem has been already studied in several layers, from physical layer to application layer. Myrinet[3] and Infiniband[1] are the representations of the OS-bypass communication technologies. The two OS-bypass communication technologies are solutions to overlapping requirement from bottom to up, hence for the traditional infrastructure such as TCP/IP running on Ethernet, the overlapping capacity cannot be easily utilized.

Boris V. Protopopov proposed a multi-threaded MPI architecture and discussed several performance and program issues [11]. In that paper, the architecture is developed in the framework of the MPICH [8] software architecture. Unfortunately, no successive information is supplied about this architecture, without knowing experimental results, no conclusion can be derived.

3. Design and Implementation

Although the multi-threaded model aims at improving the overall performance of MPI, it only has relation with point-to-point communication functions. Fortunately, the hundreds of MPI functions can be easily categorized into several modules, and point-to-point is an independent one.

3.1. Design

The point-to-point module is shown in Fig.1. It has two threads running by turns. The user thread is the main thread of the process, and a daemon thread called worker thread is started in the MPI_INIT call, running detached, and terminated automatically when the process exits.

Request queue and event queue are two lists containing Message State Objects (MSO). Every MSO corresponds to an MPI send/recv operation, either has been called by user's code or came from underlying network but not yet called. When user's code calls MPI send/recv actively, user thread first generates an MSO and checks if this request is already done in the event queue. If not, the request is appended(produced) to the request queue and consumed by worker thread. If a message comes from underlying network, the worker thread receives it without hesitance and append(produce) it to the event queue, later when the receive request is posted and user thread finds it in the event queue, this MSO copies message data to user's buffer locally.

There are another collection data structures: bridge table and pollfd array. The former is used to translate process rank and communicator to the network entry of this process, namely its host address and listening port. This table is filled up in initialization phase but the bridge's connection status is updated during send/recv operations. Pollfd

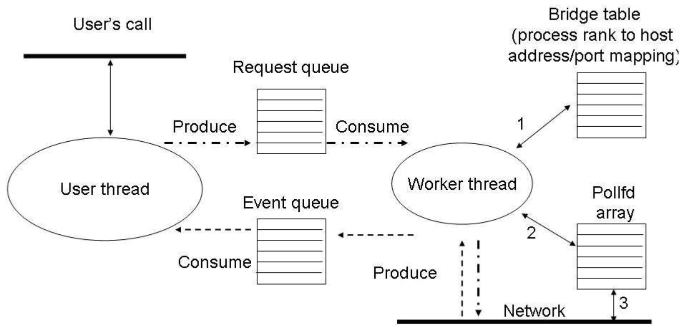

Figure 1. Point-to-point module design.There are two dashed lines representing the operation sequence. 1: translate process rank and communicator to the network entry of this process, namely its host address and listening port. 2: sets the events on which fd (file descriptor, an identity for a socket connection) worker thread is interested and gets the occurred events on fd. 3: Active polling

array stores the events on which worker thread is interested and the events occurred.

There are several other modules in an intact MPI library, but the performance is heavily dependent on collective and point-to-point modules, and all the collective operations are built on top of point-to-point ones, so the improvement in point-to-point module leads to an overall enhancement.

3.2. MPI_SEND Implementation

MPI_SEND is also called the standard mode send, compared with MPI_BSEND, SSEND and RSEND. In this mode, it is up to MPI to decide whether outgoing messages should be buffered. If MPI buffers message, the send may complete when buffering is done, before a matching receive is posted [7].

Most of the other MPI libraries choose not to buffer messages, while this one always tries to. In initialization the library code allocates a block of memory as library buffer. MPI_SEND first generates an MSO and appends it to the request queue, and then looks up enough buffer for the message from library buffer, if there is, the message is copied into buffer and waits to be sent by worker thread, and the MPI_SEND called in user thread returns; else the user thread goes to sleep until the message is sent across network by worker thread in the standard fashion. The logic is drawn in Fig.2.

On the other hand, the worker thread continuously checks the status of network event by polling and the con-

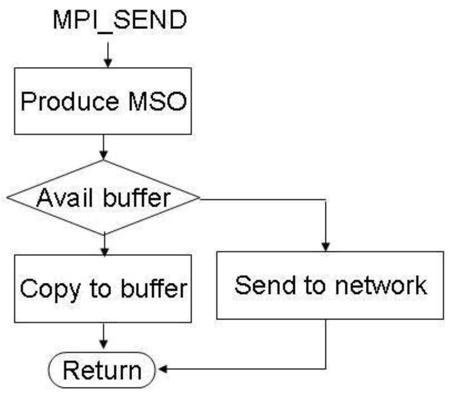

Figure 2. MPI_SEND logic diagram

tent of library buffer. If there is any byte pending in library buffer, worker thread sends it. Any writable socket's appearing also causes the return of polling, and then worker thread sends message through the socket.

3.3. MPI_BSEND Implementation

MPI_BSEND is buffered send, whose complete means the message is buffered in a user submitted buffer. In multi-threaded model, MPI_BSEND can safely return as early as the MPI standard claims. Worker thread takes the responsibility of copying messages from user submitted buffer to receive end and finishing transferring the remaining data.

3.4. MPI_ISEND Implementation

MPI_ISEND is a non-blocking send and returns after it starts the send operation. The functionality in single-threaded MPI and multi-threaded one is the same for MPI_ISEND, but the successive operation is different. In single-threaded library, after MPI_ISEND, no operation is executed before calling send complete call, such as MPI_WAIT. However, in multi-threaded library, successive operation is continuously carried out by worker thread, with the user thread running in parallel.

3.5. MPI_RECV and MPI_IRECV Implementation

When a MPI_RECV function is called, it first checks if there is any matching MSO in event queue. If there is, it simply de-queues this MSO from event queue and en-queues it to request queue and copies the message to user's buffer. Else it generates an MSO and appends it to request queue and goes to sleep until the worker thread finishes receiving this message. This portion is similar as how the single-threaded MPI dose.

The major difference is in the worker thread. Just when the worker thread is started, it begins polling read event on connected socket. If a message arrives and the matching receive MSO is already in request queue (posted), worker thread receives it; if un-posted message arrives, worker thread also receives it and buffers it. That is to say, whether send operation is prior or posterior to receive operation, the communication always proceeds as long as resource is ready. Comparatively, in single-threaded MPI, communication may often be blocked because of asynchronous send/receive pair.

The logic of MPI_IRECV is similar to MPI_RECV except that it returns immediately after appending MSO to the request queue. Communication progresses in the worker thread and completes in the *receive complete call*(e.g. MPI_WAIT).

4. Theoretical Evaluation

There are several parallel program evaluation models, such as LogP [6]. In this paper, an extended LogP model is used, which contains nine parameters: $Lo_so_rg_sg_rGP$ [9]. L and P is the same as in LogP model, and the new parameters has the following meaning:

1. o_s: overhead on sender side, the time that the host processor is engaged in sending a message and cannot do any other work. o_r: overhead on receiver side.
2. o_{sb}: overhead on sender side using buffer, the time that the host processor is engaged in copying user data to library buffer. o_{rb}: overhead on receiver side using buffer.
3. g_s: gap on sender side, the minimum time interval between consecutive message transmissions at a node. g_r: gap on receiver side.
4. G: time per byte for a long message.

For most systems, $o_s >> o_{sb}$ and $o_r >> o_{rb}$, since message buffering only runs in user level and is very simple, while copying to network interface takes a lot of time on packaging and calculating header fields by protocol code, which runs in kernel level.

In this section, a theoretical analysis of basic MPI point-to-point operations is given by comparing single-threaded implementation with multi-threaded one, and some performance expectations are deduced. In the following sections, *st* means single-threaded, and *mt* multi-threaded.

4.1. Evaluation of MPI_SEND

For single-threaded MPI_SEND, under the extended LogP model, the time spent in MPI_SEND using eager/rendezvous protocol from one process to another in different nodes takes $t_{send-st}$:

$$t_{send-eager-st} = o_s + L + o_s \quad (1)$$

$$t_{send-rndv-st} = o_s + (k-1)G + L + o_s \quad (2)$$

In multi-threaded model, when there is available library buffer, MPI_SEND copies the message to buffer and returns immediately, the time spent in MPI_SEND t_{mt} is:

$$t_{sent-mt} = o_{sb} \quad (3)$$

According to the comparison given above $o_s >> o_{sb}$, an expectation can be given:

If there is library buffer, multi-threaded MPI_SEND is much faster than single-threaded one.

4.2. Evaluation of MPI_BSEND

As mentioned in section 3.3, single threaded MPI_BSEND is implemented the same as MPI_SEND. So the equation (1) and (2) can be reused here. In multi-threaded model, MPI_BSEND is actually implemented as one scenario of MPI_SEND, and its duration time is also o_{sb}, so the second expectation is as follows:

The multi-threaded MPI_BSEND is much faster than single-threaded one.

4.3. Evaluation of MPI_ISEND

For non-blocking send, the time spent in MPI_ISEND is the same for both single-threaded and multi-threaded:

$$t_{isent} = o_s \quad (4)$$

But the application using MPI_ISEND may behave differently. Fig 3. shows the normal usage of MPI_ISEND: user code first calls MPI_ISEND and returns back to do some computing and then calls MPI_WAIT to make sure the send operation is finished. In single-threaded process, the send operation only progresses in MPI_ISEND and MPI_WAIT, so the total time spent is the summary of $t1$, $t2$ and $t3$. For multi-threaded process, after returned from MPI_ISEND, send operation continues concurrently with user code, so when user code calls MPI_WAIT, fewer steps remain for send operation and MPI_WAIT may take much less time.

Here comes the third expectation:

The time spent in MPI_ISEND is the same for single-threaded and multi-threaded implementation, but the application using MPI_ISEND and MPI_WAIT may run faster.

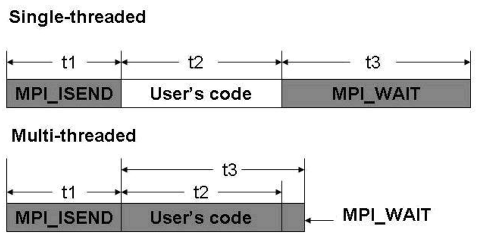

Figure 3. MPI_ISEND and MPI_WAIT

4.4. Evaluation of MPI_RECV, MPI_IRECV

As a receive operation, MPI_RECV can be called in two scenarios, which are the matching message having already arrived and not arrived yet. For the first scenario, MPI_RECV only fetch the matching message from event queue, so the time spent $t_{recv-event}$ is:

$$t_{recv-event} = o_{rb} \quad (5)$$

But for the second scenario, MPI_RECV must wait t_{wait} period for the sender to call send function and then execute the whole receive operation, thus the time $t_{recv-request}$ is:

$$t_{recv-request} = t_{wait} + t_{sent} \quad (6)$$

Obviously, $t_{recv-request}$ is much longer than $t_{recv-event}$.

In single-threaded and multi-threaded MPI, those two scenarios have the same operations and take the same time. But the possibility of the appearance of the first scenario is not that same. For multi-threaded MPI, the first scenario happens more frequently. The worker thread continuously polls every connected socket, and receives message as long as there is one. On the contrary, in single-threaded MPI, only if some previous receive operation polls the same socket can lead to receive an un-posted message, and this happens comparatively rare.

In single-threaded MPI the corresponding send operation, if it is blocking, must wait for the receive operation to be posted, and waste a lot of time doing nothing at all, while multi-threaded MPI_SEND sends message to receiver process's worker thread. Fig.4 shows this situation.

5. Results

This section presents the performance of point-to-point communication in multi-threaded and single-threaded model. In order to show the difference only incurred by multi-threading, tests are experimented on two similar implementations, single-threaded version (st) and multi-threaded version (mt). The two are almost the same, except for point-to-point communication model.

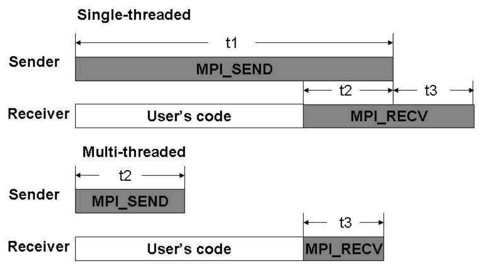

Figure 4. Communication time between sender and receiver

The experiments are executed on a PC cluster, whose nodes are connected by 100M Ethernet. Each node has a Pentium IV 2.4GHz CPU and 512M memory, and the operation system is Redhat Linux. Pthread library used is the LinuxThreads.

5.1. Blocking Send operations

Fig.5, 6 shows the time spent in MPI_SEND operation for both st and mt implementation. Although the time in MPI_SEND increases when the message length grows, multi-threaded version $t_{sent-mt}$ is always less than single-threaded time, either when the message is short or long. This result proves the first expectation given in section 4.1 is true. Because the library buffer used in current implementation is 1 MB, the exhaustion of buffer does not appear.

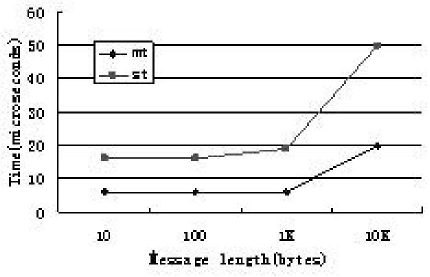

Figure 5. MPI_SEND time for short message

In st, when the message length exceeds 64K bytes, the send time increases extensively, from 50 microseconds at 10K to 3 seconds. This is because the message is long enough to use rendezvous protocol to interact with the receive end. Contrastively, send operation in mt keeps low latency since it only copies the message to library buffer.

According to the analysis in section 4.2, MPI_BSEND in st is similar to MPI_SEND in st and MPI_BSEND in mt is

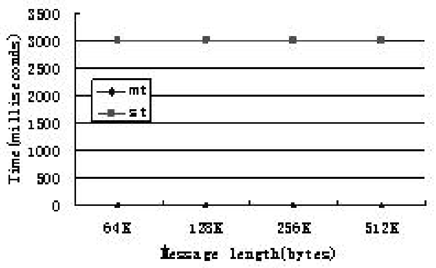

Figure 6. MPI_SEND time for long message

Operation	MPI_ISEND		MPI_WAIT	
Length	ST	MT	ST	MT
10	52	6	15	15
100	63	6	14	16
1K	66	6	15	15
10K	147	20	14	14
64K	205	167	13	17
128K	62	343	2063322	15
256K	100	852	2187928	16
512K	87	2017	2487301	15

Table 1. Non-blocking send operation time (microsecond)

similar to MPI_SEND in mt, so the results are similar and not mentioned again.

5.2. Non-Blocking Send operations

Evaluating the performance of non-blocking send operation should take the time spent in MPI_ISEND and MPI_WAIT into consideration. Actually the time spent on user code between MPI_ISEND and MPI_WAIT may also affect the result, but in this paper this time is set manually to 2 seconds. Tab. 1 shows the results for different message length. Also MPI_ISEND finishes in the same period, MPI_WAIT behaves differently. As time in MPI_WAIT increases badly in st, MPI_WAIT in mt always takes several microseconds, since major operation is finished between the 2-seconds' user code.

5.3. Receive operations

As shown in section 4, MPI_RECV should be same for multi-threaded and single-threaded model. But the two scenarios for MPI_RECV vary a lot. Fig.7 shows that in the first scenario the receive operation is posted after the message has arrived, MPI_RECV keeps costing only dozens of microseconds, but in the second scenario, MPI_RECV costs a lot time for long message.

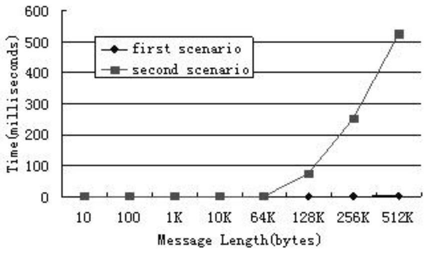

Figure 7. MPI_RECV operation time

According to the analysis in section 4.4, the possibility of the first scenario's happening is much larger in mt than in st, so the averaged performance of MPI_RECV is enhanced in multi-threaded model.

6. Conclusions and Future Work

From the theoretical analysis and experiment results, an overall conclusion that a multi-threaded model achieves a higher point-to-point MPI communication performance can be made. Although for some scenario multi-threaded architecture is not better than the single-threaded one, most of the time it dose behave much better. In this paper only simple send/recv operations are tested, but the proposed architecture is very likely to show wonderful result in real application on larger scale cluster.

The architecture described in this paper consists of two threads, a worker thread and a user thread. Some research proposes allocating a thread for every connection, which is commonly called multi fabric communication. This architecture may achieve surprising results for collective operations.

Portability is another issue for multi-threaded program. Although Pthread[2] has already been accepted across operating systems, there are still platforms which don't support it. For those platforms , point-to-point module must be reimplemented.

References

[1] Infiniband trade association. *http://www.infinibandta.org*, 1999.

[2] U. S. 9945-1:1996. *Information Technology -Portable Operating System Interface (POSIX) Part 1: System Application Program Interface (API) [C Language]*. Institute of Electrical and Electronic Engineers, 1996.

[3] N. J. Boden, D. Cohen, R. E. F. A. E. Kulawik, C. L.Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A gigabit-per-second local area network. *IEEE Micro*, 15(1):26–39, Feburary 1995.

[4] R. Brightwell and K. D. Underwood. An analysis of the impact of mpi overlap and independent progress. *Proceedings of the 18th annual international conference on Supercomputing*, pages 298–305, 2004.

[5] G. Burns, R. Daoud, and J. Vaigl. Lam: An open cluster environment for mpi'. *Proc. of Supercomputing Symposium*, pages 379–386, 1994.

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: Towards a realistic model of parallel computation. *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'93)*, 1993.

[7] M. P. I. Forum. Mpi2: A message passing interface standard. *International Journal of High Performance Computing Applications*, 12(1-2):1–299, 1998.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[9] E. A. Len, A. B. Maccabe, and R. Brightwell. Instrumenting logp parameters in gm: Implementation and validation. *LCN*, pages 648–657, 2002.

[10] B. Lewis and D. J. Berg. *Threads Primer: a guide to multithreaded programming*. SunSoft Press, Mountain View, CA, 1996.

[11] B. V. Protopopov and A. Skjellum. A multithreaded message passing interface (mpi) architecture: Performance and program issues. *J. Parallel Distrib. Comput.*, 61(4):449–466, 2001.