CMPSCI 691W Parallel and Concurrent Programming **Spring 2006**

Lecture 13: April 17

Lecturer: Emery Berger *Scribe: Ting Yang*

13.1 Overview

This lecture describes the collective communication in MPI. Besides the point-to-point communication, MPI also provides a rich set of collective operations to support the need of scientific computation.

13.2 Advantages of collective communication

Unlike point-to-point communication, which operates on a single target, collective communication operates on a group of targets, which is called **communicator** in MPI. By default, MPI uses the MPI_COMM_WORLD that includes all the processors in the system.

Collective communication not only allow programmers to handle distributed data structures at a higher level, but also can be implemented in more efficient ways and dramatically reduces the bandwidth needed for communication. One can implement the collective communication using minimum spanning tree based on the underlying topology of processors. Usually the processors are organized in a mesh, which makes this mechanism works well. Another way of implementing the collective communication is using pipelining, which only sends out next piece of data when requested. It works like an assembly-line and hides the latency of transferring data.

Slide 4 gives an example of broadcasting n bytes to p processors. The naive broadcasting method needs to send out data $p-1$ times, which consumes huge amount of bandwidth. While using minimum spanning tree, the bandwidth needed for communication approaches to a constant $2(p-1)/p$. This example does not model the contention when transferring data. However in practice, contention can be avoided given the underlying topology of processors.

13.3 Synchronization

MPI uses a barrier to synchronize all the processes within a communicator. When a process reaches a barrier, it will not continue until all other processes in the same communicator reach this barrier. Therefore, the program continues past barrier only after all processes have arrived. Slide 6 shows an example of using the MPI_Barrier() API.

13.4 Communications

MPI has a rich set of collective communication operations. Most of them fall into one of these categories:

- From root task to all others: MPI_Bcast, MPI_Scatter and MPI_Scatterv

- From all tasks to root: MPI_Gather and MPI_Gatherv
- From all to all: MPI_Alltoall and MPI_Allgather
- Reductions: MPI_Reduce and MPI_Scan

We now describe all of them in detail

- **MPI_Bcast:** broadcasts a message (single datum) from the processor with rank "root" to all other processors of the group. Programmer should provide a buffer, the size to be sent, the rank of root, and the communicator (i.e. the group). By default, this API uses a tree-like algorithm to broadcast the message to other processors. The following slides shows typically how to broadcast a set of data to a group of 9 processors, with the data originally on the processor with rank 2. The red arrows show the latency direction and the green arrows show the bandwidth consumption. As you can see, MPI broadcast the data in divide-and-conquer manner, so that multiple transferring can happen simultaneously. This method reduces the bandwidth needs and hides the latency of transferring data.
- **MPI_Scatter & MPI_Scatterv:** MPI_Scatter spreads data from the "root" to all processors in a group. Each processor will receive a chunk of data that has the same size specified by programmer. You can see the effect of scatter on Slide 50. MPI_Scatterv is a variant that allows users to spread different amount of data to each processors in a group by providing an array of sizes. The following slides give an example of scattering an array on to 9 processors.
- **MPI_Gather & MPI_Gatherv:** MPI_Gather gathers data from all processors in a group to the "root" processor. Each processor contributes the same amount of data. You can see how it works on Slide 66. MPI_Gatherv is a variant that allows to collect different amount of data from each processor by specify the sizes in an array. MPI_Scatter and MPI_Gather work as a set of duals. The following slides give an example of gathering data from 9 processors to the root processor with rank 2.
- **MPI_Alltoall:** Sends a distinct message from each processor to every processor. It is very much like each processor carries out a scatter operation on its local data, but everything happens simultaneously, as you can see from Slide 81. This kind of operation is particularly useful for task like matrix transpose. It also has a variable size variant MPI_Alltoallv.
- **MPI_Allgather:** gathers data from all processors in a group and distribute it to all. It can be thought of as MPI_Gather, but where all processors receive the result instead of just "root", as shown in Slide 82. The jth block of data sent from each processor is received by every processor and placed in the jth block of the receive buffer. It also has a variable size variant MPI_Allgatherv.
- **MPI_Reduce:** combines the elements provided in the input buffer of each processor in the group, using the operation *op*, and returns the combined value in the output buffer of the "root" processor. all processors provide input buffers and output offers of the same length, with elements of the same type. MPI provides a set of pre-defined reduce operations. In addition, users may define their own operations that can be overloaded to operate on several data types. Slide 83 shows the effect of MPI_Reduce.
- **MPI_Scan:** Computes the scan (partial reductions) of data on processors in a group, as shown on Slide 84. It is used to perform a prefix reducion on data distributed across the group. The operation returns, in the receive buffer of the processor with rank i, the reduction of the values in the send buffers of processors with rank $0, \ldots, i$ (inclusive). The following slides give an example of reducing a 2D array on 16 processors. Essentially, we first reduce-scatter in columns and then reduce-scatter in rows. Similar to previous examples, the red arrows present latency directions and the green arrows shows the bandwidth consumption.

13.5 Communicator Management

The collective communication in MPI applys on a group of processors, which is named as *communicator*. The base communicator is pre-defined outside of MPI, and is called MPI_COMM_WORLD. Users can create, duplicate and split communicators to form the group as necessary.

- **MPI_COMM_CREATE:** creates a new communicator.
- **MPI_COMM_DUP:** duplicates the existing communicator and returns a new communicator with the same group and copied cached information from the old communicator. However, the new communicator has its own new context.
- **MPI_COMM_SPLIT:** partitions the group associated with the old communicator into disjoint subgroups, one for each value of color. Each subgroup contains all processes of the same color. Within each subgroup, the processes are ranked in the order defined by the value of the argument *key*, with ties broken according to ther rank in the old group. A new communicator is created for each subgroup. Slide 112 gives an example of using MPI_COMM_SPLIT. It shows how five processors are partitioned based on 2 colors and their new rank in corresponding subgroups.