# SpatialMPI: Message Passing Interface (MPI) for GIS Applications

Puri, S. (2019). SpatialMPI: Message Passing Interface for GIS Applications. *The Geographic Information Science & Technology Body of Knowledge* (2nd Quarter 2019 Edition), John P. Wilson (Ed.). DOI: 10.22224/gistbok/2019.2.6

## Abstract

MPI is a widely used message passing library for writing parallel programs. The goal of Message Passing Interface is to establish a portable, efficient, and flexible standard for message passing that will be widely used for writing message passing programs. This chapter motivates the need for using Message Passing Interface (MPI) for implementing GIS applications. It introduces MPI data types and communication functions. Then, it presents new spatial data types and operations on them using MPI. Finally, it presents pseudocode for parallelizing a range query problem and spatial domain decomposition in GIS.

## Definitions

MPI: MPI stands for Message Passing Interface. It is a standard library interface for writing parallel programs.

Process: A program in execution or a running program. Operating system schedules different processes to run on a CPU.

HPC: High Performance Computing.

MBR: Minimum Bounding Rectangle.

Parallelism: Using multiple processes or threads to solve a problem.

Distributed Memory: A network of inter-connected processors each of which has its own private memory. For instance, if a processor needs to access memory of another processor, then it has to perform communication. One way to do so is via passing messages using MPI.

## Message Passing Interface

MPI (Message Passing Interface) is a portable, standard interface for writing parallel programs using a distributed-memory programming model [2]. It is widely used for writing parallel applications, particularly in science and engineering domains. MPI programs run on a variety of systems including small compute clusters and supercomputers. The interface contains functions for communication and computation among processes. Popular MPI implementations are MPICH, MVAPICH and Open MPI. Communication among processes is required because processes do not share memory among themselves. MPI supports parallel I/O, i.e. reading and writing data by multiple processes in parallel to increase the performance. MPI programs are typically written in C/C++ and Fortran.

MPI datatypes: MPI provides a rich set of predefined datatypes [1]. For example, MPI_INT is for an integer type and MPI_FLOAT is for a single precision floating point type. However, new derived types are required to represent spatial data that are supported by Open Geospatial Consortium standard. MPI allows user-defined types that can be used to define Cartesian coordinates.

Simple MPI program: An MPI program is shown below to introduce the basic MPI functions. In addition to the four MPI functions used below, there are functions for sending and receiving a message between a pair of processes. When an MPI program is run, programmer provides the number of processes to be created as shown in Table 1. MPI_COMM_WORLD is a communicator object that contains all the processes.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
  // numProcs is the number of processes, processId is the id of the MPI process
  int numProcs, processId;

  // Initializes MPI by using the command line arguments
  MPI_Init(&argc, &argv);

  /*The number of processes is stored in the variable numProcs
    numProcs gets set using the command-line argument provided while running the program
  */
  MPI_Comm_Size(MPI_COMM_WORLD, &numProcs);

  // Each process gets a unique ID which is stored in the variable processId
  MPI_Comm_Rank(MPI_COMM_WORLD, &processId);

  printf("Process Id = %d Number of processes = %d \n", processId,  numProcs);

  // Insert Algorithm here from Section on Range Query parallelization.

  // terminates MPI
  MPI_Finalize();

  return 0;
}
```

**Table 1. Compiling and running an MPI program.**

| Compiling MPI program | Running MPI program |
|---|---|
| The **mpicc** command compiles the program and creates an executable a.out.<br>mpicc mpiprogram.c | The **mpirun** command runs the executable a.out using 4 MPI processes ( –np 4 option specifies the number of processes to be used in the MPI program).<br><br>mpirun –np 4 ./a.out |

When this program is executed, it will print 0, 1, 2 and 3 for *processId* and *numProcs* will be 4.

## Extending MPI to support spatial data types and operations

Spatial data types: MPI allows creation of user-defined data types. Implementing spatial data types is motivated by the fact that user-defined data types and operations not only enhance the abstraction and reusability of a system, but also performance in case of MPI in the presence of hardware support. Some of the data types are shown in Table 2. For instance, MPI_Rect can be defined as a contiguous type of 4 floating point numbers. Based on these spatial types, additional compound types such as multi-point and multi-line can be defined as well.

Table 2. Spatial data types and reduction operators in SpatialMPI [4].

| **Spatial Type (MPI Datatype)** | **Spatial Reduction in MPI based GIS application** | |
|---|---|---|
| | **MPI Operator** | Explanation |
| MPI_POINT<br>MPI_LINE<br>MPI_RECT | MPI_MIN<br>MPI_MAX<br>MPI_UNION | Min point (bottom left corner)<br>Max point (top right corner)<br>MBR of Geometric Union |

MPI Send and Receive operations: In a distributed memory system, processes do not share memory. When multiple processes are running in parallel in such a system, they perform some computations followed by exchanging messages. For example, if process A sends an integer to process B, then using MPI, process A executes MPI_Send function. Process B executes MPI_Recv function. Table 3 contains the function prototypes for send and receive functions.

In MPI_Send function, a process needs to specify the destination process, the datatype, and count. If the datatype is a scalar quantity, then the count is 1. If the data to be sent is an array, then the length of the array needs to be specified as count. In the MPI_Recv function, the status argument also needs to be specified. In addition to functions for communication, MPI also supports functions for computation among processes. MPI_Reduce function reduces values on all processes to a single value. For example, if processes 0 and 1 have values 10 and 20 respectively, then using sum reduction, the result will be 30. Instead of sum, other operations like min and max can also be used. The reduction function can be extended to handle GIS functions as well. For more details, readers are encouraged to refer to books and tutorials [3].

Communication using spatial data types: In MPI, if a process wants to share an item with another function, a send function needs to be invoked from a sending process and receive function needs to be invoked by a receiving process. Table 3 shows how to express message passing between two MPI processes using Point data. Point is defined as a user-defined data type using C structure. In some applications, there can be more than one pair of MPI send and receive function calls. To distinguish different types of such messages, each message has integer valued tag associated with it. MPI send and receive have similar syntax with an exception that MPI receive has an additional parameter called *MPI_Status*. Status can be used to get information about the received message.

Table 3. Spatial data communication from MPI Process 0 to Process 1.

| **MPI Process 0** | **MPI Process 1** |
|---|---|
| Point p(3.0, 5.0);<br>int tag = 0; | Point q;<br>int tag = 0; |

| | |
|---|---|
| // Sending a single point to Process 1<br>MPI_Send(&p, 1, MPI_POINT, 1, tag,<br>MPI_COMM_WORLD);<br><br>// Sending 3 points<br>Point pArray[3] = {{1,3}, {2,1}, {4,5}};<br>MPI_Send(pArray, 3, MPI_POINT, 1, tag,<br>MPI_COMM_WORLD);<br><br>/* MPI Send function for reference<br>*MPI_Send( void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator) */* | MPI_Status status;<br>// Receiving a single point from Process 0<br>MPI_Recv(&q, 1, MPI_POINT, 0, tag,<br>MPI_COMM_WORLD, &status);<br><br>// Receiving 3 points<br>Point qArray[3];<br>MPI_Recv(qArray, 3, MPI_POINT, 0, tag,<br>MPI_COMM_WORLD, &status);<br><br>/* MPI Receive function for reference<br>*MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status) */* |

**Spatial reduction**

Collective Reduction Operators for Spatial types: With derived data types, existing MPI reduction operators like MPI_MIN do not work. The min operator can be used to find the line/rectangle with minimum length/area among processes. To implement it, a user-defined function is required that generates an MBR of geometric union of rectangles [4]. It should be noted that reduction is limited to associative and commutative functions. Spatial data types and reductions make MPI spatial-aware.

Figure 1 illustrate spatial reduction. Table 4 shows an example. Here, MPI_Reduce function parameters written in order correspond to the input rectangle, output rectangle, count, data type, reduction operation, root process and communicator. Each process specifies an input rectangle and the output rectangle (MBR of geometric union of input rectangles) is computed and is available at root process with id as 0.

**Table 4. Example usage in C language.**

```
Spatial type: Rectangle *in_rect, *out_rect;
Reduction operator: MPI_UNION
MPI_Reduce(in_rect, out_rect, 1, MPI_RECT, MPI_UNION, 0, MPI_COMM_WORLD);

/* MPI_Reduce function prototype for reference
MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype,
MPI_Op op, int root, MPI_Comm communicator);
*/
```

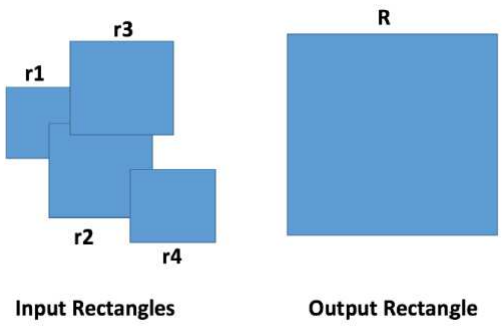


Figure 1: Spatial Reduction on Minimum Bounding Rectangles.

## Range Query

Example: Two dimensional range query using a query rectangle.

**Input**: Base layer containing N rectangles and a given query rectangle.

**Output**: All rectangles from the base layer overlapping with the query rectangle should be returned. List of overlapping rectangles denoted by variable outputList.

Integer variables: startIndex, endIndex, numRectangles, processId, numProcs;

**Algorithm**: Each MPI process reads N rectangles in the base layer and calls rectangle intersection function on a subset of the input rectangles given by startIndex and endIndex.

**Steps**

1. Compute startIndex, endIndex for each MPI process.
   startIndex = (processId * N) / numProcs;
   endIndex = ((processId + 1) * N) / numProcs;

2. Each process computes rectangle Intersection (input rectangle, query rectangle).
   The input rectangles overlapping with query rectangle are added to the output list.

```
for ( int i = startIndex; i<endIndex; i++)
{
    if( intersect(R_i, q) )
        outputList.add(R_i);
}
```

3. Process 0 aggregates the result from all processes

```
if( processId == 0 )
{
    Rectangle queryResults[N];
    for(int pId = 1; pId < numProcs; pId++)
    {
        int numOutput = N;
        MPI_Recv(queryResults, N, MPI_RECT, pId, tag, MPI_COMM_WORLD,
                                                                &status);
    }
}
```

```
else
{
    // find the number of rectangles in output list for sending it to process 0.
    int numOutput = outputList.length;
    MPI_Send(outputList, numOutput, MPI_RECT, 0, tag, MPI_COMM_WORLD);
}
```

In Step 3, Process 0 uses receive function to gather data from other processes in a loop. It should be noted that a receiving process (with rank 0) does not know how many query rectangles it will receive from other processes. The exact number is only known by each sending process (with rank 1 to numProcs – 1) at runtime. However, the maximum count can be no more than N. As such, the count variable in MPI send function is set as N. The exact number can be extracted from the status variable by the receiver. Step 3 can be implemented using *MPI_Gatherv* function as well instead of multiple point-to-point messages [2].

## Spatial domain decomposition using MPI

Domain decomposition requires data stored in an arbitrary order to be partitioned among processes where each partition is assigned to an MPI process. For parallel processing in GIS, spatial data is often partitioned using a uniform or adaptive grid. Each partition is assigned to unique process to carry out spatial computation using the data in the partition. Operations like range query can be parallelized in an efficient manner if the data is already partitioned among processes.

In this section, we will discuss the partitioning of a set geometries using a uniform grid with C cells. Let us assume there are N polygons and P processes. Each process has access to only a portion of the data stored in disk as shapefiles. Figure 2 shows the distribution in a uniform grid with 4 cells and 2 processes. We can see that geometries read by an MPI process locally is projected to a grid by process P0 and P1. Shaded area represents tasks assigned to each process. Here, a task means computational work associated with a grid cell. The first process reads 4 blue geometries and a red geometry. The second process reads 4 red geometries in the third and fourth quadrants. After partitioning, each cell in a grid should contain only those geometries that overlap with it. In order to do so, the first step is to determine the four corners of the grid. This can be done by a reduction operation. A union of all the geometries would produce the dimension of the universe as shown in Figure 1.

Applying spatial union using MPI reduce: For spatial partitioning, the dimension of the universe needs to be determined which is an MBR spatially containing all the geometries read by all the processes. New MPI UNION operator on MBRs can be used to find the grid dimensions from the union of MBRs generated by individual processes during data partitioning. This can be done in two steps. In the first step, each process computes an MBR containing all the geometries it reads. The second step requires spatial reduction using union operation that we covered earlier. Then the dimension of grid cells is computed based on the number of partitions required.

Exchanging spatial data using MPI Communication: C cells can be assigned to P processes in a round-robin fashion. For instance, in Figure 2, cell I and II is assigned to the first process and the remaining cells are assigned to the second process. A geometry from a 2D map may belong to one or more cells and since we know the bounding boxes of all the grid cells, each process can determine to which cell(s) a locally-read geometry belongs to. As such, based on the X and Y coordinates, each geometry is mapped to a cell in a 2-dimensional uniform grid. Since geometries belonging to a grid cell may not be locally available to the cell owner, a

communication step is required to complete the domain decomposition. As shown in the figure, P0 needs to send geometry A to P1. This ensures that P1 gets all the geometries that are in its spatial domain.

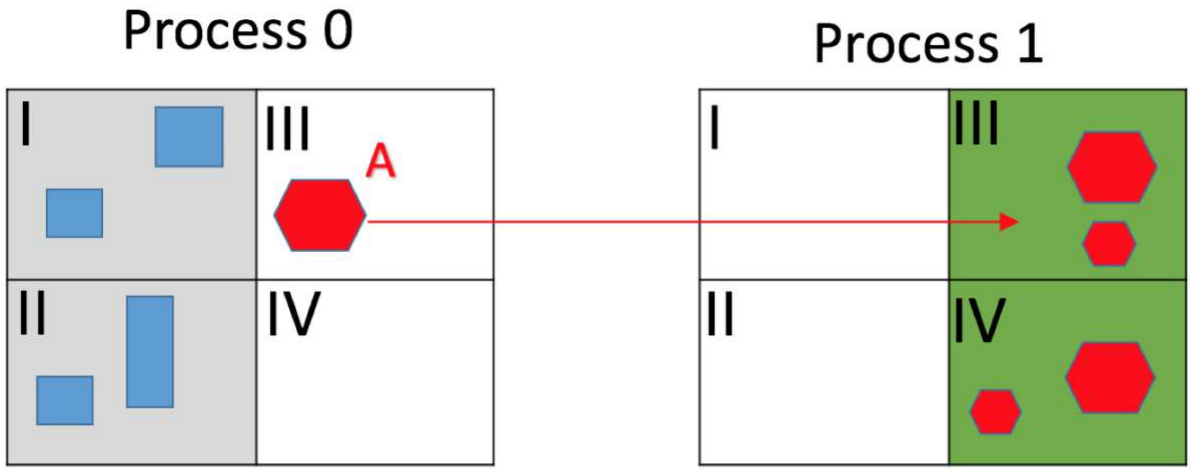


Figure 2. Need for communication of shape A during spatial data decomposition among two MPI processes with ranks 0 and 1. Rank 0 calls MPI_Send function and Rank 1 calls MPI_Recv function.

To summarize, we covered a few MPI functions here. MPI is a distributed memory programming model. MapReduce is also a distributed memory programming model. However, MPI is different when compared to MapReduce. MPI requires that the parallelism is coded explicitly by the programmer. A programmer has to devise strategy to partition the problem (domain decomposition) into tasks so that a process is responsible for a subset of tasks. Potential challenges in MPI programs include deadlocks and load imbalance.

## Learning Objectives

- Define Message Passing Interface (MPI)
- Parallelize range query using MPI
- Define spatial data types, communication and union operation using MPI
- Partition spatial data using MPI

## Learning Questions

1. What is the output of MPI_Reduce if Process P0 contains r1 = {1, 2, 3, 4} and P1 contains r2 = {4, 4, 5, 5}. Let us assume the order of coordinates is minX, maxX, minY, maxY.

2. Write MPI Send and Receive code for sending 2 rectangles from process 1 to process 0. Compile and run the program.

3. In the Range Query problem, if there are 101 rectangles, 1 query rectangle and 4 MPI processes, describe how the problem can be parallelized. Hint: calculate the start index and end index for each process.

4. Assume that the domain decomposition has already been done and there are P processes and C grid cells. Given a query rectangle, provide a strategy to implement Range Query problem in this scenario.

## Acknowledgement

This work is partly supported by the National Science Foundation Grant No. 1756000.

## References

[1] Gropp, W. D., Gropp, W., Lusk, E., & Skjellum, A. (1999). *Using MPI: Portable parallel programming with the message-passing interface* (Vol. 1). MIT press.

[2] MPI: A Message-Passing Interface Standard Version 3.1 (2015), Message Passing Interface Forum. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[3] Blaise Barney, *MPI tutorial*, https://computing.llnl.gov/tutorials/mpi/

[4] Puri, S., Paudel, A., & Prasad, S. (2018) *MPI-Vector-IO: Parallel I/O and Partitioning for Geo-Spatial Vector Data*, International Conference on Parallel Processing (ICPP).

[5] Puri, S., & Prasad, S. (2015) *A Parallel Algorithm for Clipping Polygons with Improved Bounds and a Distributed Overlay Processing System Using MPI*, 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).