CMSC416: Introduction to Parallel Computing

Topic: Advanced MPI
Date: February 13, 2024

Background

In parallel programming, there are two methods to parallelize a program: threads and processes. The main difference between the two is that different threads share the same resource space, while processes each have their own individual resource space. As a consequence, separate threads are able to access relevant data directly, while processes must send and receive relevant data. Although this does not make threads inherently better (different problems have different requirements and hardware constraints), using processes to solve your problem requires more deliberate attention to the transfer of data.

Message Passing Interface (MPI) is an interface standard that allows processes to pass data between each other. There are many open-source implementations of this, such as MPICH, MVAPICH, and OpenMPI. Going forward, we will focus on the implementations that exist in OpenMPI.

Previously, we discussed two functions in OpenMPI – MPI_Send(...) and MPI_Recv(...) – that allow processes to communicate data with each other. These are blocking calls made during point-to-point operations.

There are two main issues with these calls due to these functions being blocking:

1. If two processes are trying to communicate to each other and both decide to call MPI_Send(...) before calling MPI_Recv(...), there will be a **deadlock** as MPI_Send(...) never finishes, and the program will hang.
2. If MPI_Recv(...) is called and a process has yet to receive data, it may have non-blocking operations that it could have spent time computing, but it couldn't because MPI_Recv(...) is a blocking call. **IMPORTANT:** This is not a logistical issue, but rather an efficiency issue.

The first problem can be solved by making sure that communicating processes call MPI_Send(...) and MPI_Recv(...) in different orders. This requires deliberate care and may be difficult to implement in some situations, but it is doable.

The second problem, however, cannot be solved if processes use MPI_Send(...) and MPI_Recv(...). In this section, we will be introducing new non-blocking functions in OpenMPI –

MPI_Isend(...) and MPI_Irecv(...) – to circumvent this problem. We will also explore other methods of communicating data to processes via collective operations.

New OpenMPI Functions

MPI_Isend(...)

This is the non-blocking version of MPI_Send(...). Many of its parameters mirrors that of MPI_Send(...), except for the inclusion of "MPI_Request *request", which is needed in MPI_Wait(...); it's possible to have multiple MPI_Isend so request lets MPI_Wait(...) know which it should apply to.

Function Signature (Courtesy of Abhinav Bhatele Notes):

> int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)
>
> buf: address of send buffer
> count: number of elements in send buffer
> datatype: datatype of each send buffer element
> dest: rank of destination process
> tag: message tag
> comm: communicator
> request: communication request

MPI_Irecv(...)

This is the non-blocking version of MPI_Recv(...). Many of its parameters mirrors that of MPI_Recv, except for the inclusion of "MPI_Request *request", which is needed in MPI_Wait(...); it's possible to have multiple MPI_Irecv so request lets MPI_Wait(...) know which it should apply to.

Function Signature (Courtesy of Abhinav Bhatele Notes):

> int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)
>
> buf: address of receive buffer
> count: maximum number of elements in receive buffer

datatype: datatype of each receive buffer element
source: rank of source process
tag: message tag
comm: communicator
request: communication request

MPI_Wait(...)

This is a blocking call that you put after MPI_Isend(...) or MPI_Irecv(...) after you have completed non-blocking operations to ensure blocking operations can safely be executed. Sometimes, MPI_Wait(...) is omitted after MPI_Isend(...), but for the sake of complete correctness, it should be included.

Function Signature (Courtesy of Abhinav Bhatele Notes):

int MPI_Wait(MPI_Request *request, MPI_Status *status)

request: communication request
status: status object

"MPI_Status *status" gives additional information about the function call, such as whether there was an error. Although production-grade code should utilize this information in case of potential unforeseen errors, you may ignore this parameter using MPI_STATUS_IGNORE.

MPI_Waitall(...)

MPI_Waitall(...) is an alternative to calling chained multiple MPI_Wait(...) that you want to run in case you made multiple MPI_Isend(...)/MPI_Irecv(...) calls. Uses array_of_requests[] which holds an array of requests, to know what MPI_Isend(...)/MPI_Irecv(...) calls to apply MPI_Waitall(...) to.

Function Signature (Courtesy of Abhinav Bhatele Notes):

int MPI_Waitall(int count, MPI_Request array_of_requests[], MPI_Status *array_of_statuses[])

count: length of array_of_requests[]
array_of_requests[]: array of requests
array_of_statuses[]: array of status object

MPI_Test(...)

This is the non-blocking version of MPI_Wait(...). Later in this section, we will explain how this can improve efficiency in a program over MPI_Wait(...).

Function Signature (Courtesy of Abhinav Bhatele Notes):

```
int MPI_Test( MPI_Request *request, int *flag, MPI_Status *status )
```

request: communication request
flag: operation completion state
status: status object

MPI_Waitany(...)

This takes in a list of requests and waits until one of the specified requests completes.

Function Signature (Courtesy of Abhinav Bhatele Notes):

```
int MPI_Waitany( int count, MPI_Request array_of_requests[], int *indx, MPI_Status *
status )
```

count: length of array_of_requests[]
array_of_requests[]: array of requests
indx: index of request in array_of_requests that completed first
status: array of status object

MPI_Waitsome(...)

This takes in a list of requests and waits for some of the specified requests to complete, providing details on completed requests along the way.

Function Signature (Courtesy of Abhinav Bhatele Notes):

```
int MPI_Waitsome( int incount, MPI_Request array_of_requests[], int *outcount,
        int array_of_indices[], MPI_Status *array_of_statuses )
```

incount: length of array_of_requests[]
array_of_requests[]: array of requests
outcount: count of completed requests

array_of_indices: array of indices of requests in array_of_requests[] that have completed
array_of_statuses: array of status object

Structure Between MPI_Isend(...)/MPI_Irecv(...) & MPI_Wait(...)/MPI_Test(...)

Before we continue on, it is important to understand how the new functions we have introduced work together. In a typical MPI program, you will likely see these functions used a structure similar to this:

MPI_Isend(...)/MPI_Irecv(...)

// non-blocking operations

MPI_Wait(...)/MPI_Test(...) along with appropriate flag handling code

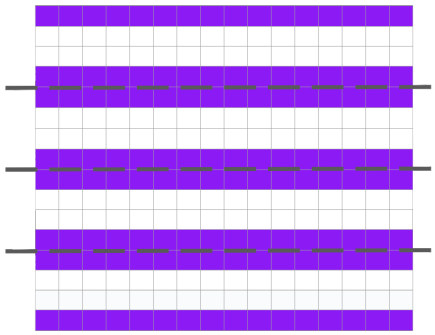
// operations that require MPI_Isend(...)/MPI_Irecv(...) to complete

IMPORTANT: It's possible to have nothing in the non-blocking operations, however, doing so defeats the purpose of using non-blocking functions MPI_Isend(...)/MPI_Irecv(...). In that case, you may want to consider simply using blocking functions MPI_Send(...) and MPI_Recv(...).

Example

Previously, we covered how we would use MPI_Send(...) and MPI_Recv(...) on a 2D stencil computation. Let's use a 2D stencil computation to see how these new functions work in practice. **NOTE:** There are simplifications within the code example to make this example easier to understand.

Consider this 2D stencil:



In this 2D stencil that is 16 X 16, there are 16 rows of data (N) being divided among 4 processes (p), or N/p; this equates to 4 rows per process (numbered 0…n-1). For every computation, every cell requires its neighboring cells to determine its new cell; this yields the need to communicate with ghost cells (purple) via MPI if neighboring cells are in a different process. In this particular 2D stencil, every process must communicate with two other processes – the one "above" it and the one "below" it with two MPI_Isend(...) and two MPI_Irecv(...). Assuming a priori wrap-around, the top-most and bottom-most processes will communicate with each other in addition to their direct neighboring process.

IMPORTANT: Make sure you know if your programming language stores matrix elements in **row-major order** (row elements stored contiguously in memory) or **column-major order** (column elements stored contiguously in memory). In the code examples below, we assume row-major ordering since C uses row-major order. If you use a programming language like Fortran, which uses column-major ordering, be sure to copy row elements into a column array before passing data along.

Here is a simplified naive version of how computations might be done:

```
int main(int argc, char *argv) {
        ...
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Irecv(&data1, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        MPI_Isend(&data3, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Isend(&data4, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        MPI_Waitall(…);

        compute();

        ...
}
```

Upon inspection, two things might stand out in this code:

1. MPI_Irecv(...) is called before Isend(...)
2. There are no non-blocking operations being called i.e. compute() is run only after MPI_Waitall(...)

The first notable detail is actually not an issue. While using MPI_Send(...) and MPI_Recv(...) in this way would result in a deadlock, MPI_Isend(...) and MPI_Irecv(...) are non-blocking calls, so no deadlock will occur. If anything, this might actually optimize your code because if you called MPI_Isend(...) first, the receiving process may have not called MPI_Irecv(...) and would not be able to receive relevant data when available; calling MPI_Irecv(...) first ensures that data from MPI_Isend(...) are received as soon as possible.

The second issue, while not program breaking, defeats the purpose of using non-blocking calls MPI_Isend(...) and MPI_Irecv(...) as opposed to just their blocking function call counterparts. This is because no non-blocking operations are computed between MPI_Isend(...)/MPI_Irecv(...) and MPI_Waitall(...).

Although it would be nice to simply put compute() before the MPI_Waitall(...), there are complications that arise; while this would be fine for the middle 2 rows ((N/p) - 2) in each process that has all the data it needs within its own process, this would be a problem for the ghost cells since these cells rely on having received data from neighboring ghost cells to finish their respective computations. To solve this problem we can modify our program to compute the middle rows in each process as the non-blocking operation and the ghost rows after Waitall(...).

Here is what the modification would look like:

```
int main(int argc, char *argv) {
        ...
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Irecv(&data1, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        MPI_Isend(&data3, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Isend(&data4, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        compute_middle_rows();

        MPI_Waitall(…);

        compute_ghost_rows();
        ...
}
```

From here, we might be tempted to use MPI_Wait(...) after both MPI_Irecv(...) and do ghost row computations before calling MPI_Isend(...) since MPI_Isend(...) won't affect computations within current process. Doing this naively would lead to MPI_Isend(...) sending overwritten data to neighboring processes. In order to resolve this, we could copy over the values at ghost rows to new variables and pass that instead so overwritten data won't get passed in MPI_Isend(...). This would potentially improve runtime (assuming speed of copies are fast) at the cost of some memory overhead.

Here is a possible implementation of this strategy:

```
int main(int argc, char *argv) {
        ...
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);

        MPI_Irecv(&data1, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Irecv(&data2, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        compute_middle_rows();

        MPI_Wait(...) // for data1
        // variable that holds copy of data3 in copy_data3

        compute_top_ghost_row(); // this would overwrite data3, hence why need to pass
        in copy_data3 to subsequent calls

        MPI_Wait(...) // for data2
        // variable that holds copy of data4 in copy_data4

        compute_bottom_ghost_row(); // this would overwritedata4, hence why need to
        pass in copy_data4 to subsequent calls

        MPI_Isend(&copy_data3, 16, MPI_DOUBLE, (rank-1)%4, 0, ...);
        MPI_Isend(&copy_data4, 16, MPI_DOUBLE, (rank+1)%4, 0, ...);

        MPI_Waitall(…); // For sends
        ...
}
```

Believe it or not, there are still ways to potentially improve the efficiency of the program. Recall that MPI_Wait(...) is a blocking call; what would happen if data2 is available before data1 in the

code above? In this case, data2 would not be able to be processed and data1 would also not be able to be processed since it hasn't been received yet, leading to the program being idle until data1 is received. A solution to this would be using a non-blocking call MPI_Test(...) for each respective MPI_Irecv(...) call and using the flag property to determine which ghost cell to process first based on which data is received first.

Throughout this example, we were able to reason through the problem and make adjustments to incrementally improve the efficiency of the program. This process of analysis and reiteration is typical practice when designing parallel programs. As an exercise, I implore you to think about other ways to further improve this program or alternative methods to achieve similar results.

Collective Operations

So far, we have been looking at point-to-point operations, where pairs of processes communicate with each other. We will now examine how we can use collective operations to interact with all processes within a communicator. First, let's get a better understanding of MPI communicators.

MPI communicators are groups (each with a uniquely assigned tag), created by the MPI runtime that hold a set of processes numbered 0…n-1. In many cases and examples we have used, we have defaulted to MPI_COMM_WORLD which is the default global communicator, commonly referred to as the global rank as it contains all processes in a program. However, you are not limited to this global communicator, and can create sub-communicators via functions MPI_Comm_split, MPI_Cart_create, MPI_Group_incl. Why would you want to do this? In the case of a 2D stencil that is 2D decomposed, you may want to do this as an easy way to share data among processes in a row by assigning each row its own sub-communicator.

Collective Operation Functions

Now that we understand what MPI Communicators are, let's explore some of the common functions used to interact with processes within communicators.

MPI_Barrier(...)

This function is used as a synchronization point, blocking all processes within a communicator until they have reached this routine.

Function Signature (Courtesy of Abhinav Bhatele Notes):

```
int MPI_Barrier( MPI_Comm comm)
```

MPI_Bcast(...)

This function broadcasts data from the root to all processes within the communicator. Note that individual processes do not need to call any receive functions to get broadcasted data.

Function Signature (Courtesy of Abhinav Bhatele Notes):

> int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)