Lecture notes: Collective operations in message-passing systems

Master M1: Parallel Algorithms and Programming

2022

This lecture introduces collective operations in distributed memory systems. It also presents some major algorithms and techniques to achieve good performance for such operations.

1 Introduction

1.1 Positioning

In the previous lectures, we have studied parallel programming in shared memory systems, that is, systems where all processing elements have access to a shared address space. In the following, we will start studying the alternative distributed memory model. The distributed memory model is often adopted when working with a system composed of multiple nodes connected through a network. We refer you to the Lecture 3 (*Parallel architectures and programming models*) for a more detailed comparison of the models.

1.2 Context

We consider a distributed memory system. Each processor (also called *node* in this context) has access to a local private memory and communicates with the other nodes by exchanging messages over a network.

Some communication and computational patterns are very common in parallel applications executing on a distributed memory system. To achieve good performance, it is important to study and optimize the performances for these patterns.

Some of these patterns involve groups of processors. In the parallel community, we name these patterns *collective operations*. Message passing libraries for parallel programming, such as MPI (The Message Passing Interface), provide implementations for the most common collective operations. Hence, you will not have ti implement then yourself. Still, it is important to study them: i) to understand their semantic; ii) to analyze and understand some algorithms that allow achieving performance at scale.

A significant difference between shared-memory and distributed-memory systems is the size. Whereas a multi-core processor will in general include at best a few tens of processing units, distributed memory systems might include hundreds, thousands, or more processing units. As a consequence, optimizing the performance of the synchronization/communication patterns between the processing units is more central to the performance of these systems.

2 Execution model

To design communication operations and analyze their performance, we first need to define an execution model. The execution model defines the assumptions that are made about the underlying hardware and software layers. The model should be at the same time simple (to allow reasoning about it) and accurate enough to capture the main characteristics of the execution platform (and so, to ensure that the designed algorithm will work well in practice).

Here is the model that we will consider in this course:

- Transferring a message `M` of size m over one link takes a time $t = L + m/B$.
  - L is the latency (or delay): the time to transfer one word from the source to the destination. L is independent of the message size.
  - B is the bandwidth: the rate at which words can be inserted on the network link.
- A *store-and-forward* model is assumed for intermediate nodes that transmit a message from a source to a destination:
  - A node receives the full message `M` before transmitting it to the next node.
- We assume *full-duplex* communication channels: data can transit in both directions at the same time on a network link.
- Finally, we assume that a processor can send data, receive data, and perform computation on local data at the same time.

This model is very simple but, still, it is able to capture major performance trends.

3 Collective operations

In this section, we introduce the main collective operations used in parallel systems. For the description, we assume a set of n processors $\{P_1, P_2, \dots P_n\}$.

Broadcast

- *One-to-all* communication pattern
- One processor P_k sends a message to all processors.

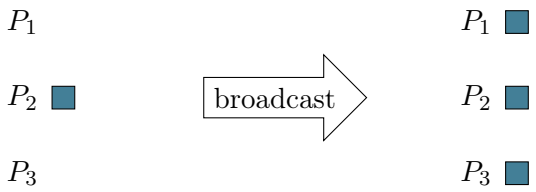

Figure 1: Broadcast

Scatter

- *One-to-all* communication pattern
- One processor P_k has a vector of n data blocks.
- Each processor receives one of the blocks.
- All blocks have the same size and type.

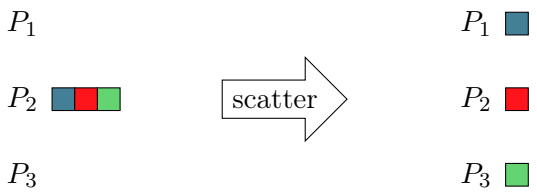

Figure 2: Scatter

Gather

- *All-to-one* communication pattern.
- Each processor has a block of data.
- A vector of blocks is constructed on one processor P_k. Block i in the vector in the block from process P_i.
- All blocks have the same size and type.

Figure 3: Gather

Reduce

- *All-to-one* communication pattern (also called *accumulation*)
- Same as gather but a reduction operation ($\bigotimes$) is applied to *accumulate* the data blocks.

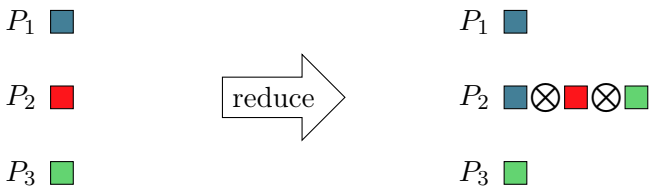

Figure 4: Reduce

All-gather

- *All-to-all* communication pattern
- Same as *Gather* excepted that all processors receive the constructed vector.

Figure 5: All-gather

All-reduce

- *All-to-all* communication pattern
- Same as *Reduce* excepted that all processors receive the computed block.

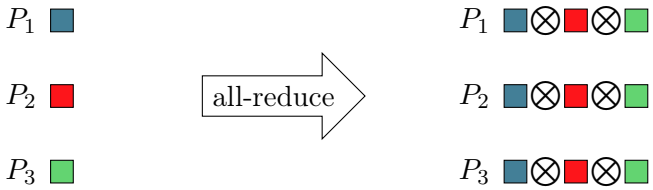

Figure 6: All-reduce

All-to-all

- *All-to-all* communication pattern
- Initially each processor has a vector of n blocks where block i should be sent to processor P_i

- After the operation, each processor has a vector of n blocks where block i was received from processor P_i

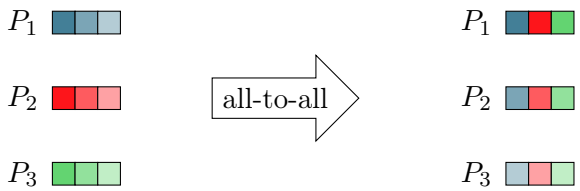

Figure 7: All-to-all

4 Implementation of collective operations

To design algorithms for collective operations, one first needs to decide about the underlying network topology that is assumed. The network topology defines the distance in number of hops between any two processors in the distributed system.

- A 1-hop distance corresponds to the case where there is a direct link between two processors.
- A n-hop distance corresponds to the case where there are $n-1$ processors on the path from the source to the destination.

One usually makes the distinction between the virtual and the physical network topology. The physical topology corresponds to the way the physical network is built on the target platform. A logical topology is an arbitrary topology that is used to reason about algorithms. An algorithm designed for a given logical topology will, with a high probability, work better on physical topologies that closely match the considered logical one.

In this section, we consider a logical topology that corresponds to a fully connected network: each node has a direct connection with every other nodes in the system. Such a topology allows reasoning about the theoretically most efficient algorithms.

The efficiency of collective operations can be evaluated according to two metrics: the latency and the throughput:

- The latency is the time from when the collective operation is initiated until when it is completed on all processes.
- The throughput (or effective bandwidth) is the amount of data that can be *processed* by the collective operation per time unit.

In the following, we are going to study 2 operations: broadcast and all-gather. By studying these two operations, we can illustrate some major strategies for scalable collective operations.

4.1 Broadcast

A simple (naive) algorithm for the broadcast algorithm is the following one: The source node sends the message to all destinations.

Let's compute the time taken for this algorithm to transmit a message of size m, assuming a system with n nodes:

$$T_{bcast}(m) = (n-1) \times (L + \frac{m}{B})$$

As we can see, this algorithm is good neither from latency nor from throughput point of view. Indeed, both the latency and the throughput terms are proportionals to the number of nodes.

Another algorithm based on a binomial tree is optimal from latency point of view for power-of-two number of processes. The binomial tree algorithm is described by the figure bellow. The figure assumes that processor P_1 is the source.

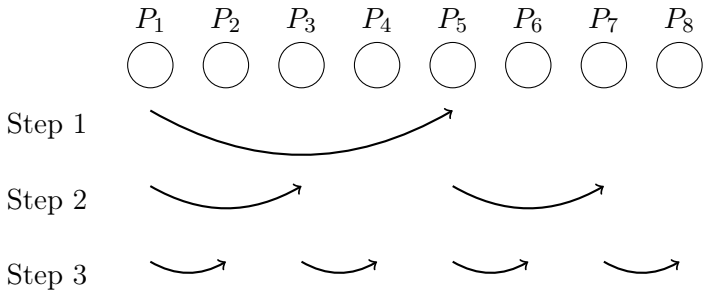

Figure 8: Binomial tree (broadcast from node P_1)

The algorithm is designed so that the number of *senders* is multiplied by 2 at every step of the algorithm. We can compute the performance of this new algorithm.

$$T_{bcast}(m) = log(n) \times (L + \frac{m}{B})$$

This new algorithm scales much better as its performance is proportional to the logarithm of the number of nodes involved in the communication. This algorithm is actually optimal from latency point of view. From throughput point of view, a better performance can be achieve by an algorithm proposed by Van de Geijn et al. This algorithm starts by running a scatter operation followed by a all-gather operation. A detailed discussion about this algorithm goes beyond the scope of this lecture.

4.2 All-gather

The *all-gather* operation is a collective operation that constructs an array out of the contributed block of each node, and where the result is shared between all nodes.

We study the *all-gather* operation because it allows applying a communication pattern called *recursive doubling*. Recursive doubling (or the complementary *recursive halving* strategy) can be used for the efficient implementation of several collective operations.

The basic idea of the recursive doubling algorithm to implement the *all-gather* operation is the following: i) at each step each processor exchange with another processor; ii) at each step the distance of the processor to exchange with is multiplied by 2.

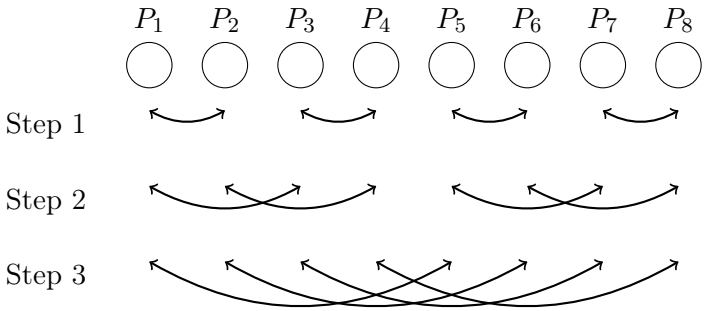

Figure 9: Recursive doubling (to implement all-gather operation)

We can check with an example that the algorithm illustrated in Figure 9 works in practice. Let us consider the processor P_6:

- At step 1, P_6 receives the block of P_5.
- At step 2, P_6 receives from P_8 the blocks of P_7 and P_8.
- At step 3, P_6 receives from P_2 the blocks of P_1, P_2, P_3 and P_4.

Hence, after 3 execution steps, the processor has received all the blocks. Again, we obtain an algorithm that scales well as it requires $log(n)$ steps to complete. Computing the exact performance of the algorithm goes beyond the scope of this lecture.

5 Collective operations on a unidirectional ring

In this section, we study a more constrained topology, namely a unidirectional ring of n processors. As described in Figure 10, the unidirectional ring implies that a processor can only receive messages from the preceding processor and send messages to the following processor in the ring.

With this topology, a broadcast operation with processor P_k as source node can be implemented with the algorithm of Figure 11.

Note: *We are going to study broadcast algorithms in a virtual ring in more details during the tutorial session. Among other things, during the tutorial session, we will study how to specify the algorithm assuming that 4 primitives are available to each processor:* `MY_NUM()`, `NUM_PROCS()`, `SEND(m, id)`, *and* `RECV(m, id)`.

Use of the virtual ring topology: Although algorithms built based on a virtual ring topology suffer from a high latency, there are use-cases where they can be very efficient in practice. This is especially the case when the number of processors involved in the communication is not a power of two and when the message is large. Indeed, for large messages, the throughput is a more important

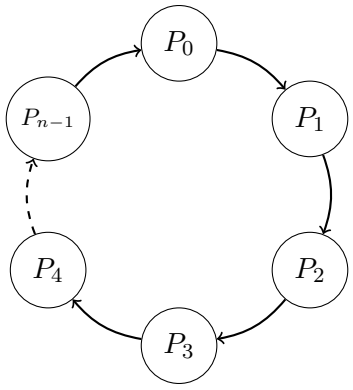

Figure 10: Unidirectional ring

```
1    /* broadcast of msg M, source is processor P_k */
2    broadcast_ring(M, k){
3      if(my_id != k){
4        Receive M from processor (my_id-1) mod n
5      }
6
7      if (my_id != k-1 mod n){
8        Send M to processor (my_id+1) mod n
9      }
10   }
```

Figure 11: Broadcast in a unidirectional ring

metric than the latency. Still, note that ring-based algorithms do not scale well with very large numbers of processors.

A domain where ring-based algorithms are used today is distributed deep learning. Distributed deep learning algorithms heavily rely on the *all-reduce* collective operation, and run this operation with large messages. Since distributed deep learning solutions use in general at most a few tens of processors, ring-based algorithms can work efficiently.

Pipelining One can observe that in the ring-based broadcast algorithm presented above, there is no parallelism in the data transfers. To increase the level of parallelism, one can split a large message into small messages (called *packets* hereafter) and apply pipelining. Applying pipelining works in the following way. Here, we assume that the source is processor P_0:

- Step 1: The source processor P_0 sends the first packet to processor P_1;
- Step 2: P_0 sends the second packet to P_1 while P_1 sends the first packet to P_2
- Step 3: P_0 sends the third packet to P_1 while P_1 sends the second packet to P_2 and P_2 sends the first packet to P_3.
- etc.

Based on this description, it is obvious that the level of parallelism has increased. We can use our performance model to better assess the improvement. For the ring-based broadcast algorithm, the performance is the following:

$$T_{bcast-ring}(m) = (n-1) \times (L + \frac{m}{B})$$

For the pipelined version of the ring-based algorithm, assuming that the message is split into r packets, the performance is:

$$T_{bcast-ring-pipelined}(m) = (n+r-2) \times (L + \frac{m}{r} \times \frac{1}{B})$$

This results is derived from two observations:

1. The time for the first packet to reach the last node in the ring is $(n-1) \times (L + \frac{m}{r} \times \frac{1}{B})$
2. After the first packet has arrived, we are still waiting for $r-1$ packets.

Pipelining will be discussed in more details during the tutorial session.

References

Some references can complement the material presented in these lecture notes:

- Section 3.5.2 of the book "Parallel Programming: For Multicore and Cluster Systems" (by Rauber and Runger) about collective operations definition.
- Section 3.3 of the book "Parallel Algorithms" (by Casanova, Robert, and Legrand) about algorithms in a ring.
- The article "Optimization of Collective Communication Operations in MPICH" (by Thakur, Rabenseifner, and Gropp) that describes and evaluates several algorithms for collective operations.