Communication Patterns and MPI Primitives

1 Models for Analyzing Communication Cost

There are more than one model for analyzing communication costs over a network intraconnect — the Hockney model [3], LogP model [2], LogGP model [1] and PLogP model [4]. All these models are described indifferent to the topological details of the underlying network intraconnect.

Hockney model: The Hockney model [3] assumes that the time to send a message of size m bytes from one node to another over the network intraconnect is $\tau + \mu \times m$, where τ is a fixed latency (aka. setup cost some times), and μ is the reciprocal of the network bandwidth[1]. Network congestion is *not* captured by this model.

LogP and related models: The LogP model and their derivatives differ slightly but they all assume a latency (L) to set up internode transfer, overhead (o) to read and write from/to the network buffer, and a minimum gap in time (g) between two consecutive sends from a sender[2]. More specifically, the LogP model assumes a message sent between two nodes is of a *fixed* size. Under this assumption, the transfer time is $L + 2 \times o$, where L is the latency and o is a measure of the additional overhead to account for the time required to place the message on the network buffer by the sender and the time to read the message from the network card by the receiver. The LogGP model extends the LogP model by allowing for variable sized messages. The time to communicate a message of size m between two nodes is equal to $L + 2o + (m - 1)G$, where G is the minimum enforced time gap between two successive sends. The PLogP model is a further extension to the above models in which the sender and receiver overheads and the gap per message are all functions of the message size.

For a review of these competing models and their relative performance, please refer to [6]. While there are more such models, the most commonly used model by the message passing community is the Hockney model. For the same reason, we will use the Hockney model for analyzing communication complexity.

In what follows, we describe some of the basic point to point and collective communication primitives supported by the Message Passing Interface (MPI). Although there are multiple open source implementations of MPI (MPICH, OpenMPI, etc.), the discussion in the text does *not per se* cater to any particular implementation. Instead we will focus on at least one efficient way to structure the algorithm for these primitives and analyze their respective communication complexities. In practice, MPI implementations are configured to take advantage of the underlying network topologies and interfaces.

[1] Network bandwidth is the speed of the network and is given by the number of transferred bytes per sec.

[2] The "L" in LogP stands for latency, o for overlap and g for gap

2 MPI Point to Point Communication Primitives

2.1 Blocking communication

MPI functions: *MPI_Send, MPI_Recv, MPI_Ssend*

A *blocking* communication is one that forces the process that issues the call to wait until the communication is "complete". The notion of "completeness" here is different between the sender and receiver versions.

Any call to *MPI_Recv* waits until the entire message is received by the receiver node and is copied in its entirety into the local application buffer. If a receiver posts its receive call prior to the corresponding send by the sender, then measuring the time for the *MPI_Recv* call to return effectively measures the entire time for that communication.

Contrary to the receive function, MPI provides two different blocking versions for sending — viz. *MPI_Send* and *MPI_Ssend*. According to the MPI standard, *MPI_Send* is only required to wait until the message has left the local application (send) buffer (i.e., the latter can be safely modified by the application). In other words, this *could* potentially mean that an MPI_Send call returns when the message transfer is still in progress, or even possibly that a corresponding receive has *not* yet been posted by the receiver. For small message sizes this is probably safe, as the local network buffer is typically well equipped to temporarily cache the message before it is dispatched to the received. However, for large message sizes, this poses the risk of packet loss.

As a safer alternative, MPI offers the *MPI_Ssend* (or the synchronous send) function. A call to *MPI_Ssend* is guaranteed to return only after the receiver has posted a receive call. Even if the message size is small enough to be buffered at the sender's local network buffer, the call waits until a matching receive call is posted by the receiver. Evidently, *MPI_Ssend* calls require a handshake and are hence more expensive and also safer than *MPI_Send*. The additional cost is however warranted in applications where message sizes could be arbitrarily large and/or multiple senders could send to the same receiver concurrently.

2.2 Nonblocking communication

MPI functions: *MPI_Irecv, MPI_Test, MPI_Wait, MPI_Isend*

The time spent waiting for a communication to complete contributes directly to parallel overhead. A popular technique that is used to reduce this overhead is to *overlap communication with computation* — the idea is to perform a nonblocking call to initiate the communication and instead of waiting for the communication to complete, perform local computations that are not dependent on the communication to finish. This often requires the algorithm designer to carefully prefetch data prior to the phase in computation when they are needed, so that the computation for which data is already locally available can be allowed to continue and used to effectively mask the communication. A key to allow this overlap is the support for nonblocking calls.

MPI supports the nonblocking *MPI_Irecv* call. This call returns immediately after posting the receive so that the receiver process can perform local computation based on available/prefetched data. Subsequently, to test if the message has been received, the receiver uses either the nonblocking *MPI_Test* function or the blocking *MPI_Wait* function. A typical usage to the nonblocking receive follows one of the following two patterns:

S1) Post *MPI_Irecv*;
S2) Perform partial computation on the next batch of available data; If no more computation is possible with available data then go to step S4;
S3) Perform *MPI_Test*. If the receive status is complete, then loop back to step S1; or else, perform loop back to step S2;
S4) Perform *MPI_Wait*; when the call returns loop back to step S1.

Alternatively,

S1) Post *MPI_Irecv*;
S2) Perform computation on all available data;
S3) Perform *MPI_Wait*; when the call returns loop back to step S1.

For symmetry, MPI also supports a nonblocking send *MPI_Isend* although it is typically identical to *MPI_Send*.

MPI also supports *one-sided* communication calls such as *MPI_Get* and *MPI_Put*. As the name implies, these calls are expected to involve only one of the two processes — the sender for *MPI_Put* or the receiver for *MPI_Get* — but not both. MPI implementations vary on the level of support for these one-sided calls. For more details about these calls the reader is referred to [7].

3 Permutations

In MPI applications, multiple pairwise interprocess communications can be arranged systematically such that all processes participate within each step of communication. Such communication patterns are referred to as *permutations*. Some of the common permutations are as follows. Recall that p denotes the number of processes.

Shift permutations: Interprocess communications are arranged such that rank i sends a message to rank $i-1$ and receives from rank $i+1$ (with the exceptions of rank 0 which only receives from rank 1, and rank $p-1$ which only sends to $p-2$. This would be a *left* shift permutation. A mirror variant is the *right* shift permutation. Other variants are possible — e.g., an interleaved shift permutation where the processes communicating are separated by a fixed number of ranks apart. In shift permutations, processes are logical ordered as a bus. If indeed the physical ordering also is consistent with this logical ordering, then the performance expectations will be met in practice. Otherwise, there will be additional delays imposed by the physical layer.

Care must be taken while implementing shift permutations. A poor implementation could destroy parallelism in communcation. Can you identify such as a case?

Ring permutations: Ring permutations are identical to shift permutations with the exception that ranks 0 and $p-1$ are allowed to communicate with one another, so as to form a logical ring of processes. Ring permutations map well to network intraconnects that have an embedded ring topology.

Care must be taken while implementing ring permutations. A poor implementation could lead to deadlocks. Can you identify such as a case?

Even though the shift and ring permutations are simple, they are adequate for many real world use-cases especially in applications relating to box simulations where the next state of a cell is affected by the states of the cells in its neighborhood. The Conway's Game of Life is an apt example.

Hypercubic permutation: A hypercubic permutation is one in which all pairwise interprocess communications happen between ranks i and j, such that the bit representation of j is given by toggling the k^{th} least significant bit (alternatively, most significant bit) in the bit representation of i, for a fixed $k \in [0, \lg p - 1]$. The term hypercube comes from the network topology of the same name where each physical node contains $\lg p$ links (or neighbors). If a hypercubic permutation is carried out on a hypercubic network there will be *no* contention.

4 MPI Collective Communication Primitives

A *collective communication* is one in which *all* the processes in a communicating world participate. Examples of collective operations are: broadcast, reducing a sum, gathering data from all processes, scattering/distributing data among processes, etc. For a collective operation to happen, all processes in a communicating group (e.g., *MPI_COMM_WORLD*) should issue a call to the corresponding communication primitive passing the same values to the communicator argument and the "root" argument (if any). Note that this also implies that a collective operation can be performed for any smaller communicating groups of processes that are subsets of the *MPI_COMM_WORLD*. A "root" is a special process identified by some (but not all) collective communicators as either the originator or the final receiver for the collective operation (e.g., a broadcast root, or a reducer's root where the reduced value will be stored).

Caveat about collective operations and MPI implementations: While it is typical to expect that all participating processes in a collective operation at least post a call before the operation starts, this is not enforced for most of the collective primitives by the latest standard for MPI, MPI-3.0 [5]. The standard also does *not* enforce that the calls should return only after the communication is completed in all participating processes. In other words, for many of the collective primitives, it is possible that a caller may return soon after its contribution to the collective operation is completed while some other processes may be still working on the communication.

4.1 Barrier

MPI functions: *MPI_Barrier*

The *MPI_Barrier* function is used to ensure that *all* processes have reached a certain stage in computation before proceeding. The barrier function is invoked on a specific communicating group (e.g., *MPI_COMM_WORLD*) and each process within that group blocks on the call until *all* other processes of that group have invoked the barrier function as well. At that point the call returns.

Note that there is no application level data that is exchanged during barrier. In contrast, all the remaining collective primitives described below involve exchanging application data transfers. For purpose of uniformity, we will use the notation m to denote the size of the message (in bytes) at any process prior to the collective operation. We will also use the notation n to denote the sum of the number of elements (of any particular data type) in all the processes, and p to denote the number of processes.

4.2 Reduce

MPI functions: *MPI_Reduce, MPI_Allreduce*

Precondition: Each process has $O(\frac{n}{p})$ elements as input. Let the input in rank i be represented as an array $A_i[1 \ldots \frac{n}{p}]$. Also given is a binary associative operator denoted as $\otimes$ (e.g., addition, logical or/and, max, min, etc.).

Postcondition: Output array $B[1 \ldots \frac{n}{p}]$ s.t., $B[k] = A_0[k] \otimes A_2[k] \otimes \ldots A_{p-1}[k]$. If the function is *MPI_Reduce*, then the array B is made available in the process identified as the root. If the function is *MPI_Allreduce*, then array B is made available in all the processes.

Assuming that each application of the binary operator $\otimes$ takes constant time, reduction can be achieved in $O(\lg p)$ stages of hypercubic permutations. Within each permutation, $\frac{n}{p}$ elements are exchanged between any two processes to compute the intermediate B arrays. The time complexity for this communication is therefore $O((\tau + \mu\frac{n}{p}) \times \lg p) = O((\tau + \mu m) \times \lg p)$.

4.3 Broadcast

MPI functions: *MPI_Bcast*

Precondition: The root process has a message of m bytes.

Postcondition: The message at the root process is available at all the other processes.

It should be easy to see that the broadcast can be implemented by reversing the communication stages of the reduce operation, implying that it can also be achieved in $O(\lg p)$ stages of hypercubic permutations. The time complexity for broadcast is therefore $O((\tau + \mu m) \times \lg p)$.

4.4 Gather

MPI functions: *MPI_Gather, MPI_Allgather*

Precondition: Each process has $O(\frac{n}{p})$ elements. Let the input in rank i be represented as an array $A_i[1 \ldots \frac{n}{p}]$.

Postcondition: Output array $B[1 \ldots n]$ which is given by concatenating all the individual rank arrays A_i in that order of ranks. If the function is *MPI_Gather*, then the array B is made available in the process identified as the root. If the function is *MPI_Allgather*, then array B is made available in all the processes.

The gather operation can also be completed in $O(\lg p)$ hypercubic permutations as follows. Let $O(m_k)$ be the message size at each process at the beginning of the k^{th} hypercubic permutation stage. During the k^{th} stage, all communicating pairs of processes perform a gather of their respective $O(m_k)$ messages such that at the end of the stage, both processes of a pair have the same message of size $O(2 \times m_k)$. Consequently, the last stage results in the gather of $O(m \times p)$ size message at the root process (alternatively, at all processes for *MPI_Allgather*). The communication time complexity is bounded by $O(\tau \times \lg p + \mu \times m \times p)$.

4.5 Scatter

MPI functions: *MPI_Scatter*

Precondition: The root process has $O(n)$ elements. Let this input be $A[1 \ldots n]$ (of size m).

Postcondition: Scatter array $A[1 \ldots n]$ to all processes such that process rank i contains elements $A[i \times \frac{n}{p} \ldots (i+1) \times \frac{n}{p} - 1]$.

The scatter operation can also be completed by reversing $O(\lg p)$ hypercubic permutations of the gather operation. Consequently, the communication time complexity is $O(\tau \times \lg p + \mu \times m))$.

4.6 Parallel Prefix

MPI functions: *MPI_Scan*

(p element case)

Precondition: An array of A of p elements is kept distributed such that rank i holds element $A[i]$. We will refer to the value of $A[i]$ as x_i. We are also specified a binary associative operator which we will denote by $\otimes$.

Postcondition: An output array B of size p available in a distributed manner such that rank i holds element $B[i]$. We will refer to the value of $B[i]$ as y_i. The value of y_i is given by:

$$y_i \ = \ x_0 \otimes x_1 \otimes \ldots x_{i-1} \otimes x_i$$

The algorithm for parallel prefix can be accomplished in $O(\lg p)$ communication steps using hypercubic permutations, similar to the reduce and broadcast primitives. For details on the parallel prefix algorithm and its applications, please refer to the lecture notes by Prof. Aluru.

(n element case)

An easy extension of the $O(\lg p)$ approach lends itself to solving the more generic case of parallel prefix where the input array A is of size n, where $n >> p$ (i.e., each processor initially holds $O(\frac{n}{p})$ entries). Here the complexity will be:
Computation complexity: $O(\frac{n}{p})$
Communication complexity: $O((\tau + \mu) \lg p)$.

4.7 Transportation Primitives

MPI functions: *MPI_Alltoall, MPI_Alltoallv*

MPI_Alltoall:

Precondition: Each processor p_i has a message size of m to send to every processor (including to itself). We will denote the set of p messages in rank i has $m_{i,0}$, $m_{i,1}$, ..., $m_{i,p-1}$.

Postcondition: Processor i holds all its messages from every processor in order — i.e., at output, rank i holds an array which is given by $m_{0,i} \cdot m_{1,i} \cdot \ldots \cdot m_{p-1,i}$.

There are a number of ways to implement the Alltoall primitive on different parallel architectures. A simple lowerbound to these implementations, however, is imposed by the maximum between the sizes of the messages that needs to sent and received at any processor. If this value is bound by $O(m)$ for a communication that happens between any two processor, then the overall complexity of the Alltoall communication is $\Omega(m \times p)$. Therefore, a simple approach to implement Alltoall is to have every processor send to every other processor in a round-robin manner (or any other deterministic way). A simple pseudocode at rank i could be as follows:

S1) For $j \leftarrow 1$ to p do:
S2) destination $k \leftarrow (i + j) \mod p$
S3) Send $m_{i,k}$ to k.

The communication complexity for this algorithm is $O(\tau p + \mu m p)$.

Note that the Alltoall operation is same as performing the equivalent of Allscatter from every processor. It is for this reason that MPI does *not* explicitly support Allscatter.

MPI_Alltoallv:

The MPI_Alltoallv is a generalization of the MPI_Alltoall primitive in that the Alltoallv version allows for variable message sizes to be sent from each processor to any other processor. The assumption here is that on any processor's local memory the values that need to be sent to the same destination processor are made available contiguously. Prior to calling the Alltoallv primitive, each rank has to specify the locations (i.e., pointers) on its local memory to each of the p send buffers and the corresponding send counts (i.e., how many values of what type going to each destination processor). Using this model, Alltoallv can be used to send arbitrary sized messages from any processor to any other processor.

In many applications that need variable size messages to be sent between processors, it is possible to guarantee that the sum of the sizes of the outgoing messages and the sum of the sizes of the incoming messages at any processor satisfies a bound $O(m)$ — i.e., at rank i, $\Sigma_{j=1}^{p} m_{i,j} = O(m)$. As an example, consider the sorting operation. The following example shows a three processor system each holding 6 elements marked for different destinations as shown below:

p_0: 0 0 1 1 1 2 | p_1: 0 1 1 2 2 2 | p_2: 0 0 0 1 2 2

Here, rank 0 has different number of values to send to each of the p processors — i.e., 2 to p_0, 3 to p_1 and 1 to p_2. However the sum of the outgoing messages and the sum of the incoming messages at any processor is bounded by 6. For these scenarios, where there is variability in the size distribution of messages but the net inbound/outbound message volume can be bounded in every processor, one can implement Alltoallv using two Alltoall communication steps so that the entire communication cost can be bounded by the cost of Alltoall — i.e., $O(\tau p + \mu m)$.

The algorithm to implement Alltoallv using two calls to Alltoall is as follows:

S1) Each processor i splits *each* of its p messages into p equal parts (generating a total of p^2 parts per processor).
S2) Rearrange the p^2 parts such that part k is marked for destination rank $(k \mod p)$.
S3) Perform an Alltoall using the send buffer generated in the above step. Note that it becomes possible to use Alltoall here because each processor is guaranteed to have the same message size ($\frac{m}{p}$) outbound to every other processor, as a result of the division in step S1.

S4) Upon Alltoall completion, every processor rearranges the p^2 parts that it received from all p processors such that part k is marked for destination rank $(k \mod p)$.
S5) Perform another Alltoall using the send buffer created by the above step.
S6) Upon Alltoall completion, each processor should have only the parts that are destinted for that processor. However, the parts will be shuffled in a cyclic permutation. Therefore, a shuffle operation is performed locally to gather parts that originated at the same processor consecutively.

For an illustrative example of the process, please refer to Figure 1.

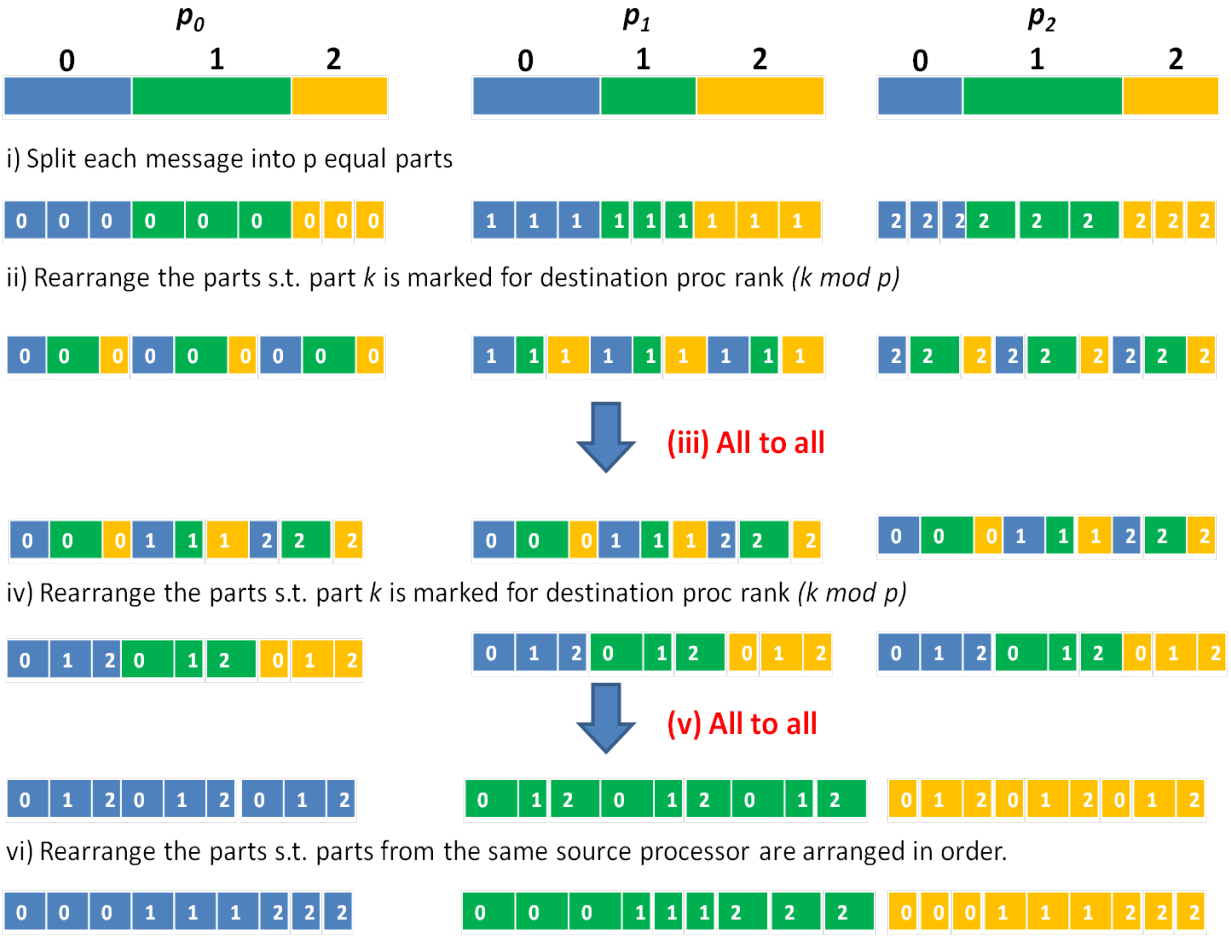

Figure 1: An example showing the Alltoallv transportation primitive algorithm using two Alltoallv communication steps.

5 Error Handling

MPI functions: *MPI_Abort*

MPI supports an abort function which can be invoked from any processor rank in the event an exception needs to be raised. A call to MPI_Abort will result in aborting the processes at all ranks.

MPI_Abort is typically useful for generating exceptions and also during debugging.

[1] Albert Alexandrov, Mihai F Ionescu, Klaus E Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP modelone step closer towards a realistic model for parallel computation. pages 95–105. ACM, 1995.

[2] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[3] Roger W Hockney. The communication challenge for MPP: Intel paragon and meiko CS-2. *Parallel computing*, 20(3):389–398, 1994.

[4] Thilo Kielmann, Henri E Bal, and Kees Verstoep. Fast measurement of LogP parameters for message passing platforms. In *Parallel and Distributed Processing*, pages 1176–1183. Springer, 2000.

[5] MPIForum. The MPI 3.0 standard: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[6] Jelena Pjeivac-Grbovi, Thara Angskun, George Bosilca, Graham E Fagg, Edgar Gabriel, and Jack J Dongarra. Performance analysis of MPI collective operations. *Cluster Computing*, 10(2):127–143, 2007.

[7] Rajeev Thakur, William D Gropp, and Brian Toonen. Minimizing synchronization overhead in the implementation of MPI one-sided communication. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 57–67. Springer, 2004.