ADAPTIVE STRATEGY FOR ONE-SIDED COMMUNICATION IN MPICH2

BY

XIN ZHAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Adviser:

Professor William D. Gropp

Abstract

The one-sided communication model (or remote memory access) supported by MPI-2 is more convenient to use than the regular two-sided send/receive communication model, because it allows the sender to specify all data transfer parameters and avoids the receiver to be explicitly involved in data receiving. One-sided communication model also has potential to provide higher performance. The MPI-2 standard provides flexibility about when RMA operations can be issued and completed, which makes the MPI implementation possible to be optimized internally. The current MPICH2 implementation uses a lazy approach to issue one-sided operations by queuing the operations and issuing them during the later synchronization phase. This has certain benefits with respect to short operations in terms of reduced network operations. For large data transfers, issuing operations in an eager fashion could be more beneficial as well as provide more scope for overlapping communication and computation. In this thesis we describe the design and implementation of an adaptive approach for all three synchronization mechanisms defined in MPI: fence, post-start-complete-wait, and lock-unlock. We evaluate our implementation with both micro benchmarks and the Graph500 benchmark to demonstrate the performance impact of our approach. The performance results show that our hybrid approach performs as good as the lazy approach for small data transfers and achieves performance similar to the eager approach for large data transfers, and its overlapping percentage is good.

Acknowledgments

This work is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois. This work is supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835.

This thesis is based on previous research conducted together by Gopal Santhanaraman, I thank him a lot for giving me many advices. I highly appreciate help and support of my advisor, Prof. William Gropp, and all our group members. Finally, sincere thanks to my family for their endless patience, love and support.

Table of Contents

Page

List of Figures . . . v

Chapter

1 Introduction . . . 1

1.1 Motivation . . . 1

1.2 One-sided Communication in MPI . . . 2

1.3 MPI-3 Standard . . . 5

2 Related Work . . . 6

3 Adaptive Approach Design . . . 7

3.1 Active Target Synchronization . . . 7

3.1.1 Fence . . . 7

3.1.2 Post-Start-Complete-Wait (PSCW) . . . 10

3.2 Passive Target Synchronization . . . 12

4 Performance Results . . . 17

4.1 Micro-Benchmarks . . . 17

4.1.1 Single-op Results . . . 18

4.1.2 Many-ops Results . . . 20

4.1.3 Overlapping Results . . . 21

4.2 Graph500 Benchmark . . . 23

5 Conclusion and Future Work . . . 27

References . . . 29

List of Figures

1.1 Fence 4

1.2 SPCW 4

1.3 Lock-Unlock 4

3.1 LAZY fence 8

3.2 EAGER fence 8

3.3 HYBRID fence 9

3.4 LAZY pscw 11

3.5 EAGER pscw 11

3.6 HYBRID pscw 11

4.1 Single-op latency on SMP for (a)PUT and (b)GET 19

4.2 Single-op latency on breadboard for (a)PUT and (b)GET 19

4.3 Many-ops latency on (a) SMP and (b) breadboard 21

4.4 Overlapping Results(lock-unlock) for (a) PUT and (b) GET . . . 22

4.5 Overlapping Results(pscw) for (a) PUT and (b) GET 23

4.6 Overlapping Results(fence) for (a) PUT and (b) GET 24

4.7 Graph500 results with (a) large graph and (b) small graph 25

4.8 Time breakdown for (a) overall time and (b) second fence time . . 26

Chapter 1

Introduction

1.1 Motivation

MPI has gained vast success in bringing message passing programming model to a wide variety of platforms because of its great portability. However, MPI has always been labeled as a communication model that mainly supports "two-sided" and "global" communication. The MPI-2 standard, released by 1997, added support for one-sided communication (or remote memory access) capabilities. One-sided communication allows one process to specify all communication parameters, both for the sending side and for the receiving side, which is more convenient to use, because it avoids the needs for global distribution of transfer parameters and explicitly polling to receive data. One-sided communication also has the potential to deliver higher performance than regular two-sided communication, particularly on networks that support one-sided communication natively.

MPI provides three different synchronization mechanisms for one-sided communication: lock-unlock, fence and post-start-complete-wait. These synchronization mechanisms ensure the correct semantics of one-sided operations. The MPI-2 standard gives much flexibility on when a one-sided operation can complete, which makes MPI implementation possible to be optimized internally, according to different number of operations or sizes of transferred data within one communication epoch (an epoch is the period between synchronization calls).

For small number of operations with short data, issuing them in the late synchronization phase instead of the communication epoch between two synchronization can effectively reduce the communication cost, because it provides opportunities to combine the operation message with the synchronization message, or to reduce the total number of collective communication. For large number of operations with significant amount of data, issuing them as early as possible is better, because their transmission latency is expensive, and issuing them early provides opportunities to overlap communication with computation within the epoch.

In many situations, it is not obvious beforehand whether issuing operations early or late is better due to the nature of communication pattern. Therefore, it is important to design an adaptive strategy that automatically select the suitable strategy based on runtime situation. This adaptive design would remove user's need to understand different implementation choices in detail, and eliminates the possibility of making mistake by the user since the MPI runtime system can adaptively monitor and choose the best approach. In this paper, we address this issue by designing and implementing an adaptive approach for One-sided communication in MPI, including fence, start-complete-post-wait and lock-unlock. Our goal is to perform well with either small or large transferred data by adaptively select the suitable approach to issue operations during the runtime.

1.2 One-sided Communication in MPI

In MPI one-sided communication model, any allocated memory is private to the MPI process by default, and can be exposed to other processes as a public memory region. This public memory region is called a *window*. After *window* is created, communication and synchronization could be performed between senders and receivers. Unlike regular two-sided communication model, one-sided model separates the communication of data transfer and synchronization of sender with receiver.

MPI-2 supports three types of data transfer: `MPI_Put` (remote write), `MPI_Get` (remote read) and `MPI_Accumulate` (remote update). MPI-3 standard further extends the set of one-sided communication operations by adding request-based operations and remote atomic operations. We shall denote by *origin* the process that performs the call, and by *target* the process in which the memory is accessed. The one-sided communication calls in MPI are nonblocking functions. They initiate the operation, but when the function call return, the completion of operation is not guaranteed. To specify when a one-sided operation can be initiated and when it is guaranteed to be completed, MPI defines different synchronization mechanisms, and they are categorized into *active target* and *passive target*.

In active target communication, both the origin and the target are explicitly involved in the synchronization, but data transfer arguments are only provided by the origin process. MPI provides two mechanisms for active target communication:

(1) **Fence**: Figure 1.1 illustrates the method of fence synchronization. Fence function call is a collective operation over the communicator associated with *window*. After the first fence call returns, a process can issue one-sided operations, and the next fence call would guarantee the completion of all operations issued between two fence calls. This mechanism is useful for loosely synchronous algorithms where the graph of communicating processes changes frequently, or where each process communicates with many other processes [1]. If only a small number of processes but not all processes in the communicator are actually communicating with each other, the collective fence call over the entire communicator will lead to unnecessary synchronization overhead.

(2) **Start-Complete-Post-Wait**: Figure 1.2 illustrates the method of start-complete-post-wait. It restricts the synchronization and avoids the drawback of fence call, because the synchronization only involves pairs of communicating pro-

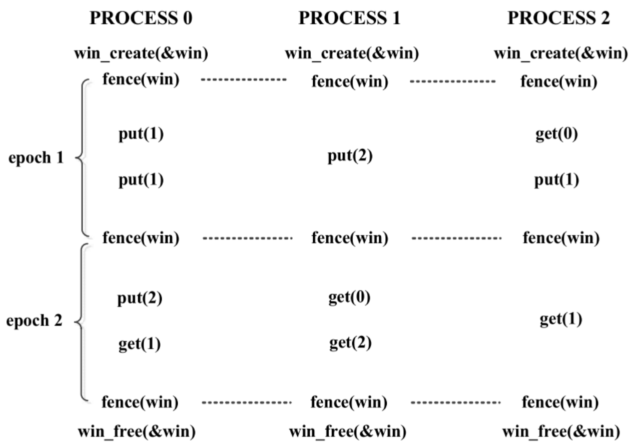


Figure 1.1: Fence

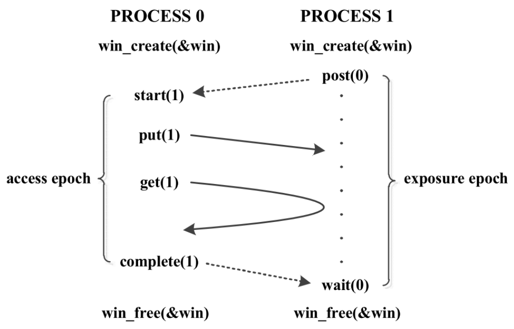


Figure 1.2: SPCW

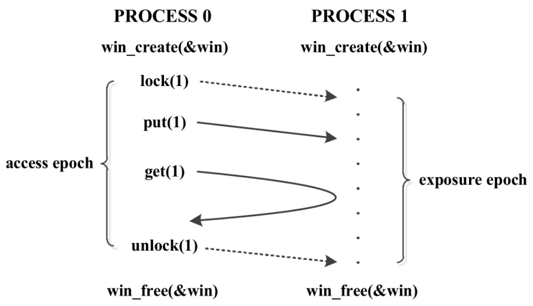


Figure 1.3: Lock-Unlock

cesses. As shown in Figure 1.2, the origin process start an access epoch by calling `MPI_Start` associated with a group argument which specifies the group of target processes that this epoch can access, and terminated it by calling `MPI_Complete`; the target process start an exposure epoch by calling `MPI_Post` associated with a group argument which specifies the group of origin processes that can access this epoch, and terminated it by calling `MPI_Wait`. This mechanism may be more efficient when each process communicate with only few neighbors, and the communication graph is fixed or changed infrequently [1].

In passive target communication, the target process does not explicitly participate in communication and synchronization. Only the origin process calls the synchronization function and specifies data transfer arguments. **Lock-Unlock** is the mechanism provided by MPI to achieve passive target communication. As is illustrated in Figure 1.3, the origin process first calls `MPI_Win_lock` function on target process to obtain shared or exclusive lock of *window*, after that, it can issue one-sided operations on the target process, and the `MPI_Win_unlock` call at last would guarantee the completion of all operations on both the origin and the target processes. Lock synchronization is useful for applications that emulate a shared memory model via MPI calls where processes can access or update different parts of the shared memory region at random times [1].

1.3 MPI-3 Standard

The MPI forum is currently working on the MPI-3 standard, which improve the semantics and extends new features of MPI one-sided operations to make MPI RMA a portable runtime system that can provide high-performance one-sided communications with rich features. Our work here will apply to the proposed MPI-3 RMA model, including the new interfaces being proposed.

Chapter 2

Related Work

There are several studies regarding the implementation of one-sided communication in MPI-2. Some MPI-2 implementations which support one-sided communication are MPICH2 [2], OpenMPI [3] and NEC [4]. Besides MPI, other programming models that also provide one-sided communication include CRAY SHMEM [5], ARMCI [6], GASNET [7] and BSP [8].

Some BSP papers, particularly [9, 10], discuss the benefits of aggregating and scheduling communication operations for better performance as well as contention avoidance. Other papers, like [11, 12], described the design choices and issues in implementing one-sided communication in MPI. The authors in [13] have studied optimizations for reducing the synchronization overhead involved in implementing one-sided communication. Designs for MPI RMA in InfiniBand clusters is described in [14] [15]. In [16] [17], the author describes a design for efficient passive synchronization using hardware support from InfiniBand atomic operations. [18] discusses some performance guidelines for one-sided communication in MPI.

Chapter 3

Adaptive Approach Design

3.1 Active Target Synchronization

3.1.1 Fence

A valid implementation of fence synchronization should obey the correct semantics as follows: An one-sided operation from the origin process cannot access the target process's window until that target process has called fence; the next fence on a process cannot return until all origin processes that need to access that process's window have completed doing so.

The current fence synchronization in MPICH-2 is implemented in a *lazy* fashion, as is shown in Figure 3.1. In the first fence, each process does nothing. For the following one-sided operations, each process does not issue them out but locally queues them up. In the second fence, each process first goes through its operation queue to determine, for each other process i, whether any of operations have i as the target, and stores this information in an array. Then all processes perform a reduce-scatter sum operation over this array. After that each process knows how many processes having operations targeted at itself and stores this number in a local counter. Now each process can issue all queued operations, and the counter at each process is decremented when all one-sided operations from one process have been arrived at that process. To achieve this, the origin process sets a field in the packet header of the last operation and the target will decrement the counter when it receives this operation.

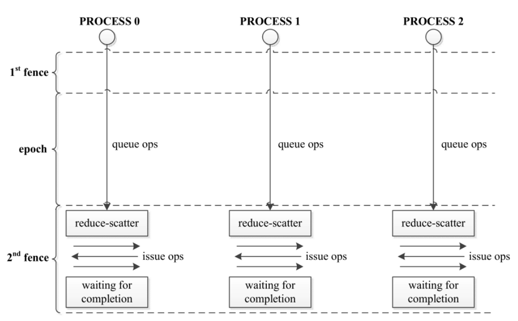


Figure 3.1: LAZY fence

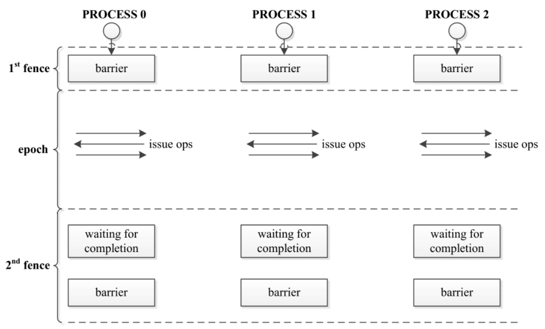


Figure 3.2: EAGER fence

Another choice of implementing fence is an *eager* approach, which means all one-sided operations are issued as early as possible. As shown in Figure 3.2, in the first fence, all processes perform a collective barrier to synchronize over the communicator. After that, each process issues one-sided operations as soon as possible. At the second fence, after all one-sided operations have been completed, all processes again perform a collective barrier to guarantee that no process leaves the second fence before all other processes have finished accessing the window. Compared with the *lazy* design, this *eager* approach is more expensive due to the collective communication, since it involves two barriers whereas the *lazy* approach only requires one reduce-scatter. However, the *eager* approach issues one-sided operations earlier than the *lazy* approach and it has no cost on processing the local operation queue.

Our *hybrid* strategy combines benefits from both eager and lazy approaches while introducing little overhead. If numbers of local operations on each process are small(no one reaches the queuing threshold), the *hybrid* approach will work in the same way with the *lazy* method. However, as long as the local number of operations reaches the queuing threshold, the process will automatically switch from *lazy* to *eager* mode between fences and issue one-sided operations immediately. The design of *hybrid* approach is shown in Figure 3.3. At the first fence,

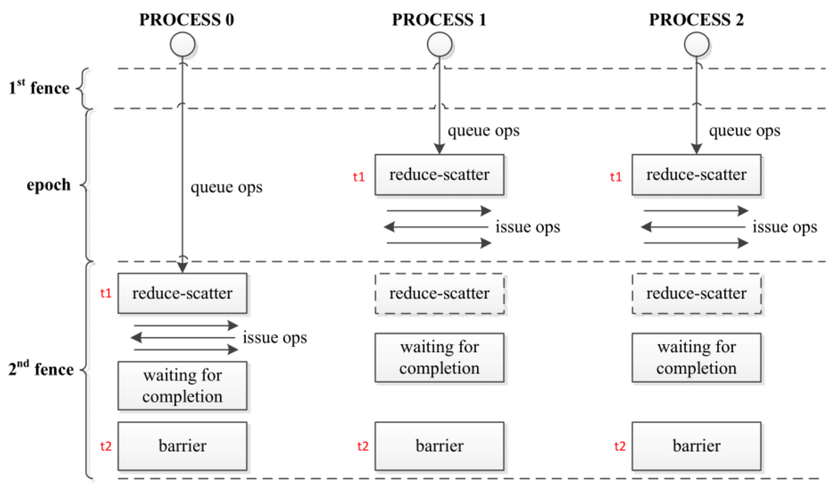


Figure 3.3: HYBRID fence

each process does nothing, like in the lazy approach. After the first fence returns, for each process, if its number of local operations does not reach the threshold, it just queues them up and goes into the reduce-scatter at the second fence(like process 0). If the number of local operations reaches the threshold(like process 1 and process 2), that process will go into the reduce-scatter before the second fence(during communication epoch) and synchronize with those processes who are in the reduce-scatter in the second fence. After synchronization, they issue all previous queued operations and the following operations immediately. In Figure 3.3, process 0, 1 and 2 are synchronized by reduce-scatter at the same time *t1*, but in different function calls. Process 0 is in fence, whereas process 1 and 2 are in one of one-sided operations. If one process has already performed reduce-scatter in the communication epoch, like process 1 and 2, it does not need to perform it again at the second fence, in another word, every process just perform one reduce-scatter. If all processes perform reduce-scatter at the second fence, they do not need to do a barrier at end and the second fence function, therefore only one synchronization is needed, just like in the *lazy* approach. If there is at least one process that performs reduce-scatter in the communication epoch, then all processes must synchronize again at the end of the second fence(*t2* in Figure 3.3). In this situation, two synchronizations are needed(one reduce-scatter and one barrier), like in the *eager* approach. We use a field in the reduce-scatter message to indicate if there are processes that perform reduce-scatter before the second fence. For example, in Figure 3.3, when process 0 sees this field, it will know it needs to do a barrier at end, even though itself does not issuing operation eagerly.

3.1.2 Post-Start-Complete-Wait (PSCW)

The correct semantic of PSCW synchronization should be as follows: a one-sided operation cannot access a process's window until that process has called `MPI_Win_post`, and one process cannot return from `MPI_Win_wait` until all processes that need to access that process's window have completed doing so and called `MPI_Win_complete`.

Like fence, the current implementation of PSCW in MPICH-2 uses a *lazy* approach. As is indicated in Figure 3.4, for any process in the origin group, in `MPI_Win_start`, it does nothing. Puts, gets and accumulates are queued locally. In `MPI_Win_complete`, the process is blocked until it receives post messages from all processes in the target group, after that it issues all operations in the queue. For each target process, the origin process set a field in the header of corresponding last message to decrement that target process's counter. If the origin process has no operation targeting at a process in the target group, then it needs to send an extra 0-byte message to that target process. `MPI_Win_complete` returns after all operations are completed locally. For any process in the target group, in `MPI_Win_post`, it sends post messages to every process in the origin group. In `MPI_Win_wait`, it is blocked until its counter reaches 0, which means it has received the last messages from all processes in the origin group.

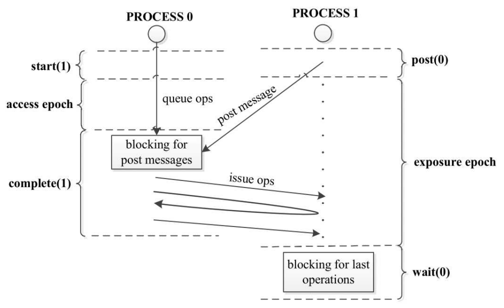


Figure 3.4: LAZY pscw

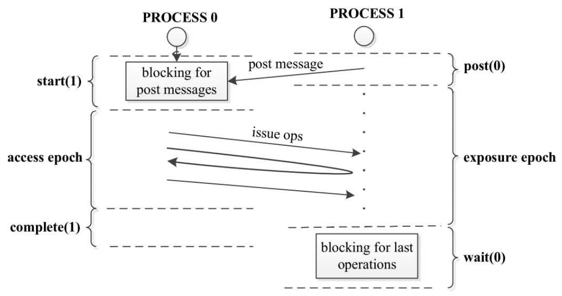


Figure 3.5: EAGER pscw

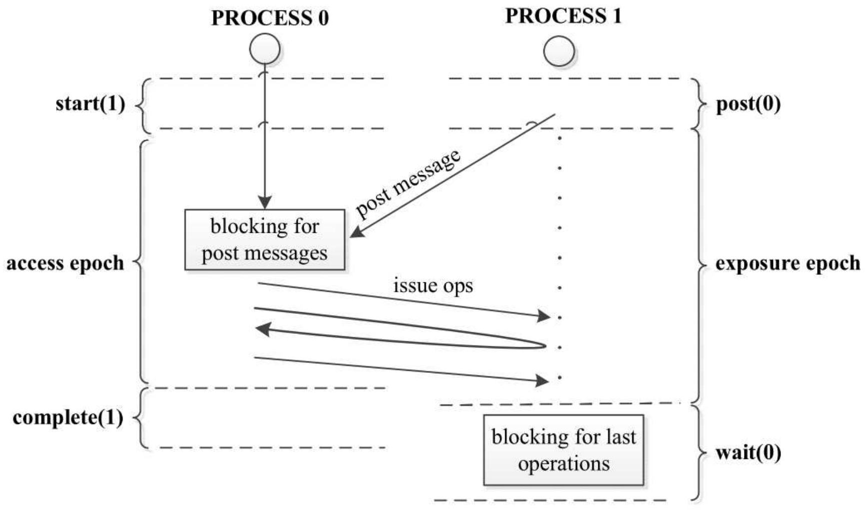


Figure 3.6: HYBRID pscw

The *eager* choice of PSCW synchronization is similar with the one in fence. As shown in Figure 3.5, `MPI_Win_start` (or the first operation) blocks until the origin process receives all post messages. Puts, gets and accumulates are issued as soon as possible without queuing. `MPI_Win_complete` waits until all operations have locally completed, and `MPI_Win_wait` blocks until it receives last messages from all processes in the origin group. Since operations are issued eagerly, the origin process have to send an additional last message to all processes in the target group, which involves more synchronization messages than the *lazy* approach.

Our design for *hybrid* maintains only one synchronization if local operation number is small, just like in *lazy* approach, and needs two synchronizations if local operation number reaches the threshold and functions are switched from *lazy* to *eager*. Figure 3.6 shows our design for *hybrid* approach. In `MPI_Win_start`, the process does nothing. For the following operations, the process initially queues them up. When the number of queued operations reaches the threshold, the process is blocked until it receives post messages from those queued operation's targets, after that it issues all queued operations. For every new following operation, the process first checks if itself has already received the post message from that operation's target. If not, it is blocked, else it just issues that operation. In `MPI_Win_complete`, if the process hasn't received the post message from some target processes, which means it has no operations destined at those target processes, it is blocked. After that it sends an additional last messages to those target processes and then returns. We avoid sending last messages to all target processes by keeping a `last_rma_op` pointer for each target process, and only this operation is issued at `MPI_Win_complete`.

3.2 Passive Target Synchronization

An implementation of passive target synchronization should follow the correct semantics below: A one-sided operation from the origin process cannot access the target process's window until the origin process has called `MPI_Win_lock` and acquired the lock of that window, and `MPI_Win_unlock` cannot return until all one-sided operations are completed at both origin and target.

Like fence and pscw, the current passive target synchronization in MPICH-2 is also implemented in a *lazy* fashion. In `MPI_Win_lock`, the origin process just queues up the lock request locally and does nothing else. Following one-sided operations are also queued locally. In `MPI_Win_unlock`, the origin process first issues the lock request. After lock is granted, it issues all queued operations to the target. The last one-sided operation, indicated by a field in the packet header, causes the target process to release the lock on the window. This strategy eliminates the overhead of the first synchronization and reduces the overhead of the second synchronization by combining the last operation message with the synchronization message. The implementation also includes an optimization for single short operation: if there is only one short operation between lock and unlock calls and datatype is predefined at the target, the origin process sends the operation data together with the lock-request packet. The drawback of this *lazy* approach is that since communication is deferred to the unlock phase, it is impossible to achieve any overlapping of computation and communication between lock and unlock calls, and additional overhead is added due to processing the operation queue when there are large number of operations.

Another choice is to implement passive target synchronization in an *eager* fashion, which is very similar to the one in pscw synchronization. In `MPI_Win_lock`, the origin process issues the lock request immediately. For each following one-sided operation, the origin process pokes the progress engine to check if currently the lock is granted. If the lock is not yet granted, it continues queuing up the operation and returns immediately; otherwise it issues all queued operations instead of waiting until the unlock phase. In `MPI_Win_unlock`, the origin process needs to send an additional last message to the target process. This approach also allows the first synchronization call to return without blocking, but an additional synchronization message must be sent to the target process after all operations are performed. However, it allows the overlapping of computation with communication between lock and unlock calls.

In this *eager* strategy, there is no optimization for single short operations as is done in the *lazy* approach, because the lock-request packet must be issued in `MPI_Win_lock`, which cannot be sent together with the operation package.

We use a simple model to estimate the latency of one-sided operations in passive synchronization with both *lazy* and *eager* strategies. We assume that the time taken to send a message between two nodes can be modeled as $\alpha + n\beta$, where α is the startup time per message, and β is the transfer time per byte. n is the number of bytes for transmitted data in one message, and it is short enough to be eligible for the optimization. In case of get operations, there is a get-request packet sent to the target followed by data transfer from the target process. We assume that number of issued operations is m, and data size of get-request, lock-request, lock-reply, additional last message (eager) and acknowledgment packet is c bytes, which is quite small. All these internal messages would take on the order of a microsecond or more, even on a fast interconnect network.

We first analyze the latency for single short operation. For the *lazy* approach:

$$\begin{aligned} T_{put} &= T_{lock\_req+data} + T_{ack} = 2\alpha + (2c+n)\beta \\ T_{get} &= T_{lock\_req+get\_req} + T_{data} = 2\alpha + (2c+n)\beta \end{aligned} \tag{3.1a}$$

For the *eager* approach:

$$\begin{aligned} T_{put} &= T_{lock\_req} + T_{lock\_reply} + T_{data} + T_{last\_msg} + T_{ack} \\ &= 5\alpha + (4c+n)\beta \\ T_{get} &= T_{lock\_req} + T_{lock\_reply} + T_{get\_req} + T_{last\_msg} + T_{data} \\ &= 5\alpha + (4c+n)\beta \end{aligned} \tag{3.1b}$$

We see that when there is only one short operation, the single operation optimization in *lazy* approach can result in significant reduction in total number of messages and therefore causes less startup overhead. Hence the time complexity is much better than the eager approach. In fact this suggests there might be a case for merging as many operations as possible into one message, though this might involve additional space for the aggregated message and the cost of copy. This is not currently implemented in the *lazy* approach in MPICH2.

Next we analyze the latency when there are more than one operations between lock and unlock calls. For the *lazy* approach with many operations:

$$\begin{aligned} T_{put} &= T_{lock\_req} + T_{lock\_reply} + mT_{data} + T_{ack} + T_{queue} \\ &= (3+m)\alpha + (3c+mn)\beta + (m+2)t_{queue} \\ T_{get} &= T_{lock\_req} + T_{lock\_reply} + mT_{get\_req} + mT_{data} + T_{queue} \\ &= (2+2m)\alpha + (2c+mc+mn)\beta + (m+2)t_{queue} \end{aligned} \tag{3.1c}$$

For the *eager* approach with many operations:

$$\begin{aligned} T_{put} &= T_{lock\_req} + T_{lock\_reply} + mT_{data} + T_{last\_msg} + T_{ack} \\ &= (4+m)\alpha + (4c+mn)\beta \\ T_{get} &= T_{lock\_req} + T_{lock\_reply} + mT_{get\_req} + T_{last\_msg} + mT_{data} \\ &= (3+2m)\alpha + (3c+mc+mn)\beta \end{aligned} \tag{3.1d}$$

For the *eager* approach, the time complexity is almost the same as the one in *lazy* approach for many operations, except for the latency of additional last message. On the other hand, for the *lazy* approach, there is additional overhead due to queue processing, which can become significant depending on number of operations.

Our strategy for the passive target synchronization is a *hybrid* approach which is similar to pscw and combines the benefits of both lazy and eager strategies. In `MPI_Win_lock`, we do nothing but just queue up the lock request locally. In the very first one-sided operation, we also queue up the operation. If there are more than one operations to be issued, we issue the lock request in the second operation and poke the progress engine in every operation to check if currently lock is granted. If lock is not granted, we continue queuing up operations, and we use a `last_rma_op` pointer to always keep the last RMA operation. The run-time can automatically switch between lazy and eager mode by using a criteria of threshold value for number of operations in the queue. If current number of queued operations reaches this value and the lock has been granted, the origin process issues all operations to the target, except for the last RMA operation (pointed by `last_rma_op`); otherwise it continues queuing. For the experiments in this paper, the threshold value is set to 1 in lock-unlock case, therefore once the lock is granted, the operations are issued immediately. This threshold value can be tuned for a particular system. In `MPI_Win_unlock`, if lock has already been granted in previous operations, we just issue the last RMA operation; otherwise we wait for lock to be granted and then issue all queued operations and the last operation. When the target process receives this last operation, it releases the lock on its window. Our approach preserves the optimization for single short operation, because if there is only one short operation and also avoids sending the additional last synchronization message because the last operation is always kept in the `last_rma_op` pointer and will be sent in the unlock phase. Furthermore, since we issue most operations in an eager fashion(except for the single short operation and the last operation), overlapping of computation and communication can be achieved.

For all three types of synchronization above, we added a new configure option (`--enable-hybridrma`) into MPICH2, which can be used to set the runtime system at *hybrid* mode.

Chapter 4

Performance Results

We implemented the *adaptive* approach for MPI one-sided communication based on MPICH2-1.4.1p1 release. For each synchronization strategy (fence, lock-unlock, post-start-complete-wait), we additionally implemented the *eager* method (with `--enable-hybridrma` option) described in section 3.1 and section 3.2, and compared performance of our *hybrid* approach with both the *eager* approach and the origin *lazy* approach in MPICH2. Three implementations are labeled as LAZY, EAGER and HYBRID in the following content.

We run tests on two different systems: (i) An SMP machine with 4 Intel Core i5 CPU (2.67 GHz) and 8GB memory on which the communication latency is very low, therefore we use it to simulate the architecture with very fast interconnect network; (ii) "breadboard" cluster at ANL on which each node has two Intel Xeon quad-core processors (2.66 GHz) and 16GB memory, and nodes are connected with Ethernet. We use "breadboard" to examine the performance on a slow interconnect network.

While all experiments in this section make use of a simple communication layer, the idea applies even to one-sided transports, particularly those that could implement the one-sided semantics by directly exploiting the hardware features.

4.1 Micro-Benchmarks

In this section we present the performance results running with micro-benchmarks. We wrote three types of benchmarks: (i) single-op test, in which only one RMA operation is issued between pair of synchronization calls; (ii) many-ops test, in which more than one RMA operations are issued between pair of synchronization calls; (iii) overlapping test, in which different amount of computation is inserted between synchronization calls, which measures how much computation can be absorbed by overlapping with one-sided communication.

4.1.1 Single-op Results

In this test, we measured the latency of the entire one-sided communication when there is only one one-sided operation between synchronization calls, with transferred data size varying from 1 byte to 64 bytes. We tested Lock-Unlock and SCPW synchronizations and separately tested put, get and accumulate operations. Since results of accumulate operation are very similar with put, due to the space limitation, we do not show its results here.

For lock-unlock, Figure 4.1 shows the results for put and get operations on SMP machine. It is shown that LAZY outperforms EAGER for both put and get. This is expected, because optimization for single short operation is not applicable for EAGER (lock request must send early so that it can never send with operation), but it can be used in LAZY, therefore the number communication messages is reduced. In EAGER case, expect for the separate transmission of lock request and operation message, an additional last message also needs to be sent to indicate the completion. HYBRID performs similar with LAZY, because it preserves the optimization for single short operation and avoid the additional last message as LAZY does.

Figure 4.2 shows the results for put and get operation on the "breadboard" cluster. Performance for both put and get is similar as it is on SMP machine, but the performance gap between LAZY(HYBRID) and EAGER on is more significant, due to the effect of slow interconnect network on "breadboard".

For post-start-complete-wait, both LAZY and HYBRID perform slightly bet-

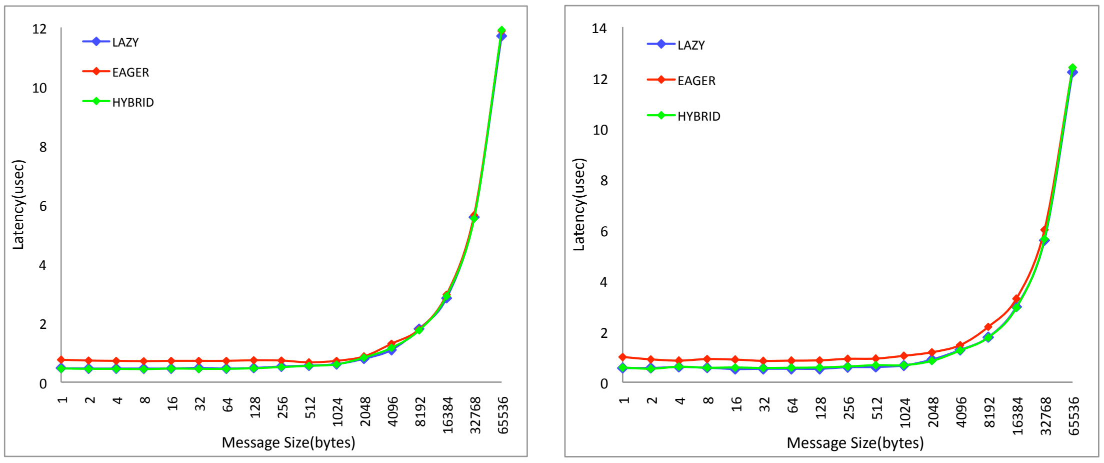


Figure 4.1: Single-op latency on SMP for (a)PUT and (b)GET

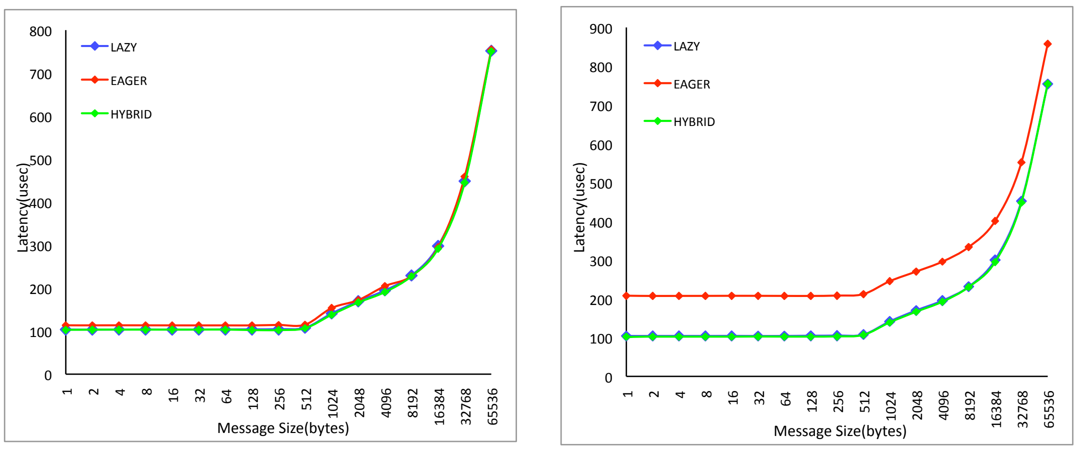


Figure 4.2: Single-op latency on breadboard for (a)PUT and (b)GET

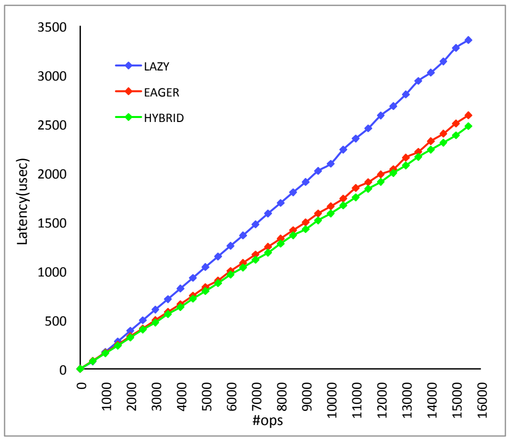
ter than EAGER when tranferred data is small, because both of them eliminates the last additional message. However, the performance gap between LAZY/ HYBRID and EAGER is smaller than it is in lock-unlock, this is because there is no optimization for single operation in post-start-complete-wait, therefore the benefit of LAZY issuing is not that obvious. Since the additional last message contains 0-byte data (only have package header), when the data volume in the operation is large enough, this additional small message can be ignored.

For fence synchronization, the advantage of LAZY(HYBRID) over EAGER is much more obvious than lock-unlock and pscw, either on SMP or on breadboard, this is because EAGER approach has two barriers which is expensive, whereas both LAZY and HYBRID only have one reduce-scatter.

4.1.2 Many-ops Results

In this test, we measure one-sided communication latency when there are more than one RMA operations between synchronization calls. The origin process performs different number of RMA operations (1 to 16000) at the target with 8-byte data transferred for each operation.

Figure 4.3 shows the results for put operations on SMP and "breadboard" machine, running with lock-unlock synchronization. We see that both on SMP system and breadboard, EAGER and HYBRID perform better than LAZY. This is due to the fact that on one hand, queuing overhead is eliminated in EAGER/HYBRID approaches since operations are issued as early as possible after lock is granted; on the other hand, lock-request packet is also issued earlier instead of deferred to the unlock phase, therefore the time taken for waiting for the lock-granted packet can be overlapped with the following RMA operations, whereas in LAZY case, queuing overhead is introduced and lock request is issued late in the unlock phase. Performance results of fence and post-start-complete-wait show similar trends with lock-unlcok in many-ops testing.



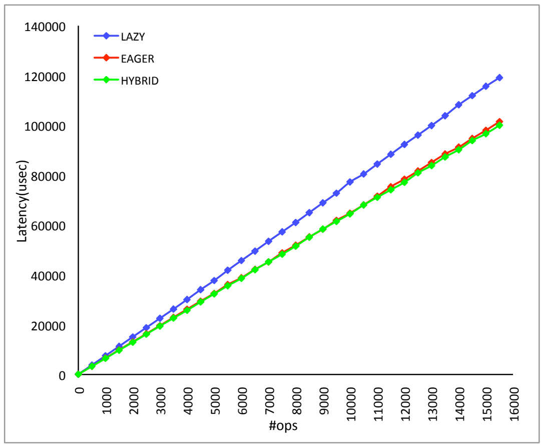


Figure 4.3: Many-ops latency on (a) SMP and (b) breadboard

4.1.3 Overlapping Results

In this test, we measure the overlapping capabilities of LAZY, EAGER and HYBRID for three synchronization mechanisms. In this micro-benchmark, the origin process does a certain number of one-sided operations on the target window. At first, we do not insert any computations between synchronization calls and just measure the entire communication time for lock-ops-unlock, fence-ops-fence and start-ops-complete sequences. After that, we insert certain amount of computation after each one-sided operation. The amount of computation inserted is a percentage of the net communication time measured before. We vary the amount of computation and measure the overall execution time. As long as the overall time does not change, it implies that computation is overlapped with the one-sided communication, or the computation time is hidden by the communication latency. We change the message size of one-sided operations and compute the overlapping percentage corresponding to each message size. We did not run this test on SMP. Since the message transferring is done by memory copy on SMP machine, the communication latency is very short and it is hard to achieve any overlapping results.

Figure 4.4 shows the overlapping results of put and get operations on the Linux

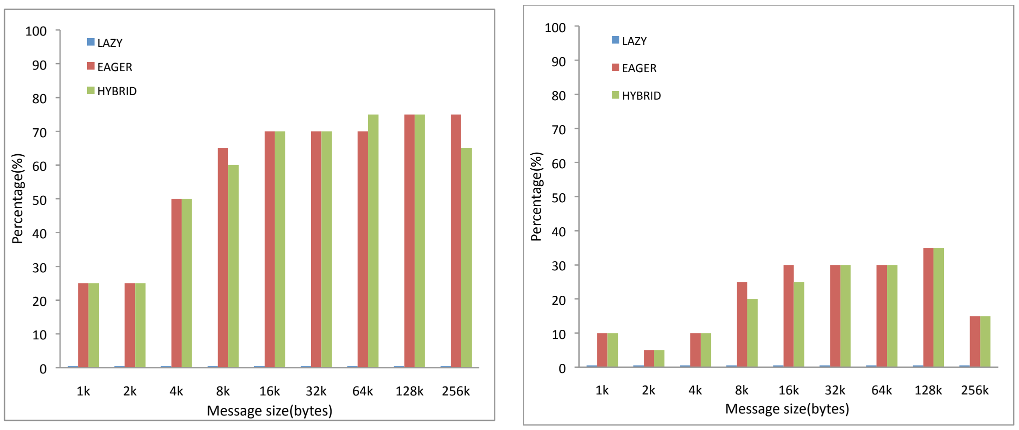


Figure 4.4: Overlapping Results(lock-unlock) for (a) PUT and (b) GET

cluster for lock-unlock synchronization. For put operations, we can see that there is virtually no overlapping achieved for LAZY. This is because all operations are issued in the unlock phase, the communication time cannot be overlapped with the computation inserted before the unlock call. EAGER and HYBRID are able to issue operations as early as possible once the lock is granted, thus as long as there are sufficient operations or computation to ensure that the lock is granted before unlock, operations can be issued early and their communication time can be overlapped with the computation.

We observed up to 75% overlapping percentage for HYBRID when message size is larger than 64KB. For small message size, we do not expect good overlapping results as the startup overhead per message dominates the latency and this part cannot be overlapped. Results of accumulate operations are similar with put operations, so we do not show them here.

For get operations, we could not achieve any overlapping initially irrespective of the message size. After further analysis we found that though we issued operations as early as possible, when the data is returned, the actual receiving of that data occurs only when progress engine is poked. Since we do not poke the progress engine in between, a lot of time is spent at unlock in receiving data, and

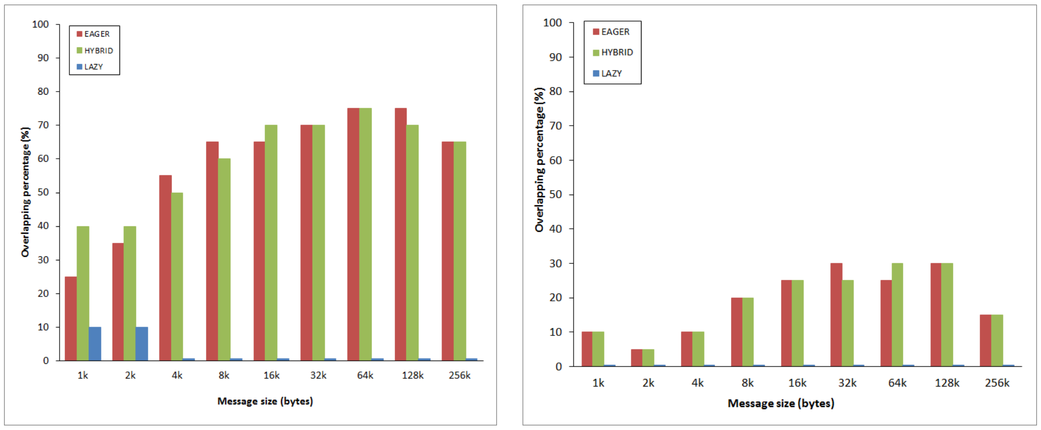


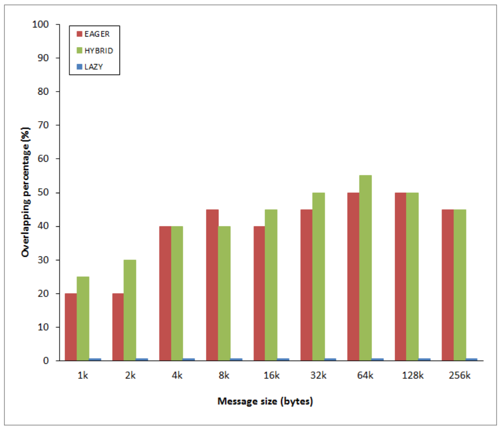
Figure 4.5: Overlapping Results(pscw) for (a) PUT and (b) GET

this time cannot be overlapped by computation. Therefore, we added a progress poking call after each get operation is issued to receive the data early. After that, we could gain overlapping results for EAGER and HYBRID, up to 35% percentage when message size is 128KB. Figure 4.4(b) shows the results for the get operations on the Linux cluster. One thing needed to be noted here is that the overlapping percentage is also limited by the capability of the underlying transport. We can expect higher overlap benefits if there is better support for asynchronous progress. Higher overlapping percentage can also be expected when there is better hardware support in the form of RDMA get.

Post-start-complete-wait achieves similiar overlapping results as lock-unlock does, as shown in Figure 4.5. Fence achieves at most 50% overlapping percentage, as shown in Figure 4.6, which is smaller than the other two, this is because in fence communication, some processes have to spend time waiting at collective synchronization, which can not be overlapped with computation.

4.2 Graph500 Benchmark

We used Graph500 benchmark to test the performance impact of fence implementation on "breadboard" cluster. The benchmark version is 2.1.4, it randomly con-



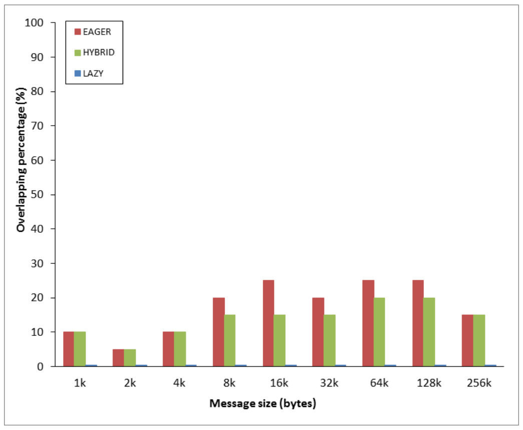


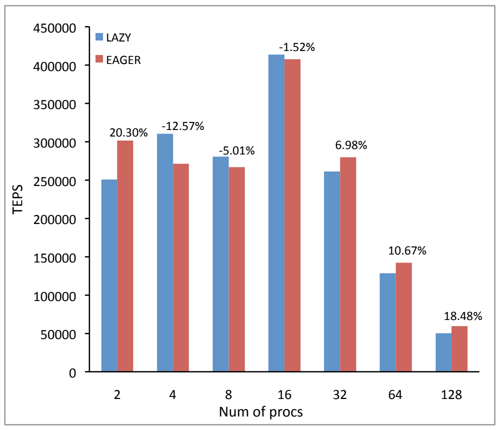
Figure 4.6: Overlapping Results(fence) for (a) PUT and (b) GET

structs a graph and performs parallel Breadth First Search (BFS) on that graph. One implementation of BFS in Graph500 is done by using `MPI_accumulate` operations and fence synchronization.

We tested the Graph500 benchmark with two different size of the input graph: (i) 2^{13} vertices and 2^{17} edges and (ii) 2^{7} vertices and 2^{11} edges. The first one has larger amount of one-sided operations between fences whereas the second one has smaller number of operations. We increased the number of processes up to 128 and plot the corresponding TEPS (traversed edges per second) in Fig 4.7.

In Figure 4.7(a), we can see that, when process number is smaller than 16, LAZY performs better than EAGER in most cases(except for 2-process case). This is because number of local outgoing operations on each process is not large enough, for EAGER case, the overhead of two barriers still dominates the communication latency. When number of processes is increased to 32 or larger, number of outgoing operations on each process increases too, which makes the benefit of issuing operations early outperforms the cost of barriers, therefore EAGER performs better than LAZY. Since LAZY defers issuing operations to synchronzation phase, it has to spend longer time waiting for the completion of these operations.

In Figure 4.7(b), we can see that, when problem size is not large, LAZY



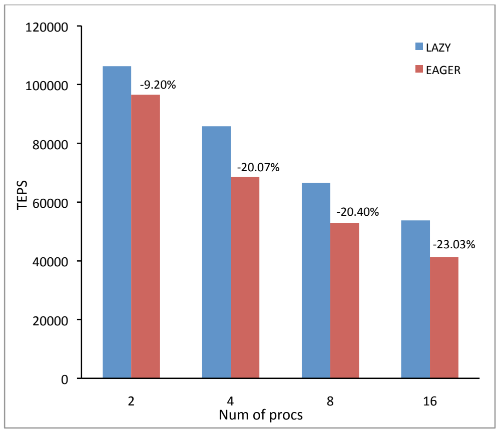
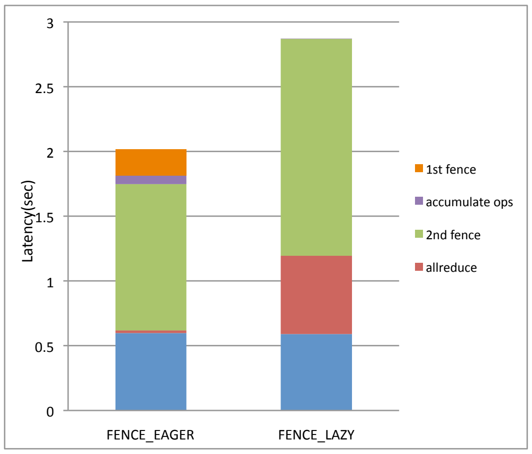


Figure 4.7: Graph500 results with (a) large graph and (b) small graph

always performs better than EAGER. This is because there is no enough one-sided operations between fences and the advantage of issuing them early is not as much as the synchronization overhead and the overhead of waiting for arrival of additional last message, so EAGER performs worse than LAZY.

To further study the difference of EAGER and LAZY in fence, we break down the timing for the second fence and measure the time taken by each part inside the fence function. Figure 4.8 shows the breakdown for eager and lazy fence for 128 processes with large problem size.

There is no barrier operation in LAZY approach and no reduce-scatter operation in EAGER approach. EAGER has the cost of sending the additional last message, which does not exist in LAZY. However, EAGER completely eliminates the time of issuing operations as all operations have been issued before fence, whereas in LAZY all the outgoing operations are queued and issued in this fence. Both approaches spend most of their time waiting for operation completion. LAZY spends more time than EAGER because it issues operations late. For EAGER, there is also overhead of barrier taken place in the first fence, whereas for LAZY mode, the time of the first fence is nearly zero.



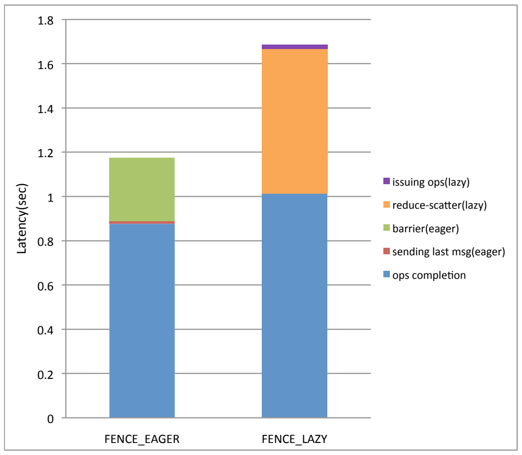


Figure 4.8: Time breakdown for (a) overall time and (b) second fence time

Chapter 5

Conclusion and Future Work

This thesis describes the design and implementation of an adaptive strategy for one-sided communication in MPI. The current MPICH2 implementation uses a lazy approach to issue operations, which queues the operations and issues them at the second synchronization phase. This has certain benefits with respect to short operations in terms of reduced network operations. For large data transfers, issuing the operations as early as possible is more beneficial than deferring them to later synchronization phase, and it can provide more scope for overlapping communication and computation. Our design and implementation of adaptive approach combine the benefits of both lazy and eager approaches with small overhead. The performance results show that the hybrid approach performs as good as the lazy approach for small data transfers and still be able to achieve performance similar to eager for large data transfers. We also demonstrate the benefits of fence implementation for the Graph 500 benchmark.

In our implementation, the hybrid approach for lock-unlock queues up the operations until the lock is granted, and then issues all the operations immediately. Once the lock is granted there is no further operations being queued. We achieve this by internally setting the queue threshold parameter to 1. In cases where there are lots of operations and contention in the network, the hybrid approach could be tweaked to queue up operations to a certain number of operations or a certain data volume. Currently, in our testing of fence and post-start-complete-wait, we set the threshold based on the performance result of lazy and eager approach. The queue threshold should be chosen appropriately for a given system, and should avoid user to learn about the communication pattern for a specific application. We are trying to propose a systematic way for user to select the suitable queuing threshold based on system parameters. Furthermore, by keeping track of the history of the operations and communication pattern, the threshold can also be dynamically adjusted in the runtime. We plan to explore these possibilities in the future. We are also working on testing our approach on more commonly used one-sided benchmarks, especially for passive target communication, to examine the effectiveness of our design and implementation.

References

[1] Message Passing Interface Forum, "MPI-2: A Message Passing Interface Standard," *High Performance Computing Applications*, 1988, Vol. 12, 1–299.

[2] Argonne National Laboratory, "MPICH2," *http://www-unix.mcs.anl.gov/mpi/mpich2/.*

[3] B. W. Barrett, G. M. Shipman and A. Lumsdaine, "Analysis of Implementation Options for MPI-2 One-Sided," *Proceedings of Euro PVM/MPI 2007.*

[4] J. Traff and H. Ritzdorf and R. Hempel, "The Implementation of MPI-2 One-Sided Communication for the NEC SX," *Proceedings of Supercomputing 2000.*

[5] Cray Research Inc, "Cray T3E C and C++ optimization guide," 1994.

[6] J. Nieplocha and B. Carpenter, "ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems," *Lecture Notes in Computer Science*, 1999, Vol. 1586.

[7] D. Bonachea, "GASNet Specification, v1.1," *Technical Report UCB/CSD-02-1207, Computer Science Division, University of California at Berkeley*, October, 2002.

[8] M. Goudreau, K. Lang, S. B. Rao, T. Suel and T. Tsantilas, "Portable and Efficient Parallel Computing Using the BSP Model" *IEEE Transactions on Computers*, 1999, 670-689.

[9] Jonathan M. D. Hill, "Lessons learned from Implementing BSP" *Oxford University Technical Laboratory, Technical report 21-96.*

[10] Jonathan M. D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas and Rob H. Bisseling “BSPlib: The BSP programming library” *Parallel Computing 24(14):1947-1980*, 1998.

[11] William D. Gropp and Rajeev Thakur “Revealing the Performance of MPI RMA Implementations” *EUROPVM/MPI’07*, 272-280, 2007.

[12] William D. Gropp and Rajeev Thakur “An evaluation of implementation options for MPI one-sided communication” *EUROPVM/MPI’05*, 415-424, 2005.

[13] Rajeev Thakur, W. Gropp and B. Toonen “Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication” *EUROPVM/MPI’04*, 2004.

[14] J. Liu, W. Jiang, Hyun-Wook Jin, D. K. Panda, W. Gropp and Rajeev Thakur “High Performance MPI-2 One-Sided Communication over InfiniBand” *International Symposium on Cluster Computing and the Grid (CCGrid 04)*, April, 2004.

[15] W. Jiang, J. Liu, H. W. Jin, D. K. Panda, D. Buntinas, R. Thakur and W. Gropp “Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters” *EuroPVM/MPI 2004*, 2004.

[16] Gopal Santhanaraman, Sundeep Narravula and Dhabaleswar K Panda “Designing passive synchronization for MPI-2 one-sided communication to maximize overlap” *IPDPS 2008*, April, 2008.

[17] G. Santhanaraman and P. Balaji and K. Gopalakrishnan and R. Thakur W. Gropp and D. K. Panda "Natively Supporting True One-sided Communication in MPI on Multi-core Systems with InfiniBand" *CCGRID 2009*, May, 2009.

[18] Jesper Larsson Traff, William D. Gropp and Rajeev Thakur "Self-Consistent MPI Performance Guidelines" *IEEE Trans. Parallel Distrib. Syst.*, 698-709, 2010.