3.1 Lecture Summary

3.1 Single Program Multiple Data (SPMD) Model

Lecture Summary: In this lecture, we studied the *Single Program Multiple Data (SPMD)* model, which can enable the use of a *cluster* of distributed nodes as a single parallel computer. Each node in such a cluster typically consist of a multicore processor, a local memory, and a network interface card (NIC) that enables it to communicate with other nodes in the cluster. One of the biggest challenges that arises when trying to use the distributed nodes as a single parallel computer is that of *data distribution*. In general, we would want to allocate large data structures that span multiple nodes in the cluster; this logical view of data structures is often referred to as a *global view*. However, a typical physical implementation of this global view on a cluster is obtained by distributing pieces of the global data structure across different nodes, so that each node has a *local view* of the piece of the data structure allocated in its local memory. In many cases in practice, the programmer has to undertake the conceptual burden of mapping back and forth between the logical global view and the physical local views. Since there is one logical program that is executing on the individual pieces of data, this abstraction of a cluster is referred to as the Single Program Multiple Data (SPMD) model.

In this module, we will focus on a commonly used implementation of the SPMD model, that is referred to as the *Message Passing Interface (MPI)*. When using MPI, you designate a fixed set of processes that will participate for the entire lifetime of the global application. It is common for each node to execute one MPI process, but it is also possible to execute more than one MPI process per multicore node so as to improve the utilization of processor cores within the node. Each process starts executing its own copy of the MPI program, and starts by calling the mpi.MPI_Init() method, where mpi is the instance of the MPI class used by the process. After that, each process can call the mpi.MPI_Comm_size(mpi.MPI_COMM_WORLD) method to determine the total number of processes participating in the MPI application, and the MPI_Comm_rank(mpi.MPI_COMM_WORLD) method to determine the process' own rank within the range, 0…(S−1), where S=MPI_Comm_size().

In this lecture, we studied how a global view, *XG*, of array *X* can be implemented by S local arrays (one per process) of size, *XL.length* = *XG.length*/S. For simplicity, assume that XG.length is a multiple of S. Then, if we logically want to set XG[i]:=i for all logical elements of XG, we can instead set XL[i]:=L×R+i in each local array, where L = XL.*length*, and R=MPI_Comm_rank(). Thus process 0's copy of XL will contain logical elements XG[0…L−1], process 1's copy of XL will contain logical elements XG[L…2×L−1], and so on. Thus, we see that the SPMD approach is very different from client server programming, where each process can be executing a different program.

Optional Reading:

1. Wikipedia article on the SPMD model

2. Documentation on Open MPI's Java interface

3.2 Point-to-Point Communication

Lecture Summary: In this lecture, we studied how to perform *point-to-point communication* in MPI by *sending* and *receiving* messages. In particular, we worked out the details for a simple scenario in which process 0 sends a string, "ABCD", to process 1. Since MPI programs follow the SPMD model, we have to ensure that the same program behaves differently on processes 0 and 1. This was achieved by using an *if-then-else* statement that checks the value of the rank of the process that it is executing on. If the rank is zero, we include the necessary code for calling MPI_Send(); otherwise, we include the necessary code for calling MPI_Recv() (assuming that this simple program is only executed with two processes). Both calls include a number of parameters. The MPI_Send() call specifies the substring to be sent as a subarray by providing the string, offset, and data type, as well as the rank of the receiver, and a tag to assist with matching send and receive calls (we used a tag value of 99 in the lecture). The MPI_Recv() call (in the *else* part of the *if-then-else* statement) includes a buffer in which to receive the message, along with the offset and data type, as well as the rank of the sender and the tag. Each send/receive operation waits (or is *blocked*) until its dual operation is performed by the other process. Once a pair of parallel and compatible MPI_Send() and MPI_Recv() calls is matched, the actual communication is performed by the MPI library. This approach to matching pairs of send/receive calls in SPMD programs is referred to as *two-sided communication*.

As indicated in the lecture, the current implementation of MPI only supports communication of (sub)arrays of primitive data types. However, since we have already learned how to serialize and deserialize objects into/from bytes, the same approach can be used in MPI programs by communicating arrays of bytes.

Optional Reading:

1. Wikipedia article on an example MPI program (written in C, not Java).

2. Documentation on MPI Send() and MPI Recv() API's (in C/C++, not Java)

3.3 Lecture Summary

3.3 Message Ordering and Deadlock

Lecture Summary: In this lecture, we studied some important properties of the message-passing model with send/receive operations, namely *message ordering* and *deadlock*. For message ordering, we discussed a simple example with four MPI processes, R0,R1,R2,R3 (with ranks 0…3 respectively). In our example, process R1 sends message A to process R0, and process R2 sends message B to process R3. We observed that there was no guarantee that process R1's send request would complete before process R2's request, even if process R1 initiated its send request before process R2. Thus, there is no guarantee of the temporal ordering of these two messages. In MPI, the only guarantee of message ordering is when multiple messages are sent with the same sender, receiver, data type, and tag -- these messages will all be ordered in accordance with when their send operations were initiated.

We learned that send and receive operations can lead to an interesting parallel programming challenge called *deadlock*. There are many ways to create deadlocks in MPI programs. In the lecture, we studied a simple example in which process R0 attempts to send message X to process R1, and process R1 attempts to send message Y to process R0. Since both sends are attempted in parallel, processes R0 and R1 remain blocked indefinitely as they wait for matching receive operations, thus resulting in a classical deadlock cycle.

We also learned two ways to fix such a deadlock cycle. The first is by interchanging the two statements in one of the processes (say process R1). As a result, the send operation in process R0 will match the receive operation in process R1, and both processes can move forward with their next communication requests. Another approach is to use MPI's *sendrecv()* operation which includes all the parameters for the send and for the receive operations. By combining send and receive into a single operation, the MPI runtime ensures that deadlock is avoided because a *sendrecv()* call in process R0 can be matched with a *sendrecv()* call in process R1 instead of having to match individual send and receive operations.

Optional Reading:

1. Wikipedia article on Deadlock

3.4 Lecture Summary

3.4 Non-Blocking Communications

Lecture Summary: In this lecture, we studied non-blocking communications, which are implemented via the MPI_Isend() and MPI_Irecv() API calls. The I in MPI_Isend and MPI_Irecv stands for "Immediate" because these calls return immediately instead of blocking until completion. Each such call returns an object of type MPI_Request which can be used as a handle to track the progress of the corresponding send/receive operation. In the example we studied, process R0 performs an Isend operation with req0 as the return value. Likewise process R1 performs an Irecv operation with req1 as the return value. Further, process R0 can perform some local computation in statement S2 while waiting for its Isend operation to complete, and process R1 can do the same with statement S5 while waiting for its Irecv operation to complete. Finally, when process R1 needs to use the value received from process R0 in statement S6, it has to first perform an MPI_Wait() operation on the req1 object to ensure that the receive operation has fully completed. Likewise, when process R0 needs to ensure that the buffer containing the sent message can be overwritten, it has to first perform an MPI_Wait() operation on the req0 object to ensure that the send operation has fully completed. For convenience, MPI_Waitall() can be invoked on an array of requests to wait on all of them with a single API call.

The main benefit of this approach is that the amount of idle time spent waiting for communications to complete is reduced when using non-blocking communications, since the Isend and Irecv operations can be overlapped with local computations. As a convenience, MPI also offers two additional wait operations, MPI_Waitall() and MPI_Waitany(), that can be used to wait for all and any one of a set of

requests to complete. Also, while it is common for Isend and Irecv operations to be paired with each other, it is also possible for a nonblocking send/receive operation in one process to be paired with a blocking receive/send operation in another process. In fact, a blocking Send/Recv operation can also be viewed as being equivalent to a nonblocking Isend/Irecv operation that is immediately followed by a Wait operation.

Optional Reading:

1. Documentation on MPI Isend() and MPI Irecv() API's (in C/C++, not Java)

3.5 Lecture Summary

3.5 Collective Communication

Lecture Summary: In this lecture, we studied collective *communication*, which can involve multiple processes, in a manner that is both similar to, and more powerful, than multicast and publish-subscribe operations. We first considered a simple *broadcast* operation in which rank R0 needs to send message X to all other MPI processes in the application. One way to accomplish this is for R0 to send individual (point-to-point) messages to processes R1,R2,… one by one, but this approach will make R0 a sequential bottleneck when there are (say) thousands of processes in the MPI application. Further, the interconnection network for the compute nodes is capable of implementing such broadcast operations more efficiently than point-to-point messages. Collective communications help exploit this efficiency by leveraging the fact that all MPI processes execute the same program in an SPMD model. For a broadcast operation, all MPI processes execute an MPI_Bcast() API call with a specified *root* process that is the source of the data to be broadcasted. A key property of collective operations is that each process must wait until all processes reach the same collective operation, before the operation can be performed. This form of waiting is referred to as a *barrier*. After the operation is completed, all processes can move past the implicit barrier in the collective call. In the case of MPI_Bcast(), each process will have obtained a copy of the value broadcasted by the root process.

MPI supports a wide range of collective operations, which also includes *reductions*. The reduction example discussed in the lecture was one in which process R2 needs to receive the sum of the values of element Y[0] in all the processes, and store it in element Z[0] in process R2. To perform this computation, all processes will need to execute a Reduce() operation (with an implicit barrier). The parameters to this call include the input array (Y), a zero offset for the input value, the output array (Z, which only needs to be allocated in process R2), a zero offset for the output value, the number of array elements (1) involved in the reduction from each process, the data type for the elements (MPI.INT), the operator for the reduction (MPI_SUM), and the root process (R2) which receives the reduced value. Finally, it is interesting to note that MPI's SPMD model and reduction operations can also be used to implement the *MapReduce* paradigm. All the local computations in each MPI process can be viewed as *map* operations, and they can be interspersed with *reduce* operations as needed.

Optional Reading:

1. Documentation on MPI Bcast() and MPI Reduce() API's (in C/C++, not Java)

Mini Project 3: Matrix Multiply in MPI

---

miniproject_3.zip

Project Goals and Outcomes

This week we learned how to express Single Program Multiple Data (SPMD) parallelism using MPI. MPI offers high-level APIs for sending messages, receiving messages, and performing collectives across groups of MPI ranks called communicators.

One kernel that can be parallelized using SPMD parallelism is dense matrix-matrix multiplication, in which we multiply two input matrices A and B to produce an output matrix C. The value for cell (i, j) of matrix C is computed by taking the dot product of row i in matrix A and column j in matrix B. A review of dense matrix-matrix multiplication can be found at:

https://www.khanacademy.org/math/precalculus/precalc-matrices/multiplying-matrices-by-matrices/v/matrix-multiplication-intro

In this mini-project, your task is to implement parallel matrix-matrix multiplication using MPI.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

MPI Installation

If you would like to test your solution on your local machine, you will need to install an MPI implementation.

If you are using Ubuntu, you can install OpenMPI with the following commands:

If you are using Linux or Mac OS, we recommend downloading the OpenMPI implementation from:

https://www.open-mpi.org/software/ompi/v2.0/

and following the build instructions in the "User Builds" section of the included INSTALL file. Following installation, you must also add the created OpenMPI bin/ folder to your PATH and the created OpenMPI lib/ folder to your LD_LIBRARY_PATH (on Linux) or your DYLD_LIBRARY_PATH (on Mac OS).

If you are on Windows or find the installation instructions for OpenMPI difficult to follow, we recommend using print statements and the Cousera autograder to test your code. The Coursera autograder automatically links in an OpenMPI implementation for you.

Project Files

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_3.zip at the top of this description, you should see the project source code file at

miniproject_3/src/main/java/edu/coursera/distributed/MatrixMult.java

and the project tests in

miniproject_3/src/test/java/edu/coursera/distributed/MpiJavaTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of MatrixMult.java. You should not change the signatures of any public or protected methods inside of MatrixMult.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of MpiJavaTest.java in the final grading process, so do not change that file or any other file except MatrixMult.java.

Your main goals for this assignment are as follows:

1. Inside MatrixMult.java, implement the parallelMatrixMultiply method to perform a matrix-matrix multiply in parallel using SPMD parallelism and MPI. There are helpful TODOs in MatrixMult.java to help guide your implementation.

Note that because this mini-project requires multiple MPI processes to be spawned, performing local testing using Maven as you normally would will only result in a single MPI rank executing. To execute a local MPI program with 4 MPI ranks we have provided a Maven profile which will handle this automatically for you.

To run the provided Maven profile from the command line, simply use the following command after compiling your solution from the same directory as this mini-project's pom.xml file:

As usual, you can also use the Coursera autograder to automatically run all tests for you, which will handle the creation of multiple MPI ranks.

Project Evaluation

Your assignment submission should consist of only the MatrixMult.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera’s auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. See the Common Pitfalls page under Resources for more details. Also note that for all assignments in this course you are free to resubmit as many times as you like. Please give it a few minutes to complete the grading. Once it has completed,

you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 10% - performance of testMatrixMultiplySmall on 2 cores
- 10% - performance of testMatrixMultiplySmall on 4 cores
- 10% - performance of testMatrixMultiplyRectangular1Small on 2 cores
- 10% - performance of testMatrixMultiplyRectangular1Small on 4 cores
- 10% - performance of testMatrixMultiplyRectangular2Small on 2 cores
- 10% - performance of testMatrixMultiplyRectangular2Small on 4 cores
- 10% - performance of testMatrixMultiplySquareLarge on 2 cores
- 10% - performance of testMatrixMultiplySquareLarge on 4 cores
- 10% - performance of testMatrixMultiplyRectangularLarge on 2 cores
- 10% - performance of testMatrixMultiplyRectangularLarge on 4 cores